EFFICIENT SCHEDULING AND ANALYSIS FOR COMPLEX REAL-TIME SYSTEMS

Shareef Ahmed

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

Chapel Hill
2025

Approved by:

James H. Anderson

Benjamin Berg

Parasara Sridhar Duggirala

Joël Goossens

Shahriar Nirjon

ABSTRACT

Shareef Ahmed: Efficient Scheduling and Analysis for Complex Real-Time Systems
(Under the direction of James H. Anderson)

Real-time systems typically have two conflicting requirements: computations must provably satisfy application-specific timing constraints, and the system must efficiently utilize the underlying processing resources to meet constraints related to size, weight, power, and cost. Satisfying timing constraints while ensuring high resource utilization requires the formal analysis of such systems to be as tight as possible. Unfortunately, obtaining tight analysis for even simple real-time systems is computationally intractable, often leading to inefficient resource utilization. In today's artificial-intelligence-powered real-time systems, this challenge is further exacerbated by complex workloads involving parallel real-time tasks, precedence constraints, and non-processing shared resources. Moreover, these workloads are often deployed on multiprocessor platforms augmented with hardware accelerators, introducing additional complexities.

The goal of this dissertation is to take a step toward tighter analysis of real-time systems exhibiting complex runtime behaviors due to precedence constraints, shared resources, and parallelism. The analyses presented here focus on two types of resource-management algorithms: scheduling algorithms and synchronization algorithms. The specific contributions of this dissertation are threefold.

First, this dissertation presents a polynomial-time tight response-time analysis for a practically common class of sequential tasks under *global earliest-deadline-first* (G-EDF) scheduling and its variants. It also presents an exact response-time analysis for such systems that can be performed in pseudo-polynomial time. Furthermore, the analysis is extended to systems with precedence constraints.

Second, this dissertation presents an optimal suspension-based locking protocol for mutual exclusion sharing under *first-in-first-out* (FIFO) scheduling. It also establishes new lower-bound results to show that existing asymptotically optimal suspension-based locking protocols for mutual exclusion under G-EDF and its variants are nearly optimal.

Finally, this dissertation presents response-time analysis for parallel tasks with co-scheduling requirements, known as gang tasks, both with and without precedence constraints. It also provides intractability

results for scheduling gang tasks—even in systems with soft timing constraints—and demonstrates that G-EDF and FIFO are not optimal for such systems.

To my family

ACKNOWLEDGEMENTS

Firstly, I express my gratitude to Almighty Allah for giving me the strength and ability to complete this dissertation.

The completion of this dissertation would not have been possible without the support of many people. Firstly, I would like to express my deepest gratitude to my advisor, Jim Anderson, for his unwavering guidance, support, and mentorship throughout my doctoral journey. His dedication to research, keen attention to detail, and tireless efforts in carefully reviewing and refining my papers have been instrumental in shaping this dissertation. I am especially thankful for his patience, generosity with his time, and allowing me flexibility during times when I was challenged by different needs. Jim placed a great deal of trust in me, something I must admit felt overwhelming at times, and granted me tremendous freedom in my work. It has been a privilege to work under his supervision.

I would also like to sincerely thank my dissertation committee members—Ben Berg, Parasara Sridhar Duggirala, Joël Goossens, and Shahriar Nirjon—for their time, thoughtful feedback, and valuable suggestions throughout the course of my research. I am grateful for their careful reading of my dissertation and their insightful comments, which helped improve the quality and clarity of this work. I am especially thankful to Sridhar and Joël for their generous support in writing recommendation letters for my academic job search. I am also grateful to Nirjon and Sridhar for their many advices that were very helpful.

During my doctoral study, I had the privilege to work with many UNC real-time systems people. I am exceedingly grateful to all my co-authors: Joshua Bakita, Jingyuan "Leo" Chen, Saujaus Nandi, Sims Osborne, Stephen Tang, Zelin "Peter" Tong, Sizhe Liu, Denver Massey, and Rohan Wagle. I also extend my appreciation to other past and present members of the UNC Real-Time Systems group: Syed Ali, Tanya Amert, Joseph Goh, Catherine Nemitz, Nathan Otterness, Sergey Voronov, Ming Yang, Tyler Yandrofski, and Hongyi Zhang. I would like to thank UNC real-time systems group alumnus Kecheng Yang for his help during the academic job search process. I am also thankful to the broader systems community at UNC, including faculty and students, for fostering an intellectually stimulating environment, particularly during the weekly cyber-physical-systems lunch.

I would also like to thank the dedicated staff of the UNC Department of Computer Science for their invaluable support throughout my time in the program. In particular, I am grateful to Denise Kenney, Soji Marcel Weeks, Rafael Zaldivar, Missy Wood, Tatyana Davis, and late Bil Hays for their responsiveness, kindness, and assistance in navigating both academic and administrative matters.

My stay in Chapel Hill would have been tremendously difficult without the support of many friends. I would like to thank Bashima Islam, Tamzeed Islam, Jisan Mahmud, Md Asadullah Turja, Nahid Sultana Ruku, Taksir Hasan Majumder, Mahathir Monjur, and Md Mohaiminul Islam for their friendship and support during my time at Chapel Hill. I am especially grateful to Bashima, Tamzeed, Jisan, and Turja for their support during my early days in Chapel Hill. Their support eased me to settle in and adapt to a new culture. Bashima and Tamzeed let me stay at their place for some days, and even cooked my favorite dishes. I would also like to thank Shamim Hasan Zahid and Turja for their patience and generosity in teaching me how to drive a car.

I lived in Greensboro for a significant portion of my doctoral study and I am deeply thankful to the friends who supported me during that time. In particular, I would like to thank Md Arifur Rahman Khan (Topu), Rajata Suvra Chakrovorty, Mohammad Bakhtiar Uddin, Israt Jahan, Sajib Aninda Dhar, and Md Ataullah Nuri for their friendship, encouragement, and support. I am especially grateful to Shamimul Islam and Ayesha Khan for their kindness in helping take care of my daughter during times when both my wife and I were occupied with work. Their generosity and care made a meaningful difference and are deeply appreciated.

I would also like to thank Md. Saidur Rahman, who supervised my research at Bangladesh University of Engineering and Technology. He laid the foundation for my analytical thinking and approach to proving theorems, which greatly helped me during my doctoral studies. I am also grateful to all my teachers at BUET, especially Mohammad Kaykobad, Mohammad Sohel Rahman, Eunus Ali, Anindya Iqbal, Mahmuda Naznin, and Atif Hasan Rahman, for their support and encouragement to pursue a Ph.D. A special thanks is due to the Islamic Center of Greensboro for making past few years' Ramadan comfortable.

Above all, I am forever grateful to my parents, Md Abdur Razzaque and Nazmim Zahan, whose unwavering love, sacrifices, and prayers have shaped who I am today. I am thankful to my siblings, Tania Sharmin and Shafeen Ahmed, for their constant support and encouragement throughout this journey. I also extend my heartfelt appreciation to my in-laws, Md Jalil Howlader and Rehana Begum, for their kindness, support, and belief in me. I am grateful to all my extended family members for their encouragement and good

wishes over the years. Most importantly, I thank my wife, Tahniat Afsari, for her boundless love, strength, and patience——for standing by my side and, quite literally, keeping me alive through the most challenging times. Honestly, I would be nowhere near completing this dissertation without her support. My deepest joy comes from my daughter, Saniyat Shareef, whose laughter, curiosity, and presence have been a constant source of happiness and inspiration.

Funding for this research was provided by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, CPS 2038960, CNS 2151829, and CPS 2333120, ARO grant W911NF-20-1-0237, ONR grant N00014-20-1-2698, and a dissertation completion fellowship from the UNC Graduate School.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
CPU	Central Processing Unit
CS	Critical Section
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
EDF	Earliest-Deadline-First
EDZL	Earliest Deadline Zero Laxity
FIFO	First-In-First-Out
FP	Fixed Priority
FPGA	Field-Programmable Gate Arrays
GEL	Global-EDF-Like
GPU	Graphics Processing Unit
HRT	Hard Real Time
ILP	Integer Linear Program
JLDP	Job-Level Dynamic-Priority
JLFP	Job-Level Fixed-Priority
LCM	Least Common Multiple
LLF	Least Laxity First
OS	Operating System
Pi	Priority Inversion
rp	Restricted Parallelism
SWap-C	Size, Weight, Power, and Cost
SRT	Soft Real Time
SMT	Simultaneous Multithreading
TPU	Tensor Processing Unit
WCET	Worst-Case Execution Time
WCER	Worst-Case Execution Requirement

CHAPTER 1: INTRODUCTION

Due to the physical and thermal limitations of increasing clock speeds, hardware manufacturers have primarily focused on designing parallel computing platforms over the past few decades. Consequently, today's computing landscape is dominated by computing platforms equipped with multiple processing cores, such as multicore machines, graphics processing units (GPUs), tensor processing units (TPUs), field-programmable gate arrays (FPGAs), *etc.* These hardware advances have fueled unprecedented advances in artificial intelligence (AI) algorithms that use highly parallel computation to enable autonomous functionalities. Real-time systems are also evolving to adopt such autonomous functionalities, with active industry efforts in the pursuit of autonomous vehicles as a notable example. However, enabling autonomous functionalities in many real-time systems presents two significant challenges that do not typically arise in general-purpose computing systems, as discussed below.

Certification. Many real-time systems, such as autonomous vehicles, interact with the physical world in safety-critical ways, where incorrect operation can lead to catastrophic consequences. For example, an autonomous vehicle may collide with an obstacle if the obstacle is not accurately detected. Moreover, to avoid a collision, the obstacle must also be detected within a certain time limit after sensor data is received. Thus, real-time systems must be both functionally and temporally correct. To ensure safe operation, these systems should be *certified* for both functional and temporal correctness before their real-world deployment.

Resource efficiency. Real-time systems are typically resource-constrained, as they must operate within specific *size*, *weight*, *power*, and *cost* (SWaP-C) constraints. For example, autonomous vehicles should be affordable to ensure their widespread adoption. Today, approximately 40% of a car's cost is due to electronic components [Caranddriver, 2020]. Thus, cost can be reduced by minimizing the necessary computing resources. Similarly, using fewer computing resources can reduce size, weight, and power consumption, resulting in improved fuel efficiency.

Unfortunately, achieving both certifiability and high resource efficiency is challenging due to inherent trade-offs. For example, having a significantly large number of processors simplifies the certification of

temporal correctness, as each computation can be statically assigned to a dedicated set of processors. In contrast, using fewer processors makes it more difficult to certify timing correctness due to the pessimism in formal analysis that arises when many computations must compete for limited compute resources. This dissertation aims to advance the certification of temporal correctness in real-time systems while reducing the number of required processing resources. To that end, it considers a wide range of workloads found in today's autonomous real-time systems.

We begin this chapter by providing an introduction to real-time systems and discussing features common in modern autonomous real-time systems. We then describe the specific problems addressed in this dissertation. Finally, we present the thesis statement, summarize the key contributions, and provide an outline for the remainder of the dissertation.

1.1 Real-Time Systems

A computer system typically requires *functional correctness*, meaning it produces the correct output for every valid input. In addition to being functionally correct, real-time systems also require *temporal correctness*, which means that each correct output must be produced within an application-specific time limit, typically called a *deadline*. At first glance, temporal correctness may appear to be a matter of devising efficient algorithms and their implementations. However, many *resource management* decisions in a computer system can also affect its temporal correctness. Such decisions are necessary when multiple tasks (*e.g.,* processes or threads) compete for a set of shared processors and other resources such as shared data, memory bandwidth *etc.* For example, in an autonomous vehicle, even a highly efficient object detection task may not be able to detect surrounding objects if the task is not given sufficient CPU time before the detection deadline. Thus, when designing a real-time system, resource management decisions must be carefully made, and whether such decisions ensure temporal correctness must be validated before system deployment.

Validating temporal correctness. Validating temporal correctness of a real-time system is typically considered an orthogonal problem to validating functional correctness. Commonly, a two-step approach is used to validate temporal correctness of a system. In the first step, an appropriate model of the system is developed by abstracting different computations as tasks, determining relationships among these tasks, characterizing properties of the hardware platform (*e.g.,* processor speeds), *etc.* Additionally, different model parameters,

such as task execution times, are determined. In the second step, the model is analyzed to validate temporal correctness using *schedulability analysis* with respect to the chosen resource management algorithms.

Considered resource management algorithms. In this dissertation, we focus on the effect of the following two types of resource management algorithms on the temporal correctness of real-time systems:

- **Scheduling algorithms.** A *scheduling algorithm* is an *operating system* (OS) procedure that decides which tasks to execute at any time instant. Although every computer system has a scheduling algorithm, the purpose of a scheduling algorithm for real-time systems and general-purpose computer systems differs. In a real-time system, the goal of a scheduling algorithm is to provide timing guarantees for each task in all possible execution scenarios. In contrast, a scheduling algorithm usually aims to optimize average-case behavior in general-purpose computer systems.

- **Synchronization algorithms.** *Synchronization algorithms* are used to arbitrate access to shared resources that have certain *sharing constraints*. An example is a *shared data object* that has a *mutual exclusion* (*mutex*) sharing constraint to prevent *race conditions*, which occur when multiple tasks access the shared data object concurrently.

To illustrate how these algorithms affect temporal correctness and how such correctness can be validated, we introduce a classic real-time task model, widely known as the sporadic task model.

1.1.1 Sporadic Task Model

Since the seminal paper by Liu and Layland [Liu and Layland, 1973], the sporadic task model has been widely recognized as a fundamental workload model for real-time systems. Under the sporadic task model, a system consists of a set of *sporadic* tasks. Each task is *sequential*, meaning it consists of a single thread of execution. Moreover, all tasks are *independent*, *i.e.,* the execution of one task does not depend on any other. We use the terms *sporadic task* and *sequential task* interchangeably.[1]

Each task τ_i releases a potentially infinite sequence of *jobs* (task instances) $\tau_{i,1}, \tau_{i,2}, \ldots$. The *period* of a task τ_i, denoted by T_i, represents the minimum separation time between consecutive jobs of τ_i. In other words, the inverse of the period defines the maximum rate at which jobs of a task can be invoked. A task τ_i is

[1]The term *sporadic* refers to the minimum separation time between consecutive jobs, while the term *sequential* refers to the single-threaded (non-parallel) execution of a task instance. Traditionally, the term sporadic is commonly used when referring to such tasks.

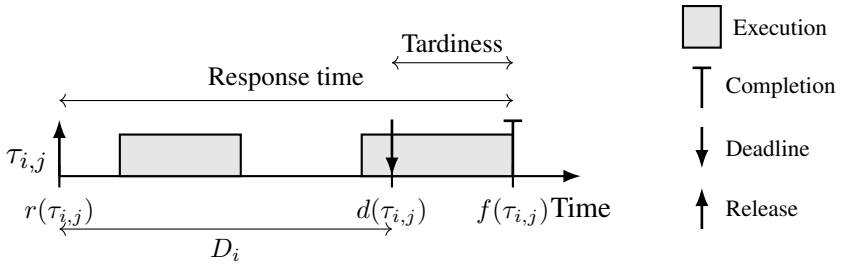


Figure 1.1: Illustration of a job.

called *periodic* if the separation time between its consecutive jobs is exactly T_i. A system is called periodic if all its tasks are periodic. Each task τ_i has a *worst-case execution time* (WCET), denoted by C_i, which represents the maximum execution time that a job of τ_i may take to complete on a unit-speed processor. Each task τ_i also has a *relative deadline*, denoted by D_i. Task τ_i has an *implicit deadline* if $D_i = T_i$, a *constrained deadline* if $D_i \leq T_i$, and an *arbitrary deadline* if no relationship between D_i and T_i is assumed. Task τ_i's *utilization* is given by $u_i = \frac{C_i}{T_i}$, which indicates the worst-case amount of processing capacity that jobs of τ_i may demand per time unit in the long term. The *total utilization* of all tasks, denoted by U_{tot}, is the sum of all per-task utilizations.

The *release time* and *completion time* of job $\tau_{i,j}$ are denoted by $r(\tau_{i,j})$ and $f(\tau_{i,j})$, respectively. The *absolute deadline*, or simply *deadline*, of a job, denoted by $d(\tau_{i,j})$, is defined as $d(\tau_{i,j}) = r(\tau_{i,j}) + D_i$ (see Figure 1.1). Thus, the deadline of a job is D_i time units after its release time. The *response time* of $\tau_{i,j}$ is $f(\tau_{i,j}) - r(\tau_{i,j})$, which is the duration between the job's completion time and release time. The *response time*, also known as the *worst-case response time*, of task τ_i is $\sup_j\{f(\tau_{i,j}) - r(\tau_{i,j})\}$. Any job that does not complete execution by its deadline is said to *miss* its deadline. If a job that misses its deadline is still allowed to complete execution, then $f(\tau_{i,j}) > d(\tau_{i,j})$. Such a job is called *tardy*, and its *tardiness* is $f(\tau_{i,j}) - d(\tau_{i,j})$. In contrast, a job that finishes execution by its deadline has zero tardiness. Thus, task τ_i's tardiness is defined as $\sup_j\{0, f(\tau_{i,j}) - d(\tau_{i,j})\}$.

In an *instantiation* of a task system, the release time and execution time of each job are known. In other words, an instantiation represents an *execution scenario* of the task system. There can be infinitely many instantiations of a task system.

Hard vs. soft real-time systems. Hard real-time (HRT) systems require every job of each task to meet its deadline; therefore, no job can be tardy in an HRT system. Missing a deadline in such systems, such as an engine control system, can lead to catastrophic consequences. In contrast, soft real-time (SRT) systems can

tolerate occasional deadline misses without severe consequences. One criterion for the temporal correctness of an SRT system is that each task must have a bounded response time (and hence, bounded tardiness).[2] An example of an SRT system is a streaming application, where a certain number of video frames are expected to be processed per time unit. However, occasionally dropping or delaying some frames does not significantly degrade the quality of service. Note that a system may contain both HRT and SRT components; for example, an autonomous vehicle's object detection component is HRT, whereas its infotainment component is SRT.

Since the inception of real-time systems as a research area, HRT systems have been the primary focus of real-time systems research. However, SRT systems also warrant attention due to their widespread presence in industrial applications. A recent survey of industrial real-time systems reported that approximately 67% of the systems surveyed include SRT components [Akesson et al., 2022]. Moreover, allowing deadline misses often enables deploying systems on less powerful hardware, *e.g.,* on fewer processors. This is because strictly meeting deadlines leads to conservative designs, where all processors may be fully utilized only in worst-case scenarios (*e.g.,* when many jobs arrive in the system concurrently) but remain underutilized most of the time.

1.1.2 Hardware Model

The sporadic task model has been extensively studied in the context of uniprocessor scheduling. Since the advent of multicore technology, research focus has shifted toward shared-memory multiprocessor systems. In the multiprocessor case, various configurations are possible based on processor speeds, leading to the following taxonomy of multiprocessor models [Pinedo, 2008].

- **Identical.** All processors operate at the same speed at all times. Without loss of generality, each processor is assumed to have a unit speed.
- **Uniform.** The speed of a processor can be different from that of others, but the speed remains consistently the same.
- **Unrelated.** A processor's speed can vary with respect to the task it is executing, *i.e.,* a processor can execute jobs of different tasks at different speeds.

[2]Other criteria for temporal correctness of SRT systems exist, *e.g.,* at least a specified percentage of deadlines must be met [Jain et al., 2002], or x out of any y consecutive jobs of each task must meet their deadlines [Hamdaoui and Ramanathan, 1995; Koren and Shasha, 1995; West and Poellabauer, 2000].

According to the above classification, the identical and uniform multiprocessor models are special cases of the uniform and unrelated models, respectively. This dissertation focuses on task scheduling on identical multiprocessors.

Heterogeneity due to multiple compute elements. The uniform and unrelated multiprocessor models capture heterogeneity within a single multiprocessor platform. However, heterogeneity can also result from the integration of multiple distinct *compute elements* (CEs). For instance, a compute platform comprised of both a multicore machine and a GPU is considered heterogeneous.

1.1.3 Scheduling Algorithms

When multiple tasks are executed on a multiprocessor platform, a *scheduler* determines which jobs run at any given time. The algorithm used by the scheduler to make these decisions is called a *scheduling algorithm*. Based on whether the scheduler allows *migration*, *i.e.,* moving a job's execution between different processors, there are two major scheduling strategies on multiprocessors:[3] *global* scheduling and *partitioned* scheduling. Global scheduling allows tasks to migrate among all processors, whereas partitioned scheduling prohibits any migration. A hybrid approach, known as *clustered* scheduling, groups processors into clusters and permits migration only within each cluster.

Schedulers can be divided into two classes based on how they select jobs to execute: *table-driven* and *priority-driven* schedulers. A table-driven scheduler uses an offline-generated table of jobs to select which job to schedule at runtime. This table is finite in length and is cyclically repeated at runtime. In contrast, priority-driven schedulers assign priorities to jobs according to a defined set of rules. At any time instant, a priority-driven scheduler selects the jobs with the highest priorities for execution. Based on how priorities are assigned, priority-driven schedulers can be further divided into three subclasses.

- **Task-level fixed priority schedulers.** A task-level fixed-priority scheduler, or simply a *fixed-priority* (FP) scheduler, assigns the same priority to all jobs of a task. An example of FP schedulers is the well-known *rate-monotonic* (RM) scheduler. Under RM scheduling, tasks are prioritized in order of increasing periods, *i.e.,* tasks with smaller periods have higher priorities.

- **Job-level fixed priority schedulers.** Under a job-level fixed-priority (JLFP) scheduler, a job's priority always remains the same. However, different jobs of the same task can have different priorities. Note

[3]Migration is not applicable on a uniprocessor.

that any FP scheduler is also a JLFP scheduler. The *earliest-deadline-first* (EDF) scheduler is a JLFP scheduler but not an FP scheduler. Under EDF scheduling, jobs with earlier deadlines have higher priorities.

- **Job-level dynamic priority schedulers.** Under a job-level dynamic-priority (JLDP) scheduler, a job's priority can change dynamically with respect to time. Thus, JLDP schedulers generalize JLFP schedulers. The *least-laxity-first* (LLF) scheduler is a JLDP scheduler but not a JLFP scheduler. Under LLF scheduling, job priorities are determined at runtime based on the *laxity* (also known as *slack*) of a job. The laxity of a job at a time instant is the amount of time remaining until its deadline minus its remaining execution time. Since laxity changes continuously over time (especially for running jobs, as their remaining execution time decreases), a job's priority can change frequently.

When referring to a multiprocessor scheduler, we use the term 'global', 'partitioned', or 'clustered' (abbreviated as 'G', 'P', or 'C') in the prefix of the scheduler's name to denote whether the corresponding scheduler is a global, partitioned, or clustered scheduler, respectively. For example, G-EDF, P-EDF, and C-EDF refer to global, partitioned, and clustered EDF scheduling, respectively.

Preemptive vs. non-preemptive scheduling. Under preemptive scheduling, an executing job can be interrupted at any time and resumed later. Under a priority-driven preemptive scheduler, a job is typically preempted when a higher-priority job arrives or when the priority of a job increases. In contrast, non-preemptive scheduling does not allow a job to be interrupted once it starts executing; it must run to completion before another job can begin on the same processor. Limited preemptive scheduling is a hybrid of these two approaches, allowing preemption only at specific points during a job's execution. Preemptivity can also be defined at the job level: whether a job can be preempted depends on whether it is currently executing in a preemptive or non-preemptive section. In this dissertation, we focus on preemptive scheduling.

Work-conserving vs. non-work-conserving scheduling. In addition, a scheduler can be either *work-conserving* or *non-work-conserving*. A work-conserving scheduler ensures that no processor is idle at any time instant if there exists a ready job that is not currently scheduled.[4] In contrast, a non-work-conserving scheduler may leave a processor idle even when there is a ready but unscheduled job.

[4]This statement applies to global scheduling. More generally, under clustered work-conserving scheduling, no processor within a job's cluster can be idle when the job is ready but unscheduled.

Continuous vs. discrete time. When time is assumed to be *continuous*, scheduling decisions, job releases, and completions can occur at any real-valued time. However, real-time systems typically operate under a discrete-time model, where the system time unit is determined by the clock resolution. Throughout this dissertation, we assume time to be discrete, with the unit of time set to 1.0. All scheduling decisions are made at integer points in time. We also assume all task parameters to be integers. Even if a job completes its execution at a fractional time instant, no new job is scheduled until the next integer time point. A job completes execution at time t if it executes for the last time during $[t-1, t)$. A job completes execution before time t if it completes at or before $t-1$.

1.1.4 Schedulability, Feasibility, and Optimality

We now define the term *schedulability*, which is specific to the used scheduling algorithm.

Definition 1.1 (Schedulability)**.** A system is *HRT-schedulable* (resp., *SRT-schedulable*) on M processors under a scheduling algorithm $\mathcal{A}$ if and only if each task's response time is at most its relative deadline (resp., bounded) when scheduled on M processors under the algorithm $\mathcal{A}$. ◀

Thus, if a system is HRT-schedulable (resp., SRT-schedulable) under a scheduling algorithm, then every job meets its deadline (resp., has bounded response time) when the system is scheduled under that scheduling algorithm. We now define the term *feasibility*, which indicates whether a system is schedulable by some scheduling algorithm.

Definition 1.2 (Feasibility)**.** A system is *HRT-feasible* (resp., *SRT-feasible*) on M processors if and only if there exists a schedule under which the system is HRT-schedulable (resp., SRT-schedulable) on M processors. ◀

Note that, by Definition 1.2, any HRT-feasible system is also SRT-feasible. Thus, the following holds.

$$\text{Task system } \Gamma \text{ is HRT-feasible} \Rightarrow \text{Task system } \Gamma \text{ is SRT-feasible} \tag{1.1}$$

For any implicit-deadline sporadic task system, the following theorem is known about HRT-feasibility.

Theorem 1.1. *[Baruah et al., 1996] A task system of implicit-deadline sporadic tasks is HRT-feasible on M identical processors if and only if $\forall\, i : u_i \leq 1$ and $U_{tot} \leq M$ hold.*

The necessary conditions in Theorem 1.1 are intuitive. The per-task utilization constraint ensures that no individual task demands more processing capacity than a single processor can provide. The total utilization constraint ensures that the aggregate demand of all tasks does not exceed the total capacity of all processors. Note that determining HRT-feasibility for arbitrary-deadline task systems is NP-hard, even when tasks are periodic [Leung and Merrill, 1980]. However, Theorem 1.1, together with (1.1), implies that the SRT-feasibility of any sporadic task system can be determined using the following corollary.

Corollary 1.1. *A task system of sporadic tasks is SRT-feasible on M identical processors if and only if $\forall\, i : u_i \leq 1$ and $U_{tot} \leq M$ hold.*

In this dissertation, we mostly focus on systems that are SRT-feasible (or equivalently HRT-feasible for the implicit-deadline case). Finally, an optimal scheduling algorithm for a class of systems can schedule any system in that class.

Definition 1.3 (Optimality)**.** A scheduling algorithm $\mathcal{A}$ is *HRT-optimal* (resp., *SRT-optimal*) for a class of task system if and only if every HRT-feasible (resp., SRT-feasible) system of that class of task system is HRT-schedulable (resp., SRT-schedulable) under the algorithm $\mathcal{A}$. ◀

Note that a scheduling algorithm that is optimal for one class of systems may not be optimal for another. For example, EDF is an HRT-optimal algorithm for sporadic task systems on uniprocessors but not on multiprocessors (even when the number of processors is only two) [Dertouzos, 1973].

Capacity loss. A set of M unit-speed processors supplies M units of processing capacity per time unit. However, it may not always be possible to schedule every system with total utilization at most M under a given scheduling algorithm. This results in *capacity loss* (also called *utilization loss*), where some processing capacity remains unused. Capacity loss can be inherent to a particular task model (or a class of task systems). For example, consider a class of HRT task systems where relative deadlines are a small fraction of task periods. In such cases, no task system in that class with total utilization of M is HRT-feasible. Thus, capacity loss is inherent to that class of task systems. Even when capacity loss is not inherent to the task model, it can still occur due to the choice of scheduling algorithm. For instance, using a non-optimal scheduling algorithm may render a feasible task system unschedulable, resulting in *algorithmic* capacity loss.

Global EDF and global first-in-first-out scheduling. This dissertation focuses on a class of SRT-optimal JLFP schedulers that contains the global EDF (G-EDF) and *global first-in-first-out* (G-FIFO) schedulers.

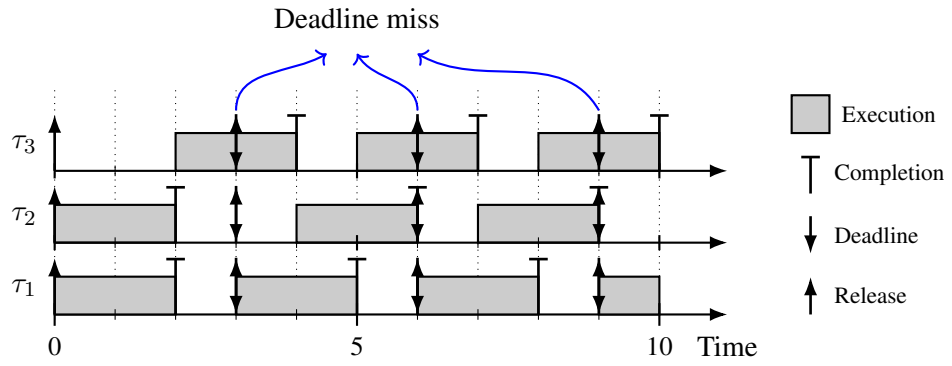


Figure 1.2: A G-EDF schedule of three implicit-deadline tasks each with $T_i = 3.0$ and $C_i = 2.0$ on two processors.

G-EDF (resp., G-FIFO) assigns job priorities according to job deadlines (resp., release times). Being global schedulers, G-EDF and G-FIFO allow job migrations. At any time instant, these schedulers schedule the M highest-priority jobs (if that many exist). Note that, despite being SRT-optimal [Devi and Anderson, 2005; Leontyev and Anderson, 2007], G-EDF and G-FIFO are not HRT-optimal for implicit-deadline tasks.

Example 1.1. Figure 1.2 shows a G-EDF schedule of three implicit-deadline tasks on two processors. The period and WCET of each task are 3.0 and 2.0, respectively. In the schedule, ties are broken according to task index. Every job of task τ_3 misses its deadline. However, tardiness of each of these jobs is 1.0. Note that the schedule is also the same under G-FIFO scheduling. ◀

Note that no partitioned scheduler can be SRT-optimal (or HRT-optimal) due to the *bin-packing* problem inherent in partitioning. Additionally, FP schedulers are not SRT-optimal, as the response time of the lowest-priority task in some SRT-feasible task systems can grow unboundedly under FP scheduling.

Example 1.2. Figure 1.3 shows a G-FP-schedule of three implicit-deadline tasks with $T_i = 3.0$ and $C_i = 2.0$ on three processors. Tasks are prioritized according to task indices. The response times of the jobs of τ_3 grow unboundedly. ◀

The role of deadlines in G-EDF for SRT systems. Although G-EDF uses deadlines to schedule jobs, meeting all job deadlines is not required in SRT systems. In such systems, meeting all deadlines is desirable but not mandatory. Alternatively, deadlines can be viewed merely as a prioritization mechanism.

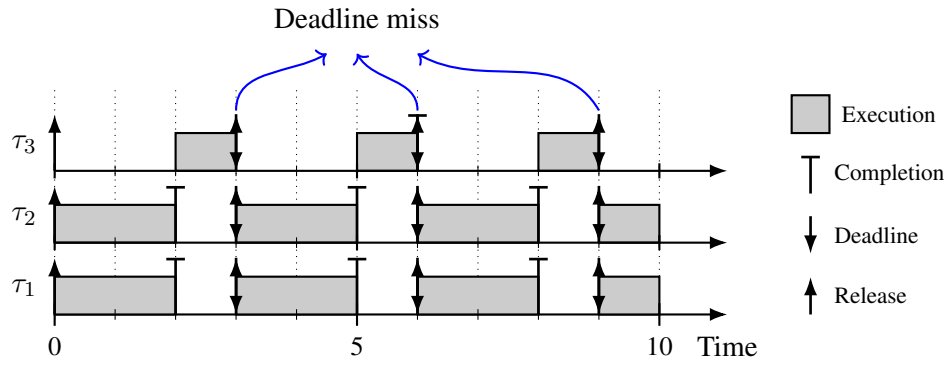


Figure 1.3: A G-FP schedule of three implicit-deadline tasks each with $T_i = 3.0$ and $C_i = 2.0$ on two processors.

1.1.5 Schedulability Test and Response-Time Analysis

A *schedulability test* is an algorithm that determines whether a task system is schedulable under a scheduling algorithm. A test is said to be *exact* for a scheduling algorithm $\mathcal{A}$ if it deems a task system to be schedulable if and only if the system is indeed schedulable under $\mathcal{A}$. In contrast, a *sufficient* schedulability test may classify a task system as unschedulable even when it is schedulable. This can lead to capacity loss, as the test may unnecessarily reject some schedulable systems.

Response-time analysis. *Response-time analysis* is commonly used to test schedulability in real-time systems. The goal of response-time analysis is to determine the worst-case response time of each task. Using response-time analysis, the schedulability of a system can be tested by simply checking whether the derived worst-case response time meets the (HRT or SRT) timing constraints of the system. However, deriving the worst-case response time of a task is generally NP-hard under many scheduling algorithms [Eisenbrand and Rothvoß, 2008, 2010]. Therefore, most response-time analyses provide an upper bound on the worst-case response time, often called a *response-time bound*, of a task. The *tightness* of a response-time analysis is defined as follows.

Definition 1.4 (Tight Bound)**.** A response-time analysis for a class of systems under a scheduling algorithm $\mathcal{A}$ is *tight* if and only if, for any processor count, there exists a system in that class such that any increase in a task WCET makes the system unschedulable under $\mathcal{A}$, and the derived response-time bound of at least one task equals its worst-case response time. The corresponding response-time bound is called a *tight* response-time bound. ◀

Definition 1.4 requires a task system with sufficiently large total utilization so that the task system does not remain schedulable under the considered scheduler if any task's utilization is increased.

We now define an *exact response-time analysis* as one that yields the true worst-case response times. For consistency, we also refer to the worst-case response times as *exact response-time bounds*.

Definition 1.5 (Exact Bound)**.** A response-time analysis for a class of systems under a scheduling algorithm $\mathcal{A}$ is *exact* if and only if, for any system in that class that is HRT-schedulable under $\mathcal{A}$, the derived response-time bound of each task τ_i equals its worst-case response time. The corresponding response-time bound is called an *exact* response-time bound. ◀

By the above definitions, an exact response-time analysis is also a tight response-time analysis.

Why does an exact test or bound matter? The lack of exact response-time bounds leads to resource wastage in both HRT and SRT systems by preventing full utilization of available resources. In an HRT system, a non-exact response-time analysis (and thus a non-exact schedulability test) may incorrectly classify some schedulable systems as unschedulable. As a result, additional processors may be required to satisfy timing constraints, even though they are not strictly necessary. In SRT systems, an exact schedulability test under G-EDF and its variants can often be performed efficiently in polynomial time by checking a simple condition. However, imprecise (*i.e.,* overly conservative) response-time bounds may be too loose to be practically useful. For example, in an SRT system, such as a video processing application [Kenna et al., 2011], task outputs may be stored in a buffer that is read at a steady rate to simulate HRT-like completion behavior. In such systems, the buffer size is determined based on the worst-case response time. An overly pessimistic bound would require a disproportionately large buffer, much of which would remain unused at runtime.

Efficient analysis. Like any algorithm, a schedulability test or response-time analysis is considered efficient if it has polynomial time complexity. Additionally, in the real-time systems literature, algorithms with *pseudo-polynomial* time complexities are also considered efficient. Note that the running time of a pseudo-polynomial time is polynomial in the largest numerical value and the length of the input [Garey and Johnson, 1990].

1.1.5.1 Emerging Real-Time Systems

In the uniprocessor era, real-time systems typically consisted of sporadic tasks. Figure 1.4(a) illustrates such a system with an example control system of an airplane. However, as shown in Figure 1.4(b), today's real-time systems—such as those in autonomous vehicles—are becoming increasingly sophisticated to

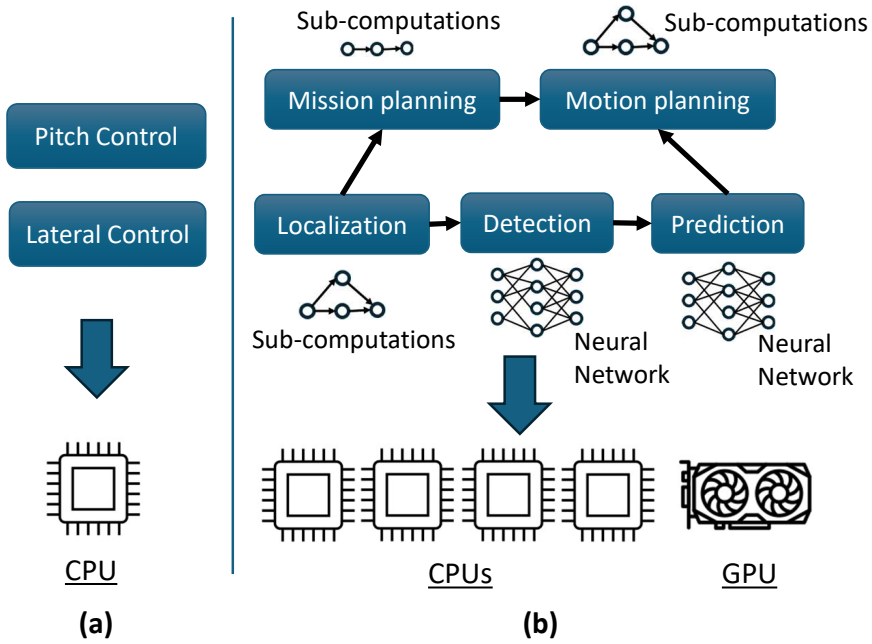


Figure 1.4: (a) Real-time systems in past vs. (b) present [Kato et al., 2018]. Depicted workloads are a simplification of real systems.

support AI-based autonomous functionalities. To model such systems, more general task models have been developed, with the sporadic task model being a special case. In the following, we describe three additional complex features present in modern real-time systems that are also considered in this dissertation.

Dataflow dependencies. Real-time systems with autonomous functionalities typically rely on sensors to repeatedly sample their surrounding environment. The sensor data are then processed to generate actuation commands. For example, an autonomous vehicle uses sensors such as cameras and LiDAR to capture images and point clouds, respectively. These data are then processed to detect objects, plan motion, and steer the vehicle accordingly. As a result, tasks in such systems are interconnected through dataflow dependencies. This creates *processing graphs* with *precedence constraints*, where a task's job can be released only after other jobs have produced the required data and completed execution. Commonly, these processing graphs take the form of *directed acyclic graphs* (DAGs), which we refer to as DAG tasks. DAG tasks naturally represent data processing in many applications, where source tasks (with no incoming edges) gather sensor data and sink tasks (with no outgoing edges) perform actuation. Similarly to sequential tasks, DAG tasks can be *sporadic* or *periodic* based on the separation time between their consecutive instances.

Gang tasks. Many autonomous real-time systems contain compute-heavy tasks that perform inferences using deep neural networks (DNNs). For example, in Figure 1.4(b), the detection task uses a DNN to detect objects.

Such DNN-based tasks require hardware accelerators, such as GPUs, to achieve sufficiently fast execution. Execution on GPUs has certain characteristics: multiple threads form a single task, and many threads of the task are co-scheduled on the required number of GPU cores. These multithreaded, co-scheduled tasks are abstracted as *gang* tasks.[5] Gang tasks can also be *sporadic* or *periodic*.

Shared resources. Many real-time systems include non-CPU resources that are shared among multiple tasks, such as shared data objects, GPUs, *etc.* For example, autonomous vehicle systems involve a large number of shared data objects across AI-based modules, which may be accessed either directly by application code or indirectly through dependency libraries and frameworks. A shared resource typically has a sharing constraint, meaning that only certain protected accesses to the resource are allowed. For instance, a shared data object may be subject to a mutex constraint, which prevents jobs from accessing the data simultaneously. To ensure protected access, a synchronization mechanism, *e.g.,* a *locking protocol*, can be employed to arbitrate access to shared resources.

1.2 Limitations of the State-of-the-Art

We now discuss some limitations of state-of-the-art scheduling algorithms, synchronization mechanisms, and their analyses. First, we examine the limitations of existing response-time analyses that can be applied to any SRT-feasible sequential or DAG tasks scheduled under G-EDF, G-FIFO, and their variants. Next, we consider suspension-based mutex sharing and highlight the limitations of existing locking protocols. Finally, we discuss the lack of existing work addressing the scheduling of SRT sporadic gang tasks and HRT gang tasks with precedence constraints.

1.2.1 Non-Tight Bounds for Sequential and DAG Tasks

Sequential tasks. To schedule sequential tasks, one approach to achieving reasonably small response times for all SRT-feasible task systems is to employ a scheduler that is HRT-optimal for scheduling implicit-deadline tasks. Many such schedulers are known, which ensure that each job $\tau_{i,j}$ finishes execution by time $r(\tau_{i,j}) + T_i$ [Baruah et al., 1995, 1996; Anderson and Srinivasan, 2004; Regnier et al., 2011]. All of these schedulers are JLDP schedulers, which may require numerous priority changes per job to achieve

[5]Gang tasks were originally introduced to reduce inter-process communication bottleneck in multiprocessors [Ousterhout, 1982].

this bound. Consequently, these schedulers often incur high runtime overheads, leading to significant overhead-related capacity loss [Brandenburg et al., 2008; Brandenburg and Anderson, 2009]. As a result, JLFP schedulers are more common in practice; 59%, 26%, and 17% respondents in a recently industrial survey reported using FP, FIFO, and EDF schedulers, respectively [Akesson et al., 2022]. Among these, G-EDF, G-FIFO, and their variants are particularly notable, as they are SRT-optimal while maintaining relatively low runtime overheads.

Since the seminal work by Devi and Anderson [Devi and Anderson, 2005], many response-time bounds have been developed that apply to any SRT-feasible sporadic task system under G-EDF, G-FIFO, and their variants. Unfortunately, these response-time bounds are neither exact nor known to be tight. In fact, extensive evaluations on synthetic workloads demonstrate that observed response times are often significantly smaller than the existing bounds. This suggests that it is unlikely for a task system to exist that demonstrates the tightness of these bounds. At the time of writing this dissertation, deriving an exact or tight response-time bound for the class of SRT-feasible sporadic task systems—even via an exponential-time algorithm—has remained an open problem for over two decades.

In addition to their lack of tightness, existing response-time bounds under G-EDF, G-FIFO, and their variants share a common characteristic: for systems with total utilizations close to the number of processors, the bounds increase as the processor count increases. This behavior can be observed in the response-time bound presented in the following theorem, which remains the best-known closed-form bound to date.

Theorem 1.2 ([Devi and Anderson, 2005])**.** *For any sporadic task system scheduled under G-EDF on M processors, the response time of a task τ_i is at most*

$$T_i + \frac{\mathcal{C}_{M-1} - \min_k\{C_k\}}{M - \mathcal{U}_{M-2}} + C_i, \tag{1.2}$$

where $\mathcal{C}_\ell = \sum_{\ell \text{ highest}} C_j$ and $\mathcal{U}_\ell = \sum_{\ell \text{ highest}} u_j$.

To see how (1.2) can grow with respect to the processor count, consider a pathological scenario where $\mathcal{C}_{M-1} \approx (M-1) \cdot \max_k\{C_k\}$ and $\mathcal{U}_{M-2} \approx (M-2)$. Substituting these values into (1.2), the response-time bound can be as large as $T_i + \frac{M-2}{2} \max_k\{C_k\} + C_i$. Due to the term $\frac{M-2}{2} \max_k\{C_k\}$, the response-time bound increases linearly with respect to the processor count for systems with $\mathcal{C}_{M-1} \approx (M-1) \max_k\{C_k\}$ and $\mathcal{U}_{M-2} \approx (M-2)$. This dependency on the number of processors is particularly problematic for systems that require a large number of processors. Despite the introduction of non-closed-form bounds [Erickson et al.,

2014; Valente, 2016], all such bounds empirically exhibit a similar (though not necessarily linear) dependency on the processor count. However, observed response times for synthetic workloads do not demonstrate such a dependency. These issues lead to the following questions: *Can we derive a* tight *or* exact *response-time analysis for all SRT-feasible sequential tasks under G-EDF, G-FIFO, and their variants? Additionally, is it possible to derive a response-time bound that does not increase with respect to the processor count?*

DAG tasks. The temporal correctness of a DAG task typically requires satisfying constraints on its *end-to-end response time*, or simply response time, which represents the time elapsed between sensing at a source node and the corresponding actuation at a sink node. Deriving response-time bounds for DAG tasks has been a major focus in real-time systems research over the past decade; recent work includes [He et al., 2022; Sun et al., 2021; Amert et al., 2019; Zhao et al., 2020; Wang et al., 2019; Voronov et al., 2021b; Nasri et al., 2019]. Unfortunately, no existing work provides an exact response-time analysis under any scheduler that avoids any capacity loss, *i.e.,* no exact response-time bounds are known under any SRT-optimal scheduler. In fact, only one scheduling approach, called the *offset-based* scheduling algorithm, is known to be SRT-optimal. Under offset-based scheduling, each DAG is converted into an "equivalent" independent sporadic task system. (We provide a detailed review of this approach in Chapter 2.) The transformation involves assigning appropriate task *offsets*—the release times of jobs relative to the release time of a job of the source task—to ensure that DAG precedence constraints are respected. However, a task's offset depends on analytically derived response-time bounds of its ancestor tasks. Under the offset-based approach, a response-time bound of a DAG is derived by taking the largest sum of the response-time bounds of all tasks on any path of the DAG. This results in an accumulation of the pessimism inherent in the response-time bounds of sequential tasks. Additionally, even devising exact response-time bounds for DAG tasks under offset-based scheduling is unlikely to yield much benefit. This is because the sink node's offset—computed based on analytical response-time bounds of predecessor tasks—is included in the DAG task's response-time bound. Thus, the following question arises: *Can we devise an SRT-optimal scheduling algorithm and a corresponding exact response-time analysis for DAG tasks?*

1.2.2 Mysteries Around Optimal Suspension-Based Locking Protocols

To access a shared resource, a job may need to wait—either by *suspending* or *spinning*—before it can access the resource. This waiting leads to *priority-inversion blocking* (pi-blocking) when a higher-priority

job is delayed while a lower-priority job executes. Pi-blocking increases the response times of higher-priority jobs. Moreover, the pi-blocking time incurred by a job must be determined and accounted for in response-time analysis to ensure that the resulting bound includes this delay. To reduce response times, the goal of designing a real-time locking protocol is to minimize the maximum pi-blocking time incurred by a job. A locking protocol that achieves the minimum possible maximum per-job pi-blocking is considered *optimal*. If the maximum per-job pi-blocking under a locking protocol is within a constant factor of the minimum, then it is considered *asymptotically optimal*.

In recent years, several suspension-based multiprocessor real-time locking protocols have been developed that provide asymptotically optimal upper bounds on pi-blocking under *suspension-oblivious* (s-oblivious) schedulability analysis [Brandenburg and Anderson, 2010a, 2011; Brandenburg, 2013a]. An s-oblivious schedulability analysis, such as Theorem 1.2, does not explicitly account for pi-blocking times. Instead, pi-blocking must be treated as computation, and task WCETs are inflated accordingly before performing the schedulability analysis. For mutual-exclusion (mutex) sharing, most (if not all) known asymptotically optimal locking protocols under s-oblivious analysis ensure that per-job pi-blocking is at most $2M - 1$ request lengths on an M-processor platform under any JLFP scheduler [Brandenburg and Anderson, 2010a, 2011].[6] The commonality of this bound is somewhat surprising as these protocols include ones that target different scheduling strategies (*e.g.,* partitioned, global, and clustered scheduling) and employ different mechanisms to cope with pi-blocking (*e.g., priority inheritance* vs. *priority donation* [Brandenburg and Anderson, 2010a, 2011]). In contrast, under s-oblivious analysis, the current best lower bound yields a per-job pi-blocking bound of at least $M - 1$ request lengths [Brandenburg and Anderson, 2010a]. This gap between the existing lower bound and upper bound raises an obvious question: *is a pi-blocking bound of* $2M - 1$ *request lengths fundamental under JLFP scheduling?*

Why does the 'two' matter in practice? When "thinking asymptotically," a factor of two may seem insignificant. However, use cases exist where doubling pi-blocking costs in schedulability analysis can have serious negative consequences. For example, a common approach for predictably sharing a hardware accelerator is to use a mutex locking protocol to ensure that each task has exclusive access when performing an accelerator operation. In the case of a GPU, the corresponding request length can be rather large, so doubling pi-blocking costs in analysis can easily make a system unschedulable.

[6] A refined statement is given in Chapter 2 by distinguishing between *request blocking* and *release blocking*.

1.2.3 Scheduling Gang Tasks

Despite the relevance of gang tasks in autonomous real-time systems, the scheduling of SRT gang tasks has received little attention. Specifically, only an SRT-schedulability test under G-EDF and corresponding response-time bounds are known [Dong et al., 2021]. Moreover, scheduling gang tasks is complicated by the internal parallelism of each task. In particular, systems with gang tasks encounter *parallelism-induced idleness*,[7] which leads to capacity loss regardless of the scheduling policy employed. This gives rise to the following question: *How can we optimally schedule SRT sporadic gang tasks?*

Although the above paragraph does not consider dependencies between gang tasks, many real-time systems feature not only gang tasks but also dataflow-related precedence constraints among them. Such systems can be modeled as gang tasks forming processing graphs. Moreover, these systems are often scheduled on multicore machines augmented with GPUs, requiring consideration of heterogeneous platforms consisting of multiple types of computing elements (CEs). Despite this relevance, the scheduling of gang tasks forming processing graphs on heterogeneous platforms has not been thoroughly studied, with the exception of a single work focused on NVIDIA-specific GPU scheduling of SRT tasks [Yang et al., 2018]. As a result, response-time analysis for such autonomous real-time systems often relies conservatively on converting the problem to the CPU-only scheduling of processing graphs formed by sequential tasks. This conversion models GPU execution either as *self-suspension* time (when no explicit GPU management is used), or as a combination of pi-blocking time and CPU execution time (when GPU access is managed through a locking protocol). Unfortunately, these approaches inherit the pessimism associated with the analysis of self-suspending tasks and locking protocols. This leads to the following question: *How can we derive response-time bounds for DAGs formed by gang tasks scheduled on multiple CEs, without requiring a conversion to a CPU-only scheduling problem?*

1.3 Thesis Statement

In this dissertation, we partially address the limitations of existing work described in Section 1.2 by supporting the following thesis statement.

[7] We give a detailed explanation in Chapter 6.

Exact and tight (within a constant factor) response-time bounds can be derived efficiently for periodic and graph-based tasks on identical multiprocessors under G-EDF, G-FIFO, and their variants when task periods satisfy certain properties. Multiprocessor locking protocols for mutex sharing exist that are optimal or nearly optimal under G-EDF, G-FIFO, and their variants. For HRT (resp., SRT) systems of gang tasks with (resp., without) precedence constraints, capacity loss can be significantly reduced by designing new schedulability conditions and response-time bounds.

1.4 Contributions

In this section, we elaborate on our contributions that support the above thesis. We first present our work on developing tight and exact response-time analysis techniques for sequential and DAG tasks. Next, we discuss our contributions on new optimality results for suspension-based multiprocessor locking protocols. Finally, we present our contributions on scheduling gang tasks.

1.4.1 Tight and Exact Response-Time Bounds for Sequential and DAG Tasks

To overcome the limitations of prior work on scheduling sequential and DAG tasks, as discussed in Section 1.2.1, we develop new response-time analysis techniques that provide tighter and exact bounds. In the following, we first describe our contributions related to sequential tasks, followed by those concerning DAG tasks.

1.4.1.1 Periodic Tasks

In Chapter 3, we give a closed-form response-time bound that is tight within a constant factor for a class of periodic tasks, called *pseudo-harmonic* tasks, under G-EDF, G-FIFO, and their variants. A set of tasks is called *pseudo-harmonic* if all task periods divide the largest period. Pseudo-harmonic tasks have attained significant interest in real-time systems research due to their applicability in several application domains such as avionics, robotics, control applications, *etc.* [Busquets-Mataix et al., 1996; Shih et al., 2003; Li et al., 2003; Anssi et al., 2013; Fu et al., 2010]. For pseudo-harmonic tasks, we show that the worst-case response time of a task τ_i is at most $T_i + \max_k\{T_k\}$ under G-FIFO and $2T_i + \max_k\{T_k\} - \min_k\{T_k\}$ under G-EDF where

tasks have implicit deadlines. We also demonstrate that these bounds are tight within a constant factor by providing an example task system where a task's exact response time closely approaches the derived bounds.

In Chapter 3, we also present a simulation-based technique for computing the exact response-time bound of any periodic task system. Developing such a technique involves addressing two fundamental questions: first, what should each job's execution time be during the simulation to observe exact response times? Second, when should the simulation be terminated? The answer to the first question is straightforward under G-EDF and its variants: the worst-case response time occurs when all jobs execute for their WCETs. In contrast, determining when to terminate the simulation is more involved. It requires identifying a time instant t_e such that, for each task τ_i, a job experiencing the worst-case response time completes by time t_e. We show how such a t_e can be determined by exploiting the *schedule repetition* property of G-EDF schedules. Compared to existing simulation-based techniques, our approach offers two advantages. First, while existing techniques apply only to systems that are HRT-schedulable under G-EDF, our technique is applicable to any SRT-feasible task system. Second, our method requires simulation for a pseudo-polynomial number of *hyperperiods* (the least common multiple of all task periods), whereas existing work on similar problems requires an exponential number. Consequently, the simulation runs in pseudo-polynomial time for systems with pseudo-harmonic tasks.

1.4.1.2 Periodic DAG Tasks

In Chapter 4, we present a *server*-based scheduling algorithm for DAG tasks. In this approach, each task receives a *budget* from a server and is allowed to execute only when sufficient budget is available. For such scheduling, we develop a simulation-based technique to derive exact response-time bounds for periodic DAG tasks by leveraging the repetitive property of server schedules. The derived response-time bounds remain valid even when a certain number of jobs from the same task are allowed to execute in parallel—a feature common in computer vision applications where multiple video frames may be processed concurrently. We also give *slack-reallocation* methods to reclaim unused server budgets, when possible, without violating the derived response-time bounds.

1.4.2 Optimality Results for Suspension-Based Locking Protocols

The next contribution in this dissertation concerns new optimality results for multiprocessor suspension-based locking protocols, resolving long-standing mysteries surrounding the $2M - 1$ pi-blocking upper bound mentioned in Section 1.2.2.

FIFO scheduling. In Chapter 5, we propose a suspension-based mutex locking protocol called the *optimal locking protocol under FIFO scheduling* (OLP-F). The OLP-F achieves the optimal s-oblivious pi-blocking bound under C-FIFO scheduling. Thus, the OLP-F is also optimal under G-FIFO and P-FIFO scheduling. We also consider an extension of mutex sharing called k*-exclusion* sharing, which allows k simultaneous lock holders. For k-exclusion sharing, we propose the *optimal locking protocol for k-exclusion under FIFO scheduling* (k-OLP-F) and show that it achieves the optimal s-oblivious pi-blocking bound under C-FIFO scheduling. Finally, we expand even further beyond mutex sharing by considering *reader-writer* (RW) sharing, where exclusive resource usage is only required for write accesses, and concurrent read accesses are permitted. For RW sharing, we propose the *read-optimal RW locking protocol under FIFO scheduling* (RW-OLP-F), which provides an optimal s-oblivious pi-blocking bound for read requests under C-FIFO scheduling. Additionally, under the RW-OLP-F, the pi-blocking bound for write requests is just under two request lengths of optimal.[8]

Non-FIFO global JLFP scheduling. In Chapter 5, we give a lower bound of $2M - 2$ request lengths on the maximum per-job pi-blocking under a broader subset of non-FIFO global JLFP schedulers that includes G-FP and G-EDF scheduling. This lower-bound result implies that the pi-blocking bound of $2M - 1$ request lengths, ensured by existing asymptotically optimal locking protocols, is just under one request length of optimal under a class of non-FIFO global JLFP schedulers. Moreover, we show that, when a locking protocol adheres to certain conditions, which most protocols would naturally adhere to, the maximum per-job pi-blocking is just one time unit smaller than $2M - 1$ request lengths under a class of non-FIFO global JLFP schedulers.

[8]All mentioned results are also valid under C-FIFO scheduling.

1.4.3 Scheduling Gang Tasks

The final contribution of this dissertation is the development of new scheduling techniques and response-time analysis for gang tasks. We consider the scheduling of both sporadic gang tasks and gang tasks with precedence constraints. Each of these is discussed in the following sections.

1.4.3.1 SRT Scheduling of Independent Gang Tasks

In Chapter 6, we develop a necessary and a sufficient condition for SRT-feasibility of independent gang tasks. Each condition involves associating an SRT task system with a corresponding HRT one that is "equivalent" in a feasibility sense. Using these feasibility conditions, we show that the SRT-feasibility problem for gang tasks is NP-hard. Leveraging the sufficient condition, we propose server-based scheduling policies along with corresponding schedulability tests for gang tasks. Finally, we analyze G-EDF scheduling of gang tasks. We show that G-EDF is not SRT-optimal for gang scheduling and present an improved schedulability test under G-EDF that outperforms existing approaches.

1.4.3.2 HRT Scheduling of Processing Graphs Formed by Gang Tasks

In Chapter 7, we introduce a new task model in which gang tasks form DAGs. We study the scheduling of such DAGs on a heterogeneous hardware platform composed of multiple CEs. This combination of workload and hardware architecture is well suited for modeling autonomous applications running on multicore machines augmented with GPUs. In the proposed model, GPUs are treated as computing resources rather than merely as shared resources accessed via locking protocols (as discussed in Section 1.4.2).

To schedule multiple DAGs, we use a *federated-scheduling* approach in which each DAG is allocated a set of processors on each CE. This approach can be realized on many hardware accelerators today through their compute-partitioning capabilities [Biondi and Buttazzo, 2017; Bakita and Anderson, 2023]. To allocate processors from different CEs to each DAG, we formulate an integer linear program (ILP). Within the allocated set of processors, we consider scheduling each DAG in either a *work-conserving* or *semi-work-conserving* manner. This is motivated by the fact that typical CPU scheduling approaches are work-conserving, whereas the default scheduling behavior of NVIDIA GPUs is semi-work-conserving under certain assumptions [Bakita and Anderson, 2024]. Under these scheduling approaches, we derive a response-time bound for each DAG on its allocated processors, assuming that each DAG has a constrained deadline.

1.5 Organization

In Chapter 2, we review relevant background and related prior work. Each subsequent chapter is self-contained with definitions and notation specific to it. In Chapters 3 and 4, we give tight and exact response-time analysis for sequential tasks and DAG tasks, respectively. In Chapter 5, we give optimal locking protocols under C-FIFO scheduling and present a lower bound on pi-blocking under a subset of non-FIFO global JLFP scheduling. In Chapter 6, we present our results on SRT scheduling of independent gang tasks. In Chapter 7, we discuss HRT scheduling of processing graphs formed by gang tasks. Finally, in Chapter 8, we give concluding remarks and discuss future work.

CHAPTER 2: BACKGROUND AND PRIOR WORK

In this chapter, we provide the necessary background for this dissertation, review relevant prior work, and position the contributions of this dissertation within the context of existing research. We begin with a review of sequential tasks (Section 2.1), followed by a discussion of DAG tasks (Section 2.2). Next, we review suspension-based mutex locks (Section 2.3) and gang tasks (Section 2.4). Finally, we give definitions, notation, and assumptions that remain throughout the dissertation (Section 2.5).

2.1 Sequential Tasks

In this section, we consider the sequential task model introduced in Section 1.1.1 and provide additional details relevant to this dissertation.

Periodic tasks. In addition to the task period and WCET, a periodic task has another parameter called its *offset* (also known as *phase*). A task τ_i's offset, denoted by Φ_i, is the release time of its first job $\tau_{i,1}$. Thus, the release time of job $\tau_{i,j}$ is

$$r(\tau_{i,j}) = r(\tau_{1,j}) + (j-1)T_i = \Phi_i + (j-1)T_i.$$

A periodic task system is *synchronous* if and only if $\Phi_i = \Phi_k$ holds for every pair of tasks τ_i and τ_k. Without loss of generality, we assume that $\Phi_i = 0$ for each task τ_i in a synchronous periodic task system. Otherwise, the system is said to be *asynchronous*.

In a *concrete* periodic task system, offsets of all tasks are known at design time. In contrast, in a *non-concrete* periodic task system, offsets are not known. Thus, different instantiations of a concrete periodic task system can differ only in job execution times. In contrast, two instantiations of a non-concrete periodic task system may also have different task offsets, leading to different job release times. Unless explicitly stated otherwise, we refer to concrete periodic task systems when discussing periodic task systems.

Pseudo-harmonic tasks. For any sporadic task system, the *hyperperiod* H is the least common multiple (LCM) of all task periods. Intuitively, the hyperperiod represents the minimum length interval after which

the job release patterns of all tasks repeat together, assuming each task releases jobs periodically. In a *pseudo-harmonic* task system, $H = \max\{T_i\}$ holds. This happens when all task periods divide the largest task period. The class of pseudo-harmonic task systems contains the well-known *harmonic* task systems. A task system is *harmonic* if and only if each task period can be divided by all smaller task periods in the system.

Example 2.1. Consider a task system where all task periods form the set $\{1, 2, 4, 5, 10\}$ms. The hyperperiod of this task system is 10ms. Since the hyperperiod is equal to the maximum period, this system is pseudo-harmonic. However, the period 5ms is not divisible by all smaller periods, so the system is not harmonic. In contrast, a task system with task periods in $\{1,2,4\}$ms is a harmonic task system. ◀

Pseudo-harmonic task systems are common in various application domains, such as automotive, avionics, robotics, and control systems [Kramer et al., 2015; Shih et al., 2003; Li et al., 2003; Anssi et al., 2013; Fu et al., 2010]. Moreover, harmonic task systems offer advantages in terms of schedulability. For example, although RM scheduling is not HRT-optimal in general, it is HRT-optimal for implicit-deadline sporadic tasks with harmonic periods on uniprocessors [Han and Tyan, 1997]. Additionally, exact HRT-schedulability test for constrained-deadline sporadic tasks with harmonic periods on uniprocessors under FP or EDF scheduling can be done in polynomial time [Bonifaci et al., 2013a], whereas the general problem is NP-hard [Eisenbrand and Rothvoß, 2008; Ekberg and Yi, 2015].

Intra-task parallelism and self-dependencies. The notion of *intra-task parallelism* and *self-dependencies* arises in SRT systems or in HRT systems where task relative deadlines exceed their periods. In such scenarios, multiple jobs of the same task may be present in the system simultaneously. The sporadic task model assumes that successive jobs $\tau_{i,j}$ and $\tau_{i,j+1}$ of a task τ_i cannot execute in parallel. This is typically the case when all jobs of a task execute on a single thread, *e.g.,* when a task executes a loop where each iteration corresponds to a job. Additionally, data dependencies between successive jobs may prevent their concurrent execution; for instance, job $\tau_{i,j+1}$ may require data produced by $\tau_{i,j}$. When jobs of a task cannot execute concurrently, the task is said to have *no parallelism* or *immediate self-dependency*. In contrast, two jobs of a task can execute concurrently when per-job threads are allowed or when there are no data dependencies between any two jobs of the task. In such a case, the task is said to have *unrestricted parallelism* or *no self-dependency*.

A generalization of the no parallelism and the unrestricted parallelism models, known as the *restricted parallelism* (rp) model, was introduced by Amert *et al.* in the context of graph-based computer vision

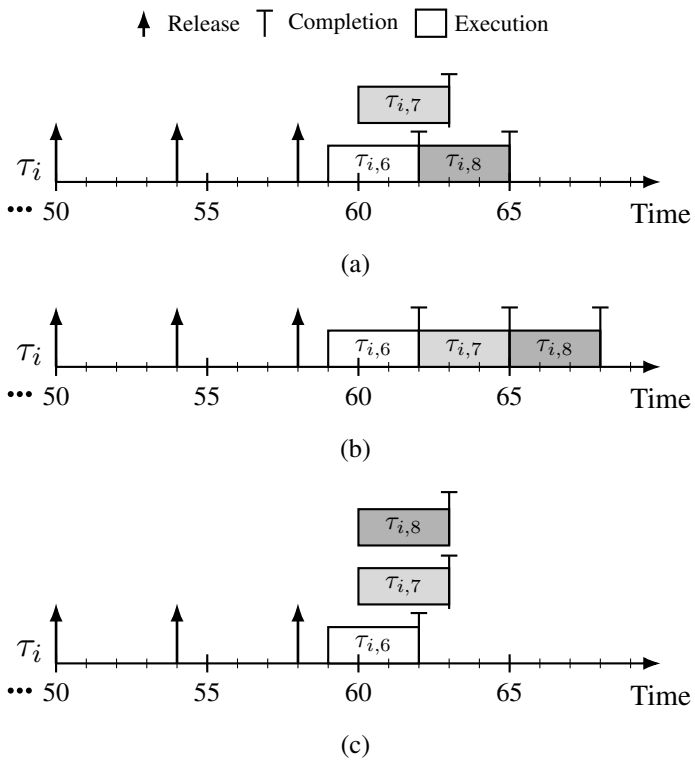


Figure 2.1: Illustration of (a) restricted parallelism with $P_i = 2$, (b) no parallelism, and (c) unrestricted parallelism.

applications [Amert et al., 2019]. Under the rp model, a task is associated with an additional parameter P_i, called its *parallelization level*. The parameter P_i defines self-dependencies among τ_i's jobs as follows: job $\tau_{i,j}$ with $j > P_i$ cannot start execution until $\tau_{i,j-P_i}$ completes. Note that P_i also represents the number of successive jobs of τ_i that can execute in parallel, *e.g.,* job $\tau_{i,j}$ can execute in parallel with jobs $\tau_{i,j-P_i+1}, \tau_{i,j-P_i+2}, \ldots, \tau_{i,j-1}$. Under the rp model, τ_i has no parallelism (immediate self-dependency) if $P_i = 1$. In contrast, τ_i has unrestricted parallelism (no self-dependency) if $P_i = \infty$.

Example 2.2. Figure 2.1 depicts the execution of a task τ_i under different parallelization levels. In Figure 2.1(a), $P_i = 2$ is assumed. Assume that τ_i is scheduled on three processors. Jobs $\tau_{i,6}$, $\tau_{i,7}$, and $\tau_{i,8}$ are released at times 50, 54, and 58, respectively. At time 59, $\tau_{i,6}$ is scheduled. Suppose $\tau_{i,7}$ and $\tau_{i,8}$ are among the highest-priority jobs at time 60. Since $P_i = 2$, $\tau_{i,7}$ is scheduled at time 60. However, $\tau_{i,8}$ is not scheduled until time 62 when $\tau_{i,6}$ completes execution. In Figure 2.1(b), no parallelism is assumed; thus, all three jobs are executed sequentially. In Figure 2.1(c), unrestricted parallelism is assumed. Thus, both $\tau_{i,7}$ and $\tau_{i,8}$ are scheduled at time 60 concurrently with $\tau_{i,6}$. ◀

In the remainder of this section, we first review known SRT-optimal schedulers for sporadic tasks on identical multiprocessors. Next, we review existing exact HRT-schedulability tests for sporadic and periodic tasks. Finally, we review existing SRT response-time analysis techniques.

2.1.1 SRT-Optimal Scheduling

In this section, we review schedulers that are SRT-optimal for scheduling sequential tasks on identical multiprocessors.

Window-constrained scheduling. Leontyev and Anderson introduced a class of schedulers called *window-constrained schedulers*, which contains many common schedulers including EDF, FIFO, *least laxity first* (LLF), *earliest deadline zero laxity* (EDZL), *etc.* [Leontyev and Anderson, 2010]. Window-constrained schedulers are defined using the concept of *prioritization functions*. A prioritization function is a function of a job and time. At any time, jobs with smaller values of their prioritization functions have higher priorities. A scheduler is *window-constrained* if and only if prioritization function of a job $\tau_{i,j}$ maps to a value within an interval $[r(\tau_{i,j}) - a_i, d(\tau_{i,j}) + b_j)$ for some task-level constants a_i and b_i. Since the values of a prioritization function can vary over time, window-constrained schedulers fall under the category of JLDP schedulers. Leontyev and Anderson showed that window-constrained schedulers are SRT-optimal [Leontyev and Anderson, 2010].

G-EDF-like scheduling. The G-EDF-like (GEL) schedulers form a subclass of window-constrained schedulers, where prioritization functions are constant functions. The constant values of the prioritization functions are called *priority points* (PPs). The priority point of a job is defined by adding a *relative PP* (RPP) to the job's release time. The RPP is a task-level property—similar to the relative deadline of a task—specified by system designers. A job's priority is determined by its PP, with earlier PPs indicating higher priority. The relative PP of a task τ_i is denoted by Y_i. We assume $Y_i \geq 0$ holds for each task τ_i. The PP of a job $\tau_{i,j}$, denoted by $y(\tau_{i,j})$, is defined as

$$y(\tau_{i,j}) = r(\tau_{i,j}) + Y_i. \tag{2.1}$$

Note that G-FIFO (resp., G-EDF) scheduler can be obtained by setting $Y_i = 0$ (resp., $Y_i = D_i$). GEL schedulers are JLFP and SRT-optimal (as they are a subclass of window-constrained schedulers).

Semi-partitioned scheduling. Semi-partitioned scheduling is a hybrid approach combining global and partitioned scheduling. Under semi-partitioned scheduling, only a subset of tasks is allowed to migrate, while the remaining tasks—called *fixed tasks*—are statically assigned to processors. The sets of fixed and migrating tasks are determined offline. During runtime, the scheduler schedules tasks according to the used scheduling algorithm, without violating constraints specific to fixed and migrating tasks. The *EDF-based optimal semi-partitioned scheduler* (EDF-os) [Anderson et al., 2014], the *EDF-based tunable scheduler for uniform platforms* (EDF-tu) [Yang and Anderson, 2015a], and the *EDF-based semi-partitioned scheduler with containers* (EDF-sc) [Hobbs et al., 2019] are SRT-optimal semi-partitioned schedulers that are derived from EDF scheduling. The *EDF with task splitting and k processor in a group* (EKG) [Andersson and Tovar, 2006] and the *notional processor scheduling-fractional capacity* (NPS-F)[Bletsas and Andersson, 2009] schedulers are also SRT-optimal semi-partitioned schedulers, which are also HRT-optimal for implicit-deadline sporadic tasks.

HRT-optimal schedulers. Since an HRT-optimal scheduler ensures that each task's worst-case response time is bounded by its relative deadline, any HRT-optimal scheduler is also an SRT-optimal scheduler. Many HRT-optimal schedulers are known for implicit-deadline periodic or sporadic tasks. Many such algorithms approximate a fluid scheduler, where each task executes at a constant rate matching its utilization. The *proportionate-fair* scheduler (PFair) [Baruah et al., 1996] is one such scheduler for synchronous periodic tasks, in which each task makes progress proportionate to its utilization. PFair is a non-work-conserving scheduler that divides the timeline into equal-length quanta and makes scheduling decisions at each quantum. Variants of PFair, such as the *early-release fair* (ERfair) [Anderson and Srinivasan, 2000] and the *pseudo-deadline* (PD) [Baruah et al., 1995] schedulers, have been devised to improve its efficiency. The PD^2 scheduler [Anderson and Srinivasan, 2004] further improves the efficiency of the PD scheduler and can optimally schedule implicit-deadline HRT sporadic tasks.

The *largest-local-remaining-execution-time* (LLREF) [Cho et al., 2006] scheduler approximates fluid scheduling without requiring time quanta by using additional scheduling events for periodic tasks. The *local-remaining-execution-TL-plane* (LRE-TL) [Funk and Nadadur, 2009; Funk, 2010] scheduler extends the LLREF scheduler to handle arbitrary-deadline periodic tasks and implicit-deadline sporadic tasks, while incurring less runtime overheads. The *boundary-fair* (BF) [Zhu et al., 2003] scheduler also avoids the need for time quanta and makes scheduling decisions only at period boundaries, *i.e.,* at job release times.

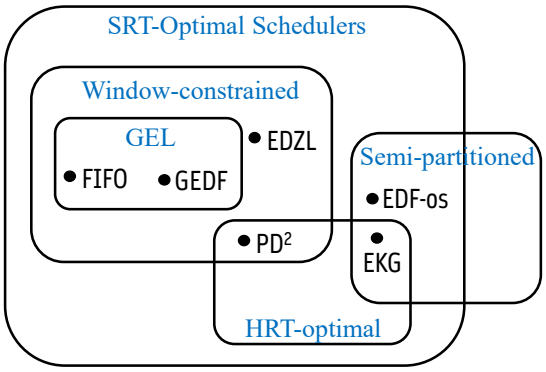


Figure 2.2: The class of known SRT-optimal schedulers for sporadic tasks.

The BF^2 scheduler extends the BF scheduler to support implicit-deadline sporadic tasks [Nelissen et al., 2014]. The EKG and NPS-F schedulers are HRT-optimal for implicit-deadline sporadic tasks [Andersson and Tovar, 2006; Andersson and Bletsas, 2008; Bletsas and Andersson, 2009]. These schedulers can be tuned to control the number of preemptions by trading off optimality. The *reduction to uniprocessor* (RUN) [Regnier et al., 2011] scheduler achieves HRT-optimality for implicit-deadline periodic tasks by reducing the multiprocessor scheduling problem to a series of uniprocessor scheduling problems. The *deadline-partitioning-fair* (DP-fair) [Levin et al., 2010] scheduling rules formalize deadline-partitioning techniques used by BF, EKG, and others. The *unfair EDF* (U-EDF) [Nelissen et al., 2011, 2014] scheduler applies scheduling principles similar to EDF without relying on fairness properties but still achieves HRT-optimality for implicit-deadline sporadic tasks.

However, there are known impossibility results regarding HRT-optimal schedulers. Hong and Leung showed that knowledge of both job release times and execution times is required to meet all deadlines for an arbitrary collection of jobs [Hong and Leung, 1988]. Later, Dertouzos and Mok demonstrated that this impossibility result still holds even when execution times are known [Dertouzos and Mok, 1989]. Subsequently, Fisher showed that no optimal scheduler exists for constrained- or arbitrary-deadline sporadic task systems if scheduling decisions must be made without knowledge of future job releases or execution times [Fisher, 2007].

Summary. Figure 2.2 shows the class of known SRT-optimal schedulers for sporadic tasks. The class contains all HRT-optimal schedulers for implicit-deadline sporadic tasks and all window-constrained schedulers. The class also contains a subset of semi-partitioned schedulers.

2.1.2 Exact HRT-Schedulability Test

In this section, we review prior work on exact-schedulability tests regarding G-EDF, G-FIFO, and G-FP scheduling. The problem of determining feasibility and schedulability of a sporadic task system exactly is intractable [Leung and Whitehead, 1982; Ekberg and Yi, 2015, 2017; Ekberg, 2020]. Thus, exact schedulability tests are not efficient unless they are applied to small sporadic task systems. Before reviewing the results regarding exact tests, we first review the concept of *sustainability*.

Sustainability. *Sustainability* is a property that indicates whether a system remains schedulable when it behaves better at runtime than its worst-case assumptions [Baruah and Burns, 2006]. Such favorable behavior may arise in several ways: jobs may execute for less than their WCETs, the true WCETs may be smaller than the specified values used in schedulability analysis, or the actual relative deadlines may be longer than assumed, *etc.* Among these, variations in execution times are of particular interest. This is because jobs often execute for less than their WCETs, and the specified WCETs are typically conservative upper bounds. C-sustainability (or simply, sustainability) ensures that a system deemed schedulable under assumed WCETs remains schedulable at runtime.

Sustainability can be attributed to both a scheduler and its associated schedulability tests. A scheduler is sustainable if it is free from *timing anomalies*, *i.e.,* no job experiences an increased response time when some jobs execute for less than their WCETs compared to when all jobs execute for their WCETs. Preemptive EDF is a sustainable scheduler, whereas non-preemptive EDF is not.

In contrast, a schedulability test or a response-time analysis is sustainable if its conclusions remain valid even when some jobs execute for less than their WCETs. If a test or analysis does not assume that all jobs execute for their WCETs, then it is sustainable by construction. If such an assumption is made, then the analysis remains valid at runtime only if the underlying scheduler is itself sustainable or if execution for full WCETs is enforced for all jobs at runtime. This enforcement can be achieved by refraining from scheduling any job on the processor where a job completed early until the full WCET duration has elapsed.

Simulation-based tests. For sustainable schedulers, the schedulability of a periodic task system can often be determined by simulating the scheduler for a finite duration, assuming each job executes for its WCET. To ensure the validity of such a conclusion, the simulation must explore all *system states* reachable under worst-case conditions. Schedulers that produce *repetitive* (also called *cyclic* or *periodic*) schedules in the

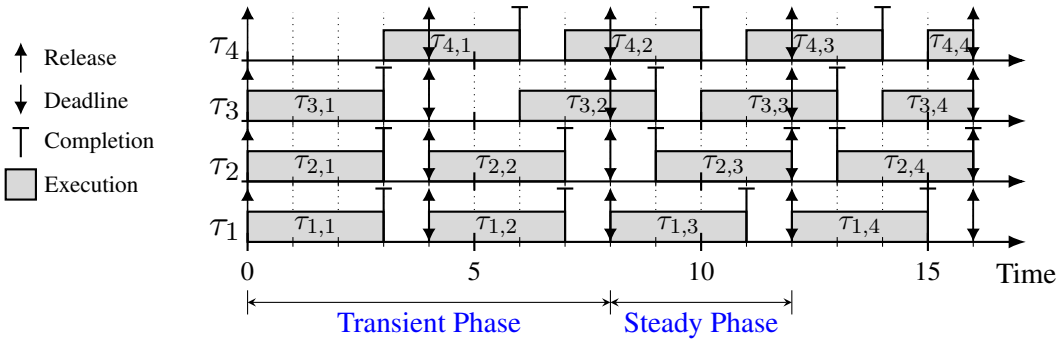


Figure 2.3: Illustration of schedule repetition.

worst case can guarantee this. Such schedules consist of two phases: a *transient* phase at the beginning, followed by a *steady* phase.

Example 2.3. Consider a periodic task system with four tasks scheduled on three processors using G-EDF. Each task has a period of 4.0, a WCET of 3.0, and an offset of 0.0. Figure 2.3 shows a G-EDF schedule $\mathcal{S}$ of the task system, assuming every job executes for its WCET. In $\mathcal{S}$, the schedule during the interval $[8, 12)$ repeats during $[12, 16)$. Thus, the transient and steady phases in $\mathcal{S}$ occur during $[0, 8)$ and $[8, 12)$, respectively. ◀

Since the job release patterns repeat after every hyperperiod, the steady phase of a schedule usually spans a number of hyperperiods. The *simulation interval* defines the time window within which schedule repetition begins, as formally defined below.

Definition 2.1 (Simulation Interval)**.** For a schedule $\mathcal{S}$ of a periodic task system, a simulation interval is a finite time interval $[0, a)$ such that $\mathcal{S}$ starts to repeat in a cycle by time a, *i.e.,* there exists an $\ell \leq a$ such that the schedule in $\mathcal{S}$ during $[a - \ell, a)$ is identical to the schedule in $\mathcal{S}$ during $[a + k\ell, a + (k + 1)\ell)$ for any integer $k \geq 0$. The length of the interval $[0, a)$ is called the *simulation length*. ◀

A simulation interval in a schedule $\mathcal{S}$ is *exact* if and only if there is no simulation interval of smaller simulation length in $\mathcal{S}$. Thus, the interval $[0, 12)$ is an exact simulation interval in Figure 2.3. The interval $[0, 14)$ is also a simulation interval in Figure 2.3, but it is not exact. For a sustainable scheduler, the worst-case response time (*i.e.,* the exact response-time bound) of each task can be determined by simulating the schedule for a simulation interval assuming every job executes for its WCET. Since scheduling decisions are typically made in polynomial time, the efficiency of a simulation-based test can be measured by the simulation length, which is typically an integer multiple of the system's hyperperiod plus the largest task offset. Note that a

simulation interval may not always exist. For example, a schedule of a system that is not SRT-schedulable under a scheduler may not have any simulation interval.

A simulation-based exact test for FP scheduling of periodic task systems on uniprocessors was introduced by Leung and Merrill [Leung and Merrill, 1980], and later tightened by Goossens and Devillers [Goossens and Devillers, 1997]. Goossens and Devillers also extended the result of [Leung and Merrill, 1980] to arbitrary-deadline periodic tasks under FP and EDF scheduling [Goossens and Devillers, 1999]. The exact simulation interval corresponding to the test was determined in [Grolleau and Choquet-Geniet, 2002]. These works assume no preemption or context-switch delays; to account for such overheads, task WCETs must be inflated before simulation. More recently, simulation intervals have been devised that explicitly account for preemption delays and context-switch costs on uniprocessors [Goossens and Masson, 2022, 2024]. However, these simulation intervals cannot be used for exact schedulability tests. In all of these works, the number of hyperperiods in the simulation intervals is pseudo-polynomial.

Simulation-based exact tests have also been studied for the JLFP scheduling of periodic tasks on multiprocessors. Cucu and Goossens gave an exact test for constrained-deadline periodic systems on uniform multiprocessors under G-FP schedulers [Cucu and Goossens, 2006]. This test requires simulation for a pseudo-polynomial number of hyperperiods. For G-FP scheduling of arbitrary-deadline periodic systems, Cucu-Grosjean and Goossens gave a simulation-based exact test assuming identical multiprocessors [Cucu-Grosjean and Goossens, 2007], which was later extended to unrelated multiprocessors [Cucu-Grosjean and Goossens, 2011]. The number of hyperperiods in the derived simulation interval is also pseudo-polynomial. Subsequently, a class of schedulers, called *deterministic* and *memoryless* schedulers, was shown to generate schedules with simulation intervals (*i.e.,* repetitive schedules), assuming all jobs of a task execute for the same duration [Grolleau et al., 2013]. Such schedulers make scheduling decisions deterministically based on solely the current state of the system. Baro *et al.* utilized the simulation interval from [Cucu-Grosjean and Goossens, 2011] to construct scheduling tables for constrained-deadline systems with simple precedence constraints [Baro et al., 2012]. Nélis *et al.* provided an exact schedulability test for constrained-deadline task systems on identical multiprocessors under any JLFP scheduler that requires simulation for a pseudo-polynomial number of hyperperiods [Nélis et al., 2013]. Goossens *et al.* later derived simulation intervals containing an exponential number of hyperperiods for arbitrary-deadline systems with complex features such as precedence constraints, non-preemptivity, *etc.*, under deterministic and memoryless schedulers [Goossens et al., 2016]. Table 2.1 summarizes these results on multiprocessor simulation intervals.

Table 2.1: Multiprocessor simulation intervals. D&M denotes deterministic and memoryless schedulers.

Work	Processor Model	Deadlines	Scheduler	Simulation length in hyperperiods
[Cucu and Goossens, 2006]	Uniform	Constrained	G-FP	Pseudo-polynomial
[Cucu-Grosjean and Goossens, 2007]	Identical	Arbitrary	G-FP	Pseudo-polynomial
[Cucu-Grosjean and Goossens, 2011]	Unrelated	Arbitrary	G-FP	Pseudo-polynomial
[Baro et al., 2012]	Identical	Constrained	Offline	Pseudo-polynomial
[Nélis et al., 2013]	Identical	Constrained	JLFP	Pseudo-polynomial
[Goossens et al., 2016]	Identical	Arbitrary	D&M	Exponential
This dissertation	Identical	Arbitrary	GEL	Pseudo-polynomial

Reachability-based methods. Exact schedulability tests for sporadic task systems are typically based on *reachability analysis* over a graph or automaton that captures all possible system states. Baker and Cirinei presented an exact schedulability test for arbitrary-deadline sporadic tasks under G-EDF and G-FP scheduling [Baker and Cirinei, 2007]. The test relies on a brute-force exploration of a finite automaton that encodes all possible system behaviors. The size of this automaton can be exponential, resulting in an exponential-time exact test. However, Geeraerts *et al.* showed that the formulation in [Baker and Cirinei, 2007] is PSPACE-complete and applied antichain techniques to improve the test's scalability [Geeraerts et al., 2013]. Guan *et al.* proposed a *timed-automata*-based exact analysis for periodic tasks under G-FP scheduling [Guan et al., 2007], while Sun and Lipari developed an exact test for G-FP schedulers using reachability analysis on linear hybrid automata [Sun and Lipari, 2016]. All these exact schedulability tests suffer from limited scalability, *e.g.,* the test in [Sun and Lipari, 2016] can handle systems with at most seven tasks and four processors on their experiment platform. *Schedule abstraction graphs* have recently been introduced, which can be used for exact analysis of non-preemptive uniprocessor systems with release jitters [Nasri and Brandenburg, 2017]. Recently, this technique has been extended for global preemptive and non-preemptive JLFP scheduling for jobs with release jitters [Nasri et al., 2018; Gohari et al., 2024]. However, the extensions to multiprocessor scheduling are only sufficient tests.

Contribution of this dissertation. In this dissertation, we give a simulation-based exact schedulability test for periodic tasks on identical multiprocessors under GEL schedulers. The simulation length is bounded by a pseudo-polynomial number of hyperperiods. The most comparable prior works are [Nélis et al., 2013]

and [Goossens et al., 2016] (see Table 2.1). Although Nelis *et al.* gave a simulation interval containing a pseudo-polynomial number of hyperperiods, it only applies to constrained-deadline systems. In contrast, although Goossens *et al.* considered a wider class of schedulers, the simulation length contains an exponential number of hyperperiods.

2.1.3 SRT Response-Time Analysis

In this section, we review prior work on response-time analysis of SRT systems. We first discuss work assuming identical multiprocessors, followed by work on heterogeneous multiprocessors.

Identical multiprocessors. Response-time bounds that apply to any SRT-feasible systems have been mostly studied under G-EDF. Devi and Anderson gave the first such bound for implicit-deadline sporadic tasks under preemptive G-EDF on identical multiprocessors [Devi and Anderson, 2005]. Later, the bound was extended for sporadic tasks with an arbitrary number of non-preemptive sections [Devi and Anderson, 2008]. For fully utilized task systems, the response-time bound by Devi and Anderson increases with respect to the number of processors. Erickson *et al.* improved and extended this bound to arbitrary-deadline sporadic systems by introducing an analysis technique called the *compliant vectors analysis* (CVA) [Erickson et al., 2010]. The current best-known G-EDF response-time bound on identical multiprocessors, called the *harmonic bound*, was given by Valente [Valente, 2016]. Although the *harmonic bound* also increases with respect to the number of processors for fully utilized systems, in the worst case, it increases logarithmically with respect to the number of processors compared to the linear increase in [Devi and Anderson, 2005]. The time complexity to compute the harmonic bound is exponential. Leoncini *et al.* improved the efficiency of the algorithm for computing the harmonic bound using the *branch-and-bound* technique [Leoncini et al., 2019].

The SRT-optimality and corresponding response-time bounds have also been studied under non-G-EDF global schedulers. Leontyev and Anderson proved a response-time bound similar to the one in [Devi and Anderson, 2005] for sporadic tasks under G-FIFO [Leontyev and Anderson, 2007]. Later, the response-time bounds in [Devi and Anderson, 2005] and [Leontyev and Anderson, 2007] were generalized for sporadic tasks under window-constrained schedulers [Leontyev and Anderson, 2010]. Erickson *et al.* achieved a tighter bound under GEL schedulers using CVA [Erickson and Anderson, 2012; Erickson et al., 2014]. For fully utilized systems, this bound also increases with the number of processors. Erickson *et al.* also introduced the *global-fair-lateness* (G-FL) scheduler and showed that G-FL has the tightest bound among all

GEL schedulers under CVA. Recently, a tight response-time bound was derived for any work-conserving scheduling of periodic task systems where all tasks have the same period and the same WCET [Buzzega et al., 2023; Buzzega and Montangero, 2024]. This bound consists only of the task WCET and period. Although the G-FP scheduler is not SRT-optimal when jobs of the same task cannot execute concurrently, Voronov *et al.* showed that the G-FP scheduler is SRT-optimal when unrestricted parallelism is allowed among jobs of the same task [Voronov et al., 2021a]. Unrestricted parallelism is also advantageous for G-EDF scheduling. When any two jobs of a task can execute in parallel, response-time bounds under G-EDF do not increase with the processor count for fully utilized systems [Erickson and Anderson, 2011].

Heterogeneous multiprocessors. Yang and Anderson showed that non-preemptive work-conserving schedules, including non-preemptive G-EDF, are not SRT-optimal on uniform multiprocessors [Yang and Anderson, 2015b]. Later, they proved the SRT-optimality of preemptive G-EDF for scheduling sporadic tasks on uniform multiprocessors [Yang and Anderson, 2017]. This required a refinement of G-EDF, called the Ufm-EDF, in which earlier-deadline jobs are scheduled on higher-speed processors. The response-time bound given by Yang and Anderson grows exponentially with the processor count. Tang *et al.* improved this result by providing a bound that is polynomial in both task parameters and processor count [Tang et al., 2019]. This bound is also applicable to G-EDF scheduling of sporadic tasks on identical multiprocessors with *affinity masks*, which are per-task bit vectors that specify the processors on which a task may execute [Tang et al., 2019]. The required refinement of preemptive G-EDF to ensure SRT-optimality on identical multiprocessors with affinity masks causes heavy migrations. Tang and Anderson showed that an alternative refinement of preemptive G-EDF with fewer migrations, as well as non-preemptive G-EDF, are not SRT-optimal [Tang and Anderson, 2020]. However, they showed that window-constrained scheduling on identical multiprocessors with affinity masks is SRT-optimal [Tang and Anderson, 2020]. Tang *et al.* gave another refinement of G-EDF for unrelated multiprocessors, called Unr-EDF, and gave a response-time bound that applies with a restriction on per-task utilizations and total utilization [Tang et al., 2021]. Later, the result was extended to window-constrained schedulers [Tang, 2024].

Contribution of this dissertation. In this dissertation, we give a tight (within a constant factor) response-time bound for pseudo-harmonic periodic task systems under preemptive GEL schedulers on identical multiprocessors. This bound can be computed in polynomial time. For such systems, we also give an exact

Table 2.2: Response-time bounds of SRT systems on identical multiprocessors.

Work	Task Model	Restriction	Scheduler	Tight	Exact
[Devi and Anderson, 2005]	Sporadic	None	G-EDF	No	No
[Leontyev and Anderson, 2007]	Sporadic	None	G-FIFO	No	No
[Leontyev and Anderson, 2010]	Sporadic	None	Window-constrained	No	No
[Erickson et al., 2010]	Sporadic	None	G-EDF	No	No
[Erickson et al., 2014]	Sporadic	None	GEL	No	No
[Valente, 2016]	Sporadic	None	G-EDF	No	No
[Buzzega and Montangero, 2024]	Periodic	Same period and WCET	Work-conserving	Yes	Yes
This dissertation	Periodic	Pseudo-harmonic	GEL	Yes	Yes

response-time bound that can be computed in pseudo-polynomial time. Table 2.2 shows a comparison of our work with relevant prior work.

2.2 DAG Tasks

A DAG-based task system consists of N *DAG task* $G^1, G^2, \ldots, G^N$. To simplify the notation, we omit the DAG index (in the superscript) when discussing a single DAG. Each DAG task G has n nodes that represent sequential tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. The WCET of node τ_i is C_i. A directed edge from τ_i to τ_k represents a precedence constraint between the *predecessor* task τ_i and the *successor* task τ_k. The set of predecessors of τ_i is denoted by $pred(\tau_i)$. Each DAG task G has a unique *source* task τ_1 with no incoming edges and a unique *sink* task τ_n with no outgoing edges. A DAG with multiple sources/sinks can be supported by adding a "virtual" source and sink, each with a WCET of zero, which connects with multiple sources and sinks, respectively.

Each DAG task G has a *period* T. A sporadic (resp., periodic) DAG G releases *DAG jobs* so that successive DAG jobs have a minimum (resp., exact) separation time of T time units. The j^{th} DAG job of G is denoted by G_j. The release time and completion time of G_j are denoted by r_j and f_j, respectively. The utilization of a task τ_i and DAG G are $u_i = C_i/T$ and $U = \sum_{i=1}^{n} u_i$, respectively. The *total utilization* of all

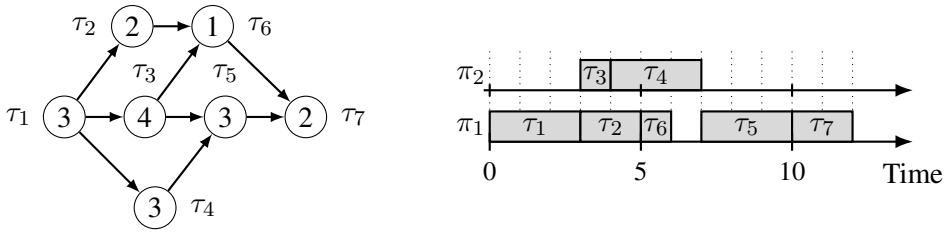


Figure 2.4: (a) A DAG G (numbers inside circles denote WCETs). (b) A schedule of G on two processors where τ_3 executes for less than its WCET.

DAGs, denoted by U_{tot}, is the sum of all per-DAG utilizations. The DAG job G_j consists of a *job* $\tau_{i,j}$ for each task τ_i in G. The *release time* and *finish time* of $\tau_{i,j}$ are denoted by $r(\tau_{i,j})$ and $f(\tau_{i,j})$, respectively. The source task τ_i releases its j^{th} job when G_j is released. The j^{th} job of each non-source task is released once the j^{th} jobs of all of its predecessor tasks finish. Thus, the release time of a job $\tau_{i,j}$ is

$$r(\tau_{i,j}) = \begin{cases} r_j & \text{if } \tau_i \text{ is a source task} \\ \max_{\tau_k \in pred(\tau_i)}\{f(\tau_{k,j})\} & \text{otherwise.} \end{cases} \tag{2.2}$$

DAG job G_j finishes execution when the j^{th} job $\tau_{n,j}$ of the sink node finishes, *i.e.,* $f_j = f(\tau_{n,j})$. The *response time* of $\tau_{i,j}$ is $f(\tau_{i,j}) - r(\tau_{1,j})$. Task τ_i's response time is $\sup_j\{f(\tau_{i,j}) - r(\tau_{1,j})\}$. G_j's response time equals $\tau_{n,j}$'s response time. G's response time equals τ_n's response time.

Example 2.4. Figure 2.4(a) depicts a DAG task G. Tasks τ_1 and τ_7 are the source and sink tasks of G, respectively. The predecessor tasks of τ_6 are τ_2 and τ_3.

Figure 2.4(b) depicts a schedule of a DAG job of G on two processors π_1 and π_2. In this schedule, the DAG job is released at time 0. Thus, by (2.2), the source node's job is also released at time 0. The jobs of τ_2, τ_3, and τ_4 are released at time 3 when the job of τ_1 completes. The job of τ_6 is released at time 5, as the jobs of τ_2 and τ_3 complete execution at times 5 and 4, respectively. ◀

In the rest of this section, we first discuss the complexities involved in scheduling and analyzing DAG tasks. Next, we review common DAG scheduling approaches. Finally, we discuss HRT- and SRT-feasibility, existing work on HRT-schedulability analysis, and SRT response-time analysis for DAG tasks.

2.2.1 Complexities in DAG Scheduling

When multiple DAG tasks are scheduled on a shared multiprocessor platform, tasks from different DAGs can *interfere* with each other, delaying each other's execution. A job $\tau_{i,j}^{v}$ is said to *interfere* with another job $\tau_{k,\ell}^{w}$ if the execution of $\tau_{i,j}^{v}$ can delay the execution of $\tau_{k,\ell}^{w}$. We use these jobs, $\tau_{i,j}^{v}$ and $\tau_{k,\ell}^{w}$, to characterize different types of interference in the following.

Intra-DAG interference. Since a DAG can consist of multiple sequential tasks, a job of a DAG can interfere with other jobs of the same DAG. This is called *intra-DAG interference*. Intra-DAG interference refers to the case where $v = w$ holds. Intra-DAG interference can be categorized into two types.

- **Intra-instance interference.** *Intra-instance interference* refers to the interference between jobs that belong to the same DAG job. This corresponds to the case where $v = w$ and $j = \ell$.
- **Inter-instance interference.** *Inter-instance interference* refers to the interference between jobs from different DAG jobs of the same DAG. This corresponds to the case where $v = w$ and $j \neq \ell$. Inter-instance interference occurs in systems with arbitrary-deadline HRT DAG tasks or SRT DAG tasks.

Inter-DAG interference. When a job of one DAG interferes with a job of another DAG, it is called *inter-DAG interference*. Thus, inter-DAG interference refers to the case where $v \neq w$.

Self-dependencies. A job's execution can also be delayed if it requires data from a prior job of the same task, but the prior job has not completed yet. Such a scenario can happen under the rp model when the task does not have unrestricted parallelism. One might think that self-dependencies are special cases of inter-instance interference described above, *i.e.,* the case with $v = w$, $i = k$, and $j \neq \ell$. However, we characterize them differently due to schedule-related scenarios that arise due to self-dependencies but not due to such interference. For example, the number of busy processors can differ when interference- or dependency-related delay of execution occurs. When a job's execution is delayed due to interfering workload, all processors become busy during that delay. In contrast, if a job's execution is delayed due to self-dependencies, then some processors may remain idle during the delay. This can be seen in Figure 2.1(a), where $\tau_{i,8}$ cannot execute due to self-dependencies during $[60, 62)$, despite one processor may be idle.

2.2.2 DAG Scheduling Approaches

We now discuss common DAG scheduling approaches.

Table 2.3: Interference and dependencies under global scheduling of multiple DAGs.

Model	HRT/SRT and Deadline	Intra-DAG		Inter-DAG	Self-Dependencies
		Intra-Instance	Inter-Instance		
Unrestricted parallelism	HRT, Constrained	Yes	No	Yes	No
	HRT, $D_i > T_i$	Yes	Yes	Yes	No
	SRT	Yes	Yes	Yes	No
Restricted parallelism	HRT, Constrained	Yes	No	Yes	No
	HRT, $D_i > T_i$	Yes	Yes	Yes	Yes
	SRT	Yes	Yes	Yes	Yes

Decomposition-based scheduling. Under decomposition-based scheduling, each DAG is decomposed into a set of sequential tasks, which are scheduled independently. The decomposition is typically performed by assigning each node of the DAG an *offset*, representing the release time of a node's job relative to the release time of the corresponding DAG job. Such offset assignments ensure that all jobs of predecessor nodes of a node τ_i complete before a job of τ_i is released. Additionally, each node is assigned a relative deadline. Decomposition-based scheduling was studied in [Liu and Anderson, 2010; Saifullah et al., 2011; Qamhieh et al., 2013, 2014; Saifullah et al., 2014; Jiang et al., 2016; Yang et al., 2016; Pathan et al., 2018; Amert et al., 2019; Jiang et al., 2020; Guan et al., 2021, 2022].

Non-decomposition-based global scheduling. Under this approach, tasks of all DAGs are scheduled under a global scheduler without any decomposition. A hierarchical prioritization approach is often used. At the top level, jobs of different DAGs are prioritized according to inter-DAG prioritization rules. At the second level, jobs of the same DAG are prioritized according to intra-DAG prioritization rules. Commonly, FP or EDF is used for inter-DAG prioritization. When FP is used, jobs of a higher-priority DAG always have higher priorities than jobs of a lower-priority DAG. Under EDF, each DAG job's absolute deadline is used to define the priorities of the jobs of the DAG. Common approaches for intra-DAG prioritization include *list scheduling*, *prioritized list scheduling*, *etc.* Work on non-decomposition-based global scheduling includes [Graham, 1969; Baruah et al., 2012; Li et al., 2013; Bonifaci et al., 2013b; Baruah, 2014; Parri et al., 2015; He et al., 2019; Wang et al., 2019; Fonseca et al., 2019; Sun et al., 2020; He et al., 2021, 2022; Ueter et al., 2023]. Table 2.3 shows different interference and dependencies present under decomposition-based and non-decomposition-based global scheduling of DAG tasks.

Partitioned scheduling. Similar to partitioned scheduling of sequential tasks, each task of a DAG task is statically assigned to a single processor under partitioned scheduling. Schedulability analysis for partitioned scheduling of DAG tasks is typically performed by modeling the execution of a DAG task on a given processor as segments of execution and self-suspension. Partitioned scheduling of DAG tasks was studied in [Fonseca et al., 2016; Baruah, 2020]. A variant of partitioned scheduling of DAGs was considered in [Shi et al., 2024].

Federated scheduling. Under *federated scheduling*, each DAG is classified into either *heavy* or *light*. The utilization[1] of a heavy DAG (resp., light DAG) is larger than (resp., at most) 1.0. Each heavy DAG is scheduled on a set of dedicated processors. In contrast, all light DAGs share a common set of processors and are each scheduled as sequential tasks. On their allocated processors, heavy DAGs may be scheduled using decomposition-based, non-decomposition-based global, or partitioned scheduling approaches. Since heavy DAGs are isolated from one another, federated scheduling avoids inter-DAG interference, which enables more accurate response-time bounds. However, allocating processors to heavy DAGs may cause capacity loss: a DAG with utilization $a + \epsilon$, for integer a and $\epsilon \to 0$, requires at least $a + 1$ processors. Existing work on federated scheduling and its variants includes [Li et al., 2014; Baruah, 2015a,b; Jiang et al., 2017, 2021; Ueter et al., 2018; Guan et al., 2023].

2.2.3 Feasibility Results

Determining whether a sporadic DAG task is HRT-feasible on an identical multiprocessor platform is NP-hard in the strong sense, regardless of whether preemption is allowed or not [Ullman, 1975]. When multiple DAG tasks are considered, intractability results pertaining to the scheduling of both a single sporadic DAG task (restricting the number of DAG to one) and a set of sporadic sequential tasks (restricting per-DAG node count to one) on multiprocessors are applicable. Therefore, intractability and impossibility results from [Eisenbrand and Rothvoß, 2010; Ekberg and Yi, 2015; Fisher, 2007] are applicable to the scheduling of multiple DAGs. Recently, it has been shown that determining the HRT-feasibility of a set of constrained-deadline sporadic DAG tasks is PSPACE-hard [Bonifaci and Marchetti-Spaccamela, 2025]. This result rules out the possibility of formulating an ILP to solve this problem efficiently.

However, the SRT-feasibility of a set of DAG tasks on M identical multiprocessors can be determined in polynomial time even when self-dependencies are specified under then rp model, as shown below.

[1]For arbitrary-deadline DAG tasks, a term called *density* is used to distinguish between heavy and light DAGs.

Theorem 2.1 ([Amert et al., 2019])**.** *A set of N sporadic DAG tasks is SRT-feasible on M identical multiprocessors if and only if $\forall\, i, v : u_i^v \leq P_i^v$ and $U_{tot} \leq M$ hold.*

For the special case of unrestricted parallelism ($P_i^v = \infty$), the condition is only $U_{tot} \leq M$ (there is no explicit restriction on u_i^v) [Yang et al., 2016]. In contrast, for the no-parallelism case, the condition is $u_i^v \leq 1$ and $U_{tot} \leq M$ [Liu and Anderson, 2010].

2.2.4 HRT-Schedulability Analysis

The HRT-schedulability of sporadic DAG tasks has been studied under various schedulers. Most of these analyses provide sufficient schedulability tests or non-exact response-time bounds. Several common analysis techniques are used to derive such tests. One technique derives a response-time upper bound for a DAG task by bounding the interfering workload that affects the jobs along any *path* (an ordered sequence of nodes where successive nodes are connected by edges) in the DAG during a *problem window*. Another technique evaluates a non-optimal scheduler by comparing it to a hypothetical optimal scheduler using the concepts of *capacity augmentation* or *resource augmentation* bounds. Finally, under decomposition-based scheduling, individual response-time bounds for each task are typically derived and then aggregated to obtain a response-time bound for the entire DAG task.

Constrained-deadline systems. Constrained-deadline HRT systems do not have any inter-instance interference or self-dependencies. Thus, schedulability tests and response-time analyses of constrained-deadline DAG tasks typically require bounding intra-instance interference and inter-DAG interference. Under federated scheduling of constrained-deadline DAG tasks, the need to consider inter-DAG interference is also obviated. Eliminating these complex features often enables more accurate response-time bound computation. Graham gave a path-based analysis for a DAG task by upper bounding the workload that can interfere with jobs on a path under any work-conserving scheduler [Graham, 1969]. Recently proposed multi-path bounds improve Graham's bound by considering multiple paths of the DAGs [He et al., 2022; Ueter et al., 2023]. Some work also considered assigning node priorities to improve Graham's bound [He et al., 2019; Zhao et al., 2020; He et al., 2021; Chang et al., 2022]. Other work on constrained-deadline HRT DAG tasks provided schedulability tests by deriving *speed-up factors* or *resource-augmentation* bounds [Bonifaci et al., 2013b; Li et al., 2013; Baruah, 2014]. Decomposition-based analysis for constrained-deadline DAG tasks has also been studied under G-EDF and G-FP schedulers [Qamhieh et al., 2013; Saifullah et al., 2014; Jiang et al., 2016].

Arbitrary-deadline systems. For arbitrary-deadline HRT systems, most work assumes no self-dependencies (*i.e.,* unrestricted parallelism) [Yang et al., 2016; Ueter et al., 2018; Wang et al., 2019; Guan et al., 2023]. Schedulability tests for multiple DAG tasks under G-EDF and G-FP schedulers are given in [Baruah et al., 2012; Bonifaci et al., 2013b]. Other work determined response-time bounds of multiple DAG tasks by considering execution along a path of each DAG [Parri et al., 2015; Wang et al., 2019; Ueter et al., 2023]. Fonseca *et al.* developed such a path-based approach to determine response-time bounds for multiple arbitrary-deadline DAG tasks [Fonseca et al., 2019]. However, their approach required that a DAG job not commence execution before all previous DAG jobs finish, leading to capacity loss.

2.2.5 SRT Response-Time Analysis

Prior work on SRT scheduling and corresponding analysis has primarily focused on SRT-optimal schedulers. The only known SRT-optimal scheduler is a decomposition-based approach called the offset-based scheduler. Under this scheduler, a DAG is converted into an "equivalent" set of independent sporadic tasks by assigning appropriate task offsets that preserve precedence constraints. Unlike the offsets in periodic sequential tasks, the offset of τ_i^v, denoted by O_i^v, in offset-based scheduling represents the relative release time of the job $\tau_{i,j}^v$ with respect to the release time of the corresponding source task's job $\tau_{1,j}^v$. Accordingly, under offset-based scheduling, (2.2) is modified as follows.

$$r(\tau_{i,j}^v) = r_j^v + O_i^v. \tag{2.3}$$

To ensure that an offset assignment can implicitly satisfy precedence constraints, a response-time bound R_i^v of each task τ_i^v is computed, assuming that all tasks are independent, sporadic, have implicit deadlines, and scheduled under G-EDF.[2] These can be computed by any response-time analysis for sequential tasks under G-EDF, as described in Section 2.1.3. After computing such values, task offsets are determined as follows.

$$O_i^v = \begin{cases} 0 & \text{if } \tau_i \text{ is a source task} \\ \max_{\tau_k^v \in pred(\tau_i^v)}(O_k^v + R_k^v) & \text{otherwise.} \end{cases} \tag{2.4}$$

[2]Other SRT-optimal schedulers for independent sporadic tasks can also be used.

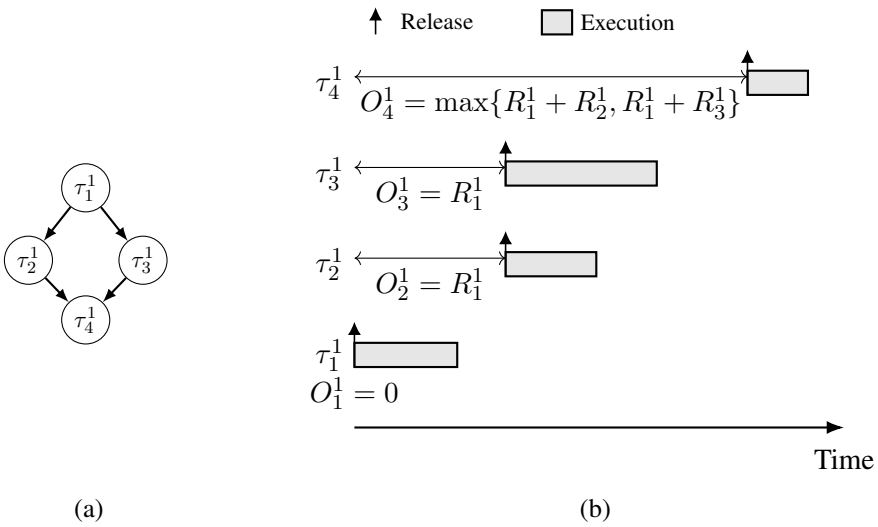


Figure 2.5: (a) A DAG task and (b) an offset-based schedule of the DAG.

Liu and Anderson first showed that job releases satisfying (2.3) and (2.4) ensure that all precedence constraints are respected when tasks have no parallelism ($P_i^v = 1$)[Liu and Anderson, 2010]. Later, the same offset assignment approach was shown to be valid for both unrestricted parallelism and arbitrary restricted parallelism [Yang et al., 2016; Amert et al., 2019]. Note that the response-time bounds used in (2.4) must be computed assuming the appropriate parallelization levels. Finally, the response time of a DAG task can be bounded by summing the sink node's offset and its response-time bound, *i.e.,* $O_{n^v}^v + R_{n^v}^v$.

Example 2.5. Figure 2.5 shows a DAG task and its offset-based schedule. Node τ_1^1's offset is 0 according to (2.4). Since τ_2^1 and τ_3^1 have a predecessor node τ_1^1, the offsets of both τ_2^1 and τ_3^1 are R_1^1. Since the predecessors of τ_4^1 are τ_2^1 and τ_3^1, the offset of τ_4^1 is $\max\{O_2^1 + R_2^1, O_3^1 + R_3^1\} = \max\{R_1^1 + R_2^1, R_1^1 + R_3^1\}$.◀

The above scheduling and its analysis ensure bounded response times for all SRT-feasible DAG task systems. This is because the analysis mostly relies on SRT response-time analysis of sequential tasks under G-EDF, which does not cause any capacity loss. However, the offset-based scheduling can delay a job's release due to the introduction of offsets. Such delays can be avoided with the concept of *early releasing*, which releases jobs by (2.2) instead of (2.3). Since most response-time bounds of independent sporadic tasks under G-EDF remain valid with early releasing, the computed response-time bound of a DAG under offset-based scheduling still applies when early releasing is allowed.

Unfortunately, reducing the response-time bound of a DAG computed under offset-based scheduling relies on improving response-time analysis of the corresponding independent sporadic task model. As

described in Section 2.1.3, the known SRT response-time analyses of sporadic tasks are pessimistic, and a tight bound is still unknown. With offsets defined using such pessimistic bounds, any improvement, even if an exact analysis can be performed, would be marginal. Other than offset-based scheduling, response-time analysis of DAGs with unrestricted parallelism was studied under some non-SRT-optimal schedulers [Liu and Anderson, 2011; Jiang et al., 2018].

Contribution of this dissertation. In this dissertation, we propose a *server-based* scheduling algorithm for periodic DAG tasks under the rp model. Server-based scheduling is a decomposition-based approach that uses the concept of *servers* or *reservations*, which are equivalent to independent periodic tasks, to provide execution budgets to jobs. A job can execute at any time instant if it has a non-zero budget and its server has sufficiently high priority. The use of servers, instead of offsets, enables the computation of exact response-time bounds for DAG tasks using simulation-based techniques. Note that response-time analysis for SRT DAG tasks under the rp model must account for all complexities listed in Table 2.3.

2.2.6 Other DAG Models

We now briefly describe some other DAG task models that generalize the one introduced in Section 2.2.

Conditional DAG tasks. In the conditional DAG task model, a DAG task can contain *conditional* nodes. These nodes represent the execution of conditional (*e.g.,* if-then-else) constructs in parallel real-time code. Thus, not all nodes may have a job in an instance of a conditional DAG task. Due to the presence of conditional nodes, determining the worst-case execution requirement of a DAG job requires considering different execution scenarios along various branches. Work on conditional DAG tasks includes [Fonseca et al., 2015; Baruah et al., 2015; Melani et al., 2015; Parri et al., 2015; Baruah, 2021; He et al., 2023a].

Typed DAG tasks. Typed DAG task systems are used to model the execution of DAG tasks on a heterogeneous compute platform consisting of multiple compute elements (CEs). In a *typed* DAG task, each node has a type associated with it. The type refers to a CE on which the node's jobs can execute. Jaffe first considered scheduling DAGs on heterogeneous platforms [Jaffe, 1980]. Later work improved this bound using less pessimistic interfering workload estimations [Han et al., 2019; He et al., 2023b]. Chang *et al.* considered the scheduling of typed DAGs by a JLDP scheduling algorithm [Chang et al., 2020]. Lin *et al.* gave a type-aware federated scheduling algorithm on two-CE platforms that allows the sharing of processors of a CE among DAGs that are light with respect to a CE [Lin et al., 2023]. Other work considered scheduling SRT DAGs

on heterogeneous platforms [Yang et al., 2016] and graph-restructuring techniques [Serrano and Quiñones, 2018].

Multi-rate DAG tasks. In a multi-rate DAG, different nodes of the same DAG may have different periods. Typically, each DAG node releases its jobs periodically, and data dependencies are resolved using the outputs of the most recently completed predecessor jobs. Much of the existing work on multi-rate DAG tasks relies on converting all DAGs into a set of single-rate DAG tasks. Recent work on multi-rate DAG tasks includes [Forget et al., 2010; Verucchi et al., 2020; Sun et al., 2023; Li et al., 2024].

2.3 Suspension-Based Mutex Locks

In this section, we review concepts and prior work related to suspension-based real-time locking protocols. For simplicity, we consider a sequential task system consisting of N implicit-deadline sporadic tasks $\{\tau_1, \tau_2, \ldots, \tau_N\}$. These tasks are scheduled on M identical processors under a global JLFP scheduler.

Resource model. We assume that the system has a set $\{\ell_1, \ldots, \ell_{n_r}\}$ of shared resources. For now, we limit attention to mutex sharing, although other notions of sharing will be considered later in Chapter 5. Under mutex sharing, a resource ℓ_q can be held by at most one job at any time. When a job $\tau_{i,j}$ requires a resource ℓ_q, it *issues* a *request* $\mathcal{R}$ for ℓ_q. $\mathcal{R}$ is *satisfied* as soon as $\tau_{i,j}$ *holds* ℓ_q, and *completes* when $\tau_{i,j}$ *releases* ℓ_q. $\mathcal{R}$ is *active* from its issuance to its completion. $\tau_{i,j}$ must *wait* until $\mathcal{R}$ can be satisfied if it is held by another job. It may do so either by *busy-waiting* (or *spinning*) in a tight loop, or by being *suspended* by the OS until $\mathcal{R}$ is satisfied. We assume that if a job $\tau_{i,j}$ holds a resource ℓ_q, then it must be scheduled to execute.[3] A resource access is called a *critical section* (CS).

We assume that each job can request or hold at most one resource at a time, *i.e.,* resource requests are non-nested. We let N_i^q denote the maximum number of times a job of task τ_i requests ℓ_q, and let L_i^q denote the maximum length of such a request. We define L_i^q to be 0 if $N_i^q = 0$. Finally, we define $L_{max}^q = \max_{1 \leq i \leq n}\{L_i^q\}$, and $L_{max} = \max_{1 \leq q \leq n_r}\{L_{max}^q\}$, and let $L_{sum,h}^q$ be the sum of the h largest L_i^q values.

Example 2.6. Figure 2.6 illustrates the timeline of a resource request $\mathcal{R}$. Job $\tau_{i,j}$ issues $\mathcal{R}$ at time t_i. After issuing $\mathcal{R}$, the job is suspended, as the resource is held by some other jobs. At time t_s, the resource is granted

[3]This is a common assumption in work on synchronization. It is needed for shared data, but may be pessimistic for other shared resources such as I/O devices.

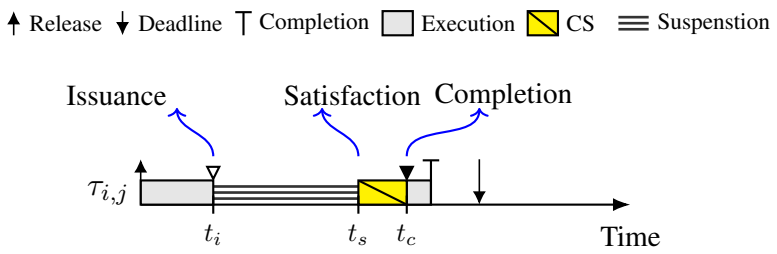


Figure 2.6: Timeline of a resource request.

to $\tau_{i,j}$ and $\mathcal{R}$ is satisfied. During $[t_s, t_c)$, the job executes its CS. The request $\mathcal{R}$ completes at time t_c when $\tau_{i,j}$ releases the resource. ◀

Priority inversions. Pi-blocking occurs when a job is delayed and this delay cannot be attributed to higher-priority demand for processing time. On multiprocessors, the formal definition of pi-blocking actually depends on how schedulability analysis is done. Of relevance to suspension-based locks, schedulability analysis may be either *suspension-oblivious* (*s-oblivious*) or *suspension-aware* (*s-aware*) [Brandenburg and Anderson, 2010a]. Under s-oblivious analysis, suspension time is *analytically* treated as computation time. In contrast, under s-aware analysis, suspension times are explicitly considered in schedulability analysis.

Definition 2.2 (S-oblivious pi-blocking [Brandenburg and Anderson, 2010a])**.** Under s-oblivious schedulability analysis for a global scheduler, a job $\tau_{i,j}$ incurs s-oblivious pi-blocking at time t if $\tau_{i,j}$ is **pending** but not scheduled and fewer than M higher-priority jobs are **pending**, where a job is considered pending at any time during $[r(\tau_{i,j}), f(\tau_{i,j}))$. ◀

By Definition 2.2, a job suffers s-oblivious pi-blocking under global scheduling if and only if it is one of the M highest-priority pending jobs but not scheduled.

Example 2.7. Figure 2.7 illustrates a G-EDF schedule of three jobs $\tau_{1,1}$, $\tau_{2,1}$, and $\tau_{3,1}$ on two processors. Job $\tau_{3,1}$ acquires resource ℓ at time 2. Job $\tau_{2,1}$ issues a request for ℓ at time 3, and it is suspended from time 3 to time 6. Since there is only one job with higher priority than $\tau_{2,1}$ during time interval $[3, 4)$, by Definition 2.2, $\tau_{2,1}$ incurs s-oblivious pi-blocking during this time interval. At time 4, job $\tau_{1,1}$, which has higher priority than both $\tau_{2,1}$ and $\tau_{3,1}$, is released. Since there are $M = 2$ jobs with higher-priority than $\tau_{2,1}$ during time interval $[4, 6)$, $\tau_{2,1}$ does not incur s-oblivious pi-blocking during this time interval. ◀

In contrast, s-aware pi-blocking is defined as follows.

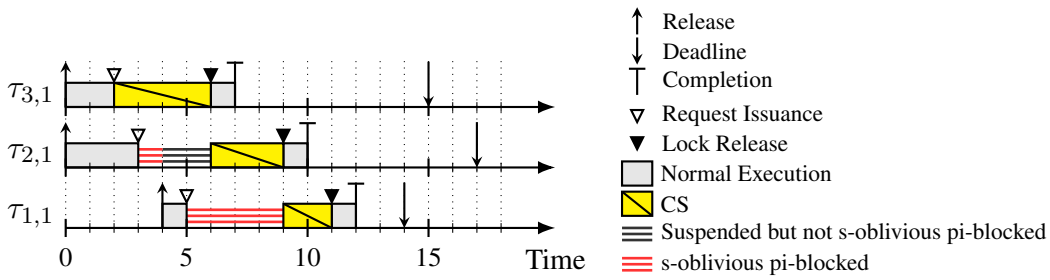


Figure 2.7: A schedule illustrating s-oblivious pi-blocking.

Definition 2.3 (S-aware pi-blocking [Brandenburg and Anderson, 2010a])**.** Under s-aware schedulability analysis for a global scheduler, a job $\tau_{i,j}$ incurs s-aware pi-blocking at time t if $\tau_{i,j}$ is **pending** but not scheduled and fewer than M higher-priority jobs are **ready**. Here, a job is ready when it is pending but not suspended. ◀

In Figure 2.7, job $\tau_{2,1}$ is suspended during the interval $[4, 5)$, but both $\tau_{1,1}$ and $\tau_{2,1}$ are ready during this interval. Thus, $\tau_{2,1}$ is not s-aware pi-blocked in $[4, 5)$. Since $\tau_{1,1}$ is suspended at time 5, only one higher-priority job is ready at that time. Therefore, $\tau_{2,1}$ is s-aware pi-blocked during $[5, 6)$.

Blocking complexity. Request lengths are unavoidable in assessing maximum pi-blocking, as a request-issuing job may have to wait for a current resource holder to complete before its request can be satisfied. As such, maximum pi-blocking bounds are usually expressed as an integer multiple of the maximum request length, *i.e.,* the number of requests that are satisfied while a resource-requesting job is pi-blocked.

Progress mechanism. Any real-time locking protocol needs to ensure a resource-holding job's progress whenever a job waiting for the same resource is pi-blocked, for otherwise, the maximum per-job pi-blocking can be very large or even unbounded. For example, consider a scenario where the lowest-priority job is holding a resource while the highest-priority job is waiting for the same resource. Suppose that all other tasks have ready jobs with higher priorities than the resource-holding job, but these jobs do not request the same resource. Without any mechanism to ensure the resource-holding job's progress, if jobs are preemptively scheduled based on their priorities, then the resource-holding job would be preempted by a higher-priority job. Consequently, the highest-priority job would need to wait until all other jobs complete execution, causing a very large pi-blocking time.

Therefore, to ensure that the maximum pi-blocking is reasonably bounded, real-time locking protocols employ *progress mechanisms* that *force* the execution of resource-holding jobs. The simplest way to force a

resource-holding job's execution is by making each CS non-preemptive. The non-preemptive execution of CSs has one major disadvantage: any higher-priority job, regardless of whether the job requires any resource or not, can be pi-blocked due to the resource-holder's non-preemptive execution.

Other common progress mechanisms usually raise a job's *effective priority* temporarily to force the execution of resource-holding jobs. Therefore, under such mechanisms, a job has an original *base* priority and an *effective* priority, which may differ in value depending on whether a locking protocol's progress mechanism elevates the effective priority.[4] One such a progress mechanism is *priority inheritance* [Sha et al., 1990; Rajkumar, 1991], which raises the effective priority of a job holding resource ℓ_q to the maximum of its priority and the priorities of all jobs waiting for ℓ_q. Priority inheritance is supported in different OSs, *e.g.,* `rt_mutex` in Linux uses it [The Linux Kernel Organization, 2024]. Although priority inheritance is effective under global scheduling, it does not work well under partitioned or clustered scheduling [Brandenburg and Anderson, 2011]. This is because the priority values may be analytically incomparable across processor (resp., cluster) boundaries under partitioned (resp., clustered) scheduling.

Priority boosting [Rajkumar et al., 1988; Rajkumar, 1990, 1991; Brandenburg and Anderson, 2010a], in contrast, works well under partitioned scheduling. Priority boosting unconditionally elevates a resource-holding job's effective priority above the highest possible non-boosted priority. Thus, priority boosting is technically similar to non-preemptive CSs. However, under priority boosting, jobs remain preemptive, *e.g.,* a job holding resource ℓ_q can have higher boosted priority (thus, preempt) than a job holding another resource ℓ_p. Priority boosting still has a similar issue to non-preemptive resource accesses, as a resource-holder's priority is boosted regardless of whether any job is pi-blocked for the same resource or not.

Another alternative is *priority donation* [Brandenburg and Anderson, 2014], which ensures that a job $\tau_{i,j}$ can only issue a request when its priority is high enough to be scheduled. Moreover, if a job $\tau_{k,\ell}$'s release causes $\tau_{i,j}$ to have insufficient priority to be scheduled, then $\tau_{k,\ell}$ "donates" its priority to $\tau_{i,j}$. This ensures that a resource holder is always scheduled. This property makes priority donation particularly effective under clustered scheduling.

Allocation inheritance [Holman and Anderson, 2006] is another progress mechanism that solves the incomparable priority problem of priority inheritance. In addition to raising the resource-holding job's priority to the highest-priority pi-blocked job waiting for the same resource, allocation inheritance allows

[4]S-oblivious schedulability analysis is performed assuming that jobs are scheduled by their base priorities. Thus, the definition of s-oblivious pi-blocking is based on base priorities.

the resource-holding job to execute on the processor where the pi-blocked job is supposed to execute. This idea also appeared under different names: *local helping* [Hohmuth and Härtig, 2001], *migratory priority inheritance* [Brandenburg, 2013a], *multiprocessor bandwidth inheritance* [Faggioli et al., 2010], *etc.*

Request vs. release blocking. As should be obvious by now, a job may experience pi-blocking each time it requests a resource—this is called *request blocking*. A locking protocol may also cause another type of pi-blocking, called *release blocking*, when a job is released. For example, with non-preemptive resource accesses, a newly released higher-priority job may not execute upon its release. Similarly, under priority donation, a newly released job may donate its priority and suffer from release blocking.

2.3.1 S-Oblivious Pi-Blocking Bounds

Multiprocessor real-time locking protocols and their s-oblivious pi-blocking bounds have mostly been studied under JLFP schedulers.

Lower bound. For mutex locks, a trivial lower bound of $M - 1$ request lengths on per-request s-oblivious pi-blocking under any JLFP scheduler is known [Brandenburg and Anderson, 2010a]. This lower-bound result can be seen from the following scenario. Consider that M jobs are released synchronously, and each of them issues a request for a resource ℓ_q immediately after being scheduled. Since no two jobs can access the resource at the same time, the request of one of these M jobs is satisfied last. This job is one of the M highest-priority jobs throughout its suspension times, incurring a pi-blocking of $M - 1$ request lengths. Recently, a lower bound of $M + M(H_{N-1} - H_M)$ on per-request s-oblivious pi-blocking under any non-JLFP scheduler has been shown [Tong et al., 2025], where $H_i = 1 + 1/2 + \ldots + 1/i$.

Contribution of this dissertation. Regarding s-oblivious pi-blocking for mutex locks, we proved a non-trivial lower bound of $2M - 2$ request lengths on per-request s-oblivious pi-blocking under a class of JLFP schedulers that includes G-EDF and G-FP but not G-FIFO scheduling.

Upper bound. Pi-blocking upper-bound results are typically established by designing locking protocols that achieve such results. Since the general per-request pi-blocking lower bound is $M - 1$ request lengths, any multiprocessor real-time locking protocol that achieves per-request pi-blocking of $M - 1$ (resp., $O(M)$) request lengths are considered *optimal* (resp., *asymptotically optimal*). Multiprocessor locking protocols such as the G-OMLP [Brandenburg and Anderson, 2010a], the C-OMLP [Brandenburg and Anderson, 2011], and the OMIP [Brandenburg, 2013a] are asymptotically optimal under any JLFP scheduling algorithm. Note

Table 2.4: Asymptotically optimal locking protocols for mutex locks under s-oblivious schedulability analysis for JLFP scheduling.

Scheduling	Protocol	Release blocking	Request blocking
Global JLFP	G-OMLP [Brandenburg and Anderson, 2010a]	0	$(2M-1)L^q_{max}$
Clustered JLFP	C-OMLP [Brandenburg and Anderson, 2011]	ML_{max}	$(M-1)L^q_{max}$
Clustered JLFP	OMIP [Brandenburg, 2013a]	0	$(2M-1)L^q_{max}$
C-FIFO	OLP-F (This dissertation)	0	$(M-1)L^q_{max}$

that the G-OMLP is applicable only under global scheduling. These locking protocols use different queue structures to order resource requests. Moreover, the G-OMLP, the C-OMLP, and the OMIP use priority inheritance, priority donation, and migratory priority inheritance, respectively, as their progress mechanisms. Despite such differences, when each job issues only one request, all these locking protocols provide a pi-blocking upper bound of $2M-1$ request lengths. Table 2.4 provides a summary of these pi-blocking bounds of these asymptotically optimal locking protocols. Note that, for the C-OMLP, the pi-blocking bound of $2M-1$ request lengths comes from a combination of release and request blocking. Recently, a locking protocol, called the NJLP, has been proposed, which achieves an asymptotically optimal s-oblivious pi-blocking bound under non-JLFP scheduling [Tong et al., 2025]. Locking protocols and their corresponding s-oblivious pi-blocking bounds have also been studied for the particular application of accessing a GPU using a lock [Ali et al., 2024].

Contribution of this dissertation. In this dissertation, we propose the *optimal locking protocol for mutual exclusion sharing under C-FIFO scheduling* (OLP-F), which achieves optimal pi-blocking under C-FIFO scheduling. To match the lower bound on pi-blocking, the OLP-F ensures that each job suffers pi-blocking for the duration of at most $M-1$ request lengths and incurs no release blocking (see Table 2.4). Thus, the OLP-F is an optimal locking protocol under C-FIFO scheduling. Note that clustered scheduling generalizes both global and partitioned scheduling, so the OLP-F is also optimal under G-FIFO and P-FIFO scheduling.

2.3.2 S-Aware Pi-Blocking Bounds

Many locking protocols have been studied under s-aware analysis. Many of these protocols (*e.g.,* the *multiprocessor priority ceiling protocol* (MPCP) [Rajkumar et al., 1988], the *parallel priority ceiling protocol*

(PPCP) [Easwaran and Andersson, 2009], the *priority inheritance protocol* (PIP) [Rajkumar, 1991], *etc.*) were inspired by classical uniprocessor locking protocols. Under s-aware analysis, an $\Omega(n)$ lower bound on pi-blocking has been established [Brandenburg and Anderson, 2010a]. The FMLP$^+$ [Brandenburg, 2014] is an extension of the *flexible multiprocessor locking protocol* (FMLP), which achieves asymptotically optimal s-aware pi-blocking under clustered JLFP scheduling. Later, linear-programming techniques were shown to improve the s-aware analysis of various protocols, including the PIP, the PPCP, and the FMLP, under G-FP and P-FP scheduling [Brandenburg, 2013b; Yang et al., 2015]. While s-oblivious pi-blocking bounds have not been studied under semi-partitioned scheduling, the *migration-based locking protocol under semi-partitioned scheduling* (MLPS) and the *non-migration-based locking protocol under semi-partitioned scheduling* (NMLPS) were proposed and analyzed under s-aware schedulability analysis [Afshar et al., 2012].

2.4 Gang Tasks

The sporadic gang task model is a parallel task model that generalizes the sporadic task model. A job of a gang task consists of multiple co-scheduled threads and requires a specified number of processors for execution. Based on the flexibility allowed in determining the number of processors assigned to a job, gang tasks can be classified as follows [Goossens and Richard, 2016].

- **Rigid.** The number of processors assigned to a rigid gang task is specified externally to the scheduler a priori and does not change at runtime. Thus, all jobs of a rigid gang task require the same number of processors.
- **Moldable.** The number of processors assigned to a job of a moldable gang task is determined by the scheduler, but remains fixed during the execution of the job. Different jobs of the same task may be assigned different numbers of processors. Typically, a minimum and maximum number of processors are specified a priori for each moldable gang task.
- **Malleable.** The number of processors assigned to a job can be adjusted by the scheduler at runtime.

In the following section, we review prior work on gang scheduling and discuss our contributions.

2.4.1 Prior Work

Gang tasks were introduced by Ousterhout *et al.* to reduce the bottleneck in interprocess communication [Ousterhout, 1982]. It is known that determining HRT-feasibility and finding an HRT-optimal schedule of preemptive and non-preemptive sporadic rigid gang tasks are NP-hard when the number of processors is part of the input [Blazewicz et al., 1986; Kubale, 1987; Goossens and Richard, 2016]. We now describe different known schedulability results for gang scheduling.

HRT scheduling of preemptive rigid gang. To our knowledge, all work on the preemptive scheduling of rigid gang tasks considers constrained-deadline systems. Goossens *et al.* showed that preemptive gang scheduling under work-conserving JLFP scheduling is not sustainable [Goossens and Richard, 2016]. They also gave a linear-program-based optimal scheduler for implicit-deadline periodic gang tasks, which runs in polynomial time when the number of processors is constant [Goossens and Richard, 2016]. However, the scheduler requires execution for full WCETs for all jobs or idling processors for some duration if a job finishes before its WCET to avoid timing anomalies. They also proposed an FP scheduler and determined its speed-up factor. Goossens and Berten devised an exact HRT-schedulability test for constrained-deadline periodic gang tasks under a class of preemptive JLFP schedulers based on the schedule repetition property [Goossens and Berten, 2010]. A sufficient HRT-schedulability test for constrained-deadline sporadic gang tasks under G-EDF was first proposed in [Kato and Ishikawa, 2009], which was later proved to be incorrect [Richard et al., 2017]. HRT-schedulability tests under G-EDF were also proposed in [Dong and Liu, 2019; Lee et al., 2022b]. Response-time analysis for constrained-deadline sporadic gang tasks under G-FP scheduling was studied in [Lee et al., 2022b]. Stationary scheduling and strict-partitioning scheduling were considered for constrained-deadline rigid gang tasks in [Ueter et al., 2021; Sun et al., 2024a]. Other work considered scheduling only one rigid gang task at a time [Ali et al., 2021] and the *mixed-criticality* scheduling of rigid gang tasks [Bhuiyan et al., 2019].

HRT scheduling of non-preemptive rigid gang tasks. Work-conserving scheduling of non-preemptive rigid gang tasks can cause pi-blocking when a low-priority job that requires fewer processors is scheduled instead of a high-priority job that requires more. Unfortunately, such pi-blocking can have a transitive effect, which is known as 2D-blocking [Dong and Liu, 2022]. A sufficient HRT-schedulability test for constrained-deadline rigid gang tasks was given in [Dong and Liu, 2022]. Schedulability under G-FP schedulers for constrained-

deadline rigid gang tasks was considered in [Lee et al., 2022a]. P-FP scheduling of constrained-deadline rigid gang tasks was considered in [Sun et al., 2024a,b].

HRT scheduling of moldable or malleable gang tasks. A greedy scheduler and a corresponding sufficient HRT-schedulability test for preemptive moldable gang tasks were proposed in [Lee et al., 2011]. Collette *et al.* gave an HRT-feasibility test and a scheduling algorithm that minimizes the number of processors required to schedule a set of preemptive malleable gang tasks [Collette et al., 2008]. Nelissen *et al.* gave a response-time analysis for non-preemptive gang tasks using the concept of schedule-abstraction graphs under work-conserving global JLFP schedulers [Nelissen et al., 2022].

SRT scheduling. SRT scheduling or HRT scheduling of arbitrary-deadline gang tasks has received little attention. While exact SRT-feasibility conditions for sporadic tasks and DAG tasks are known (Corollary 1.1 and Theorem 2.1), such conditions remain unknown for gang task models. The lone work on SRT gang scheduling [Dong et al., 2021] gives a sufficient condition for bounded response times and a response-time bound for gang tasks under preemptive G-EDF scheduling.

Bundled task model. The bundled task model is a recently introduced generalization of rigid gang tasks [Wasly and Pellizzoni, 2019]. In this model, a bundled task is represented as a chain of "bundles," where successive bundles have precedence constraints between them. Each bundle is a rigid gang task, but different bundles of the same task may have different processor requirements. The HRT scheduling of bundled tasks was studied under global and partitioned FP scheduling [Wasly and Pellizzoni, 2019; Rispo et al., 2024].

Contribution of this dissertation. In this dissertation, we consider the SRT-feasibility problem pertaining to sporadic rigid gang tasks. We derive a necessary and a sufficient SRT-feasibility condition and show that determining the SRT-feasibility of a sporadic rigid gang task system is NP-hard. We also show that G-EDF is not an SRT-optimal scheduler and derive a new condition for SRT-schedulability under G-EDF that theoretically dominates the condition in [Dong et al., 2021].

We also introduced a new task model in which rigid gang tasks can have precedence constraints among them. Thus, the model generalizes the DAG task model by allowing each node to be a gang task. We consider the HRT scheduling of such a task system on a heterogeneous platform consisting of multiple CEs. We give a response-time analysis for such DAGs, assuming that each DAG has a constrained deadline.

2.5 General Definitions and Notation

In this section, we provide definitions and notation that apply throughout the dissertation. Specific restrictions and assumptions will be explicitly mentioned when required.

We denote the i^{th} task of a task system by τ_i and the j^{th} job of τ_i by $\tau_{i,j}$. The WCET of τ_i is denoted by C_i. The largest and smallest WCETs among all tasks are denoted by C_{max} and C_{min}, respectively. The *offset* of a periodic task τ_i, denoted by Φ_i, is the release time of $\tau_{i,1}$. The largest and smallest offsets among all tasks are denoted by Φ_{max} and Φ_{min}, respectively. The *relative deadline* of τ_i is denoted by D_i. The parallelization level of τ_i is denoted by P_i.

The *release time*, *absolute deadline*, and *completion time* of job $\tau_{i,j}$ are denoted by $r(\tau_{i,j})$, $d(\tau_{i,j})$, and $f(\tau_{i,j})$, respectively. The *response time* of $\tau_{i,j}$ is denoted by $R(\tau_{i,j}) = f(\tau_{i,j}) - r(\tau_{i,j})$. The response time of τ_i is $R(\tau_i) = \sup_j R(\tau_{i,j})$. The *tardiness* of a job $\tau_{i,k}$ is defined as $\max\{0, f(\tau_{i,j}) - d(\tau_{i,j})\}$. The tardiness of task τ_i is the maximum tardiness among any of its jobs. Based on job release and finish times, we define *pending* and *ready* jobs.[5]

Definition 2.4 (Pending job)**.** A job $\tau_{i,j}$ is *pending* at time t in a schedule $\mathcal{S}$ if and only if $r(\tau_{i,j}) \leq t < f(\tau_{i,j})$. ◀

Definition 2.5 (Ready job)**.** A job $\tau_{i,j}$ is *ready* at time t in a schedule if and only if it is pending and one of the following two holds.

(i) $j \leq P_i$.

(ii) $f(\tau_{i,j-P_i}) \leq t$. ◀

Thus, a job is pending if it has been released but yet to complete execution. A pending job can be ready or not. The job is ready, *i.e.,* can be scheduled, if at most $P_i - 1$ prior jobs of τ_i are pending. Otherwise, the job is not ready to execute. Unless otherwise stated, we assume that tasks are scheduled by preemptive *global-EDF-like* (GEL) schedulers. We also assume time to be discrete and a unit of time is 1.0.

The *utilization* of τ_i is $u_i = C_i/T_i$. The *total utilization* of the task system Γ is $U_{tot} = \sum_{i=1}^{N} u_i$. We require $u_i \leq P_i$ and $U_{tot} \leq M$ to hold, which is the exact condition for bounded response times for sporadic tasks (Corollary 1.1). The *hyperperiod* H is the LCM of all periods. The periods are *pseudo-harmonic* when each period divides $\max_i\{T_i\}$, *i.e.,* $H = \max_i\{T_i\}$ holds.

[5]In the later chapters, the definition of pending and ready jobs will be amended (if needed) to deal with other features.

2.6 Chapter Summary

In this chapter, we reviewed the task models considered in this dissertation. We also reviewed existing work on scheduling sequential tasks, DAG tasks, gang tasks, and mutex resources. Furthermore, we highlighted the contributions of this dissertation in the context of existing work on each task model.

CHAPTER 3: RESPONSE-TIME BOUND FOR PSEUDO-HARMONIC SEQUENTIAL TASKS[1]

In this chapter, we give response-time bounds of periodic tasks scheduled by global-EDF-like (GEL) schedulers on an identical multiprocessor platform. For pseudo-harmonic periodic tasks, we first give a polynomial-time computable bound that is tight within a constant factor. This bound does not increase with the processor count for pseudo-harmonic periodic tasks. We then give a simulation-based exact response-time analysis for periodic tasks. This analysis can be performed in pseudo-polynomial time for pseudo-harmonic tasks.

Organization. In the rest of this chapter, we give the system model considered in this chapter (Section 3.1), derive a response-time bound for GEL schedulers (Section 3.2), show how to determine exact response-time bounds via schedule simulation (Section 3.3), discuss our experimental results (Section 3.4), and provide a summary (Section 3.5).

3.1 System Model

In this section, we provide needed assumptions and definitions. Table 3.1 summarizes the notation used in this chapter.

Task model. We consider a task system Γ consisting of N periodic *tasks* $\tau_1, \tau_2, \ldots, \tau_N$ to be scheduled on M identical processors. Each task τ_i releases a potentially infinite sequence of *jobs* $\tau_{i,1}, \tau_{i,2}, \ldots$. The *period* of task τ_i, denoted by T_i, is the separation time between two consecutive job releases by it. The largest period among all tasks is denoted by T_{max}. We assume that tasks have no parallelism, *i.e.,* $\forall i : P_i = 1$ holds. Therefore, $\tau_{i,j+1}$ cannot start execution before $\tau_{i,j}$ completes. The WCET of τ_i is denoted by C_i. The *offset* of a periodic task τ_i, denoted by Φ_i, is the release time of $\tau_{i,1}$. The largest offset among all tasks is denoted by Φ_{max}. The *relative deadline* of τ_i is $D_i = T_i$. For brevity, we denote a periodic task τ_i by (Φ_i, C_i, T_i).

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

Ahmed, S. and Anderson, J. (2021), Tight Tardiness Bounds for Pseudo-Harmonic Tasks under Global-EDF-Like Schedulers, *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*, pages 11:1–11:24.

Table 3.1: Notation summary for Chapter 3.

Symbol	**Meaning**
Γ	Task system
N	Number of tasks
M	Number of processors
τ_i	i^{th} task
T_i	Period of τ_i
C_i	WCET of τ_i
Φ_i	Offset of τ_i
Y_i	RPP of τ_i
u_i	Utilization of τ_i
$R(\tau_i)$	Response time of τ_i
(Φ_i, C_i, T_i)	Task τ_i
U_{tot}	Utilization of Γ
H	Hyperperiod of Γ
h_i	H/T_i
T_{max}	$\max_i\{T_i\}$
Φ_{max}	$\max_i\{\Phi_i\}$
Y_{max}	$\max_i\{Y_i\}$
Y_{min}	$\min_i\{\Phi_i\}$
$\tau_{i,j}$	j^{th} job of τ_i
$r(\tau_{i,j})$	Release time of $\tau_{i,j}$
$f(\tau_{i,j})$	Completion time of $\tau_{i,j}$
$y(\tau_{i,j})$	PP of $\tau_{i,j}$
$R(\tau_{i,j})$	Response time of $\tau_{i,j}$
$\mathcal{S}$	An arbitrary schedule
$\mathcal{I}$	Ideal schedule
$\mathsf{A}(\tau_i, t, t', \mathcal{S})$	Allocation of τ_i in $\mathcal{S}$ (Definition 3.1)
$\mathsf{A}(\Gamma, t, t', \mathcal{S})$	Allocation of Γ in $\mathcal{S}$ (Definition 3.1)
$\mathsf{lag}(\tau_i, t, \mathcal{S})$	lag of τ_i in $\mathcal{S}$ (3.3)
$\mathsf{LAG}(\Gamma, t, \mathcal{S})$	LAG of Γ in $\mathcal{S}$ (3.5)

The *utilization* of τ_i is $u_i = C_i/T_i$. The *total utilization* of the task system Γ is $U_{tot} = \sum_{i=1}^{N} u_i$. We require $u_i \leq 1.0$ and $U_{tot} \leq M$ to hold, which is the exact condition for bounded response times for sporadic tasks (Corollary 1.1). The *hyperperiod* H is the LCM of all periods. The periods are *pseudo-harmonic* when each period divides $\max_i\{T_i\}$, *i.e.,* $H = T_{max}$ holds.

Scheduling. We assume that jobs of Γ are scheduled by a preemptive GEL scheduler. Under GEL scheduling, each task τ_i has a relative PP Y_i. We assume that $Y_i \geq 0$ holds for each task τ_i. The maximum and minimum relative PP among all tasks in Γ are denoted by Y_{max} and Y_{min}, respectively. The PP of a job $\tau_{i,j}$, denoted by $y(\tau_{i,j})$, is defined as

$$y(\tau_{i,j}) = r(\tau_{i,j}) + Y_i. \tag{3.1}$$

Under GEL scheduling, jobs with earlier PP has higher priority. We assume ties to be broken arbitrarily but consistently by task index. Therefore, jobs are prioritized according to the following rule.

PR. Job $\tau_{i,j}$ has higher priority than job $\tau_{k,\ell}$ if and only if $(y(\tau_{i,j}) < y(\tau_{k,\ell})) \vee (y(\tau_{i,j}) = y(\tau_{k,\ell}) \wedge i < k)$.

We now illustrate the concept of LAG, which we heavily use in deriving our bounds.

3.1.1 The Concept of LAG

The concept of LAG is pivotal in the design and analysis many real-time scheduling algorithms [Baruah et al., 1996; Stoica et al., 1996; Devi and Anderson, 2008]. To define LAG, we first introduce the concept of an *ideal* schedule. We will then define LAG by comparing an arbitrary schedule with such an ideal schedule.

Ideal schedule. Let $\widehat{\pi}_1, \widehat{\pi}_2, \ldots, \widehat{\pi}_N$ be N processors with speeds $u_1, u_2, \ldots, u_N$, respectively. Thus, these processors represent a uniform multiprocessor platform. In an *ideal schedule* $\mathcal{I}$, each task τ_i is partitioned to execute on processor $\widehat{\pi}_i$. Each job starts execution as soon as it is released and completes execution by its deadline in $\mathcal{I}$. Therefore, the following property holds.

Property 3.1. *Each job $\tau_{i,k}$ finishes by time $r(\tau_{i,k}) + T_i$ in $\mathcal{I}$.*

Example 3.1. Consider a periodic task system Γ with tasks $\tau_1 = (0, 2, 3)$, $\tau_2 = (0, 2, 3)$, and $\tau_3 = (0, 4, 6)$. Figure 3.1(a) shows an ideal schedule $\mathcal{I}$ for this task system. Since τ_2's utilization is 2/3, it continuously executes on a processor with speed 2/3. ◀

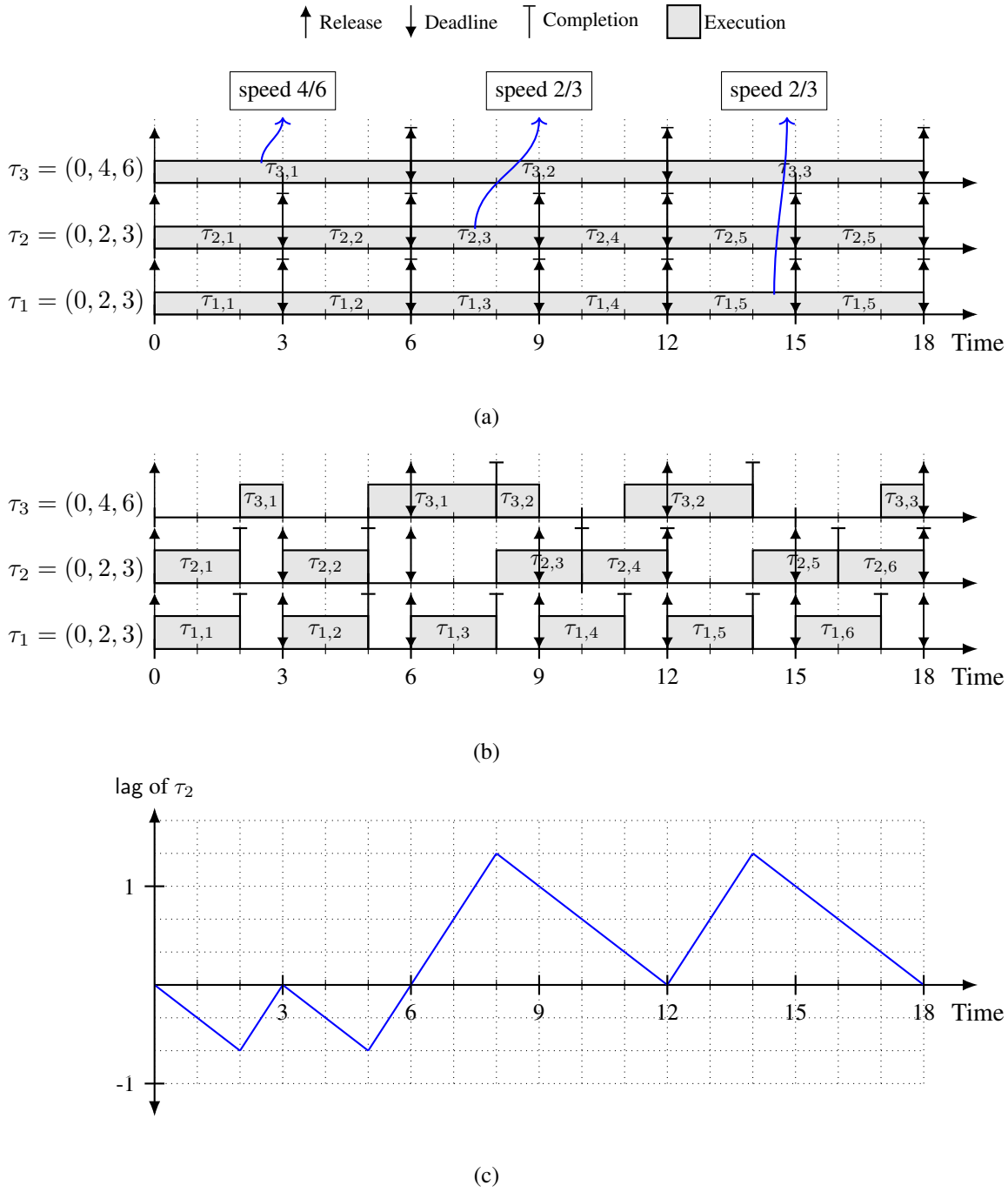


Figure 3.1: (a) An ideal schedule, (b) a G-EDF schedule of the task system in Example 3.1. (c) lag of τ_2.

To represent how much a task executes during an arbitrary interval, we now introduce the concept of *allocation*. Since an ideal schedule assumes uniform multiprocessors, we give a general definition of allocation that applies also for any arbitrary schedule on uniform multiprocessors.

Definition 3.1 (Allocation)**.** Suppose a task τ_i (resp., all tasks of Γ) executes for $x_{i,k}$ (resp., X_k) time units on a processor π_k with speed s_k over a time interval $[t, t')$ in a schedule $\mathcal{S}$. The *cumulative processor capacity* allocated to a task τ_i in schedule $\mathcal{S}$ over an interval $[t, t')$, denoted by $\mathsf{A}(\tau_i, t, t', \mathcal{S})$, is $\sum_k x_{i,k} s_k$. The cumulative processor capacity allocated to task system in a schedule $\mathcal{S}$ over an interval $[t, t')$, denoted by $\mathsf{A}(\Gamma, t, t', \mathcal{S})$, is $\sum_k X_k s_k$. Thus,

$$\mathsf{A}(\Gamma, t, t', \mathcal{S}) = \sum_{\tau_i \in \Gamma} \mathsf{A}(\tau_i, t, t', \mathcal{S}). \tag{3.2}$$

◀

Properties of an ideal schedule $\mathcal{I}$ can be expressed using the concept of allocation. A task τ_i executes at a speed of u_i in $\mathcal{I}$. Therefore, $\mathsf{A}(\tau_i, t, t', \mathcal{I}) \leq u_i(t' - t)$. Moreover, since $U_{tot} = \sum_{i=1}^{N} u_i$, for task system Γ, we have $\mathsf{A}(\Gamma, t, t, \mathcal{I}) \leq U_{tot}(t' - t)$. If τ_i is periodic and each job executes for its WCET, then $\mathsf{A}(\tau_i, t, t', \mathcal{I}) = u_i(t' - t)$ where $t, t' \geq \Phi_i$.

Example 3.1 (Continued)**.** In the ideal schedule $\mathcal{I}$ shown in Figure 3.1(a), τ_2 continuously executes at speed 2/3. Thus, τ_2's allocation in $\mathcal{I}$ during the interval $[0, 5)$ is $\mathsf{A}(\tau_2, 0, 5, \mathcal{I}) = 5 \cdot \frac{2}{3} = 10/3$. Since the total utilization of the task system is $2/3 + 2/3 + 4/6 = 2$, allocation of Γ during each time unit is 2.0. Therefore, $\mathsf{A}(\Gamma, 0, 5, \mathcal{I}) = 5 \cdot 2 = 10.0$.

A G-EDF schedule $\mathcal{S}$ for the same task system on two unit-speed processors is shown in Figure 3.1(b). In $\mathcal{S}$, task τ_2 executes for 4.0 time units during the interval $[0, 5)$. Since all processors have unit speeds in $\mathcal{S}$, τ_2's allocation in $\mathcal{S}$ during the interval $[0, 5)$ is $\mathsf{A}(\tau_2, 0, 5, \mathcal{S}) = 4.0$. During the interval $[0, 5)$, all processors are busy except the sub-interval $[2, 3)$. During the interval $[2, 3)$, one processor is busy. Therefore, $\mathsf{A}(\Gamma, 0, 5, \mathcal{S}) = 9.0$. ◀

lag and LAG. We are now ready to define lag and LAG. The lag of a task τ_i at time t in a schedule $\mathcal{S}$ is defined as

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

Thus, lag defines how much "ahead" or "behind" a task's execution is in $\mathcal{S}$ compared to an ideal execution of the task to meet all deadlines in $\mathcal{I}$. A positive lag means that the task has executed less in $\mathcal{S}$ than $\mathcal{I}$, so $\mathcal{S}$ has allocated less processor capacity to the task than $\mathcal{I}$. A negative value means the opposite that $\mathcal{S}$ over-allocates

processor capacity to the task. Since $\mathsf{lag}(\tau_i, 0, \mathcal{S}) = 0$, for $t' \geq t$ we have

$$\mathsf{lag}(\tau_i, t', \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S}) + \mathsf{A}(\tau_i, t, t', \mathcal{I}) - \mathsf{A}(\tau_i, t, t', \mathcal{S}). \tag{3.4}$$

The LAG of a task system Γ in a schedule $\mathcal{S}$ at time t is defined as

$$\mathsf{LAG}(\Gamma, t, \mathcal{S}) = \sum_{\tau_i \in \Gamma} \mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{A}(\Gamma, 0, t, \mathcal{I}) - \mathsf{A}(\Gamma, 0, t, \mathcal{S}). \tag{3.5}$$

Similarly, LAG defines how much "ahead" or "behind" the total execution of all tasks of Γ in $\mathcal{S}$ compared to their ideal execution in $\mathcal{I}$. Since $\mathsf{LAG}(\Gamma, 0, \mathcal{S}) = 0$, for $t' \geq t$ we have

$$\mathsf{LAG}(\Gamma, t', \mathcal{S}) = \mathsf{LAG}(\Gamma, t, \mathcal{S}) + \mathsf{A}(\Gamma, t, t', \mathcal{I}) - \mathsf{A}(\Gamma, t, t', \mathcal{S}). \tag{3.6}$$

Example 3.1 (Continued)**.** Since τ_2's allocation over interval $[0, 5)$ in $\mathcal{S}$ and $\mathcal{I}$ are 4.0 and 10/3, respectively, τ_2's lag in $\mathcal{S}$ at time 5 is $\mathsf{lag}(\tau_2, 5, \mathcal{S}) = 10/3 - 4 = -2/3$. Note that both jobs of τ_2 that are released before time 5 complete their execution in $\mathcal{S}$. However, the second job is still pending in $\mathcal{I}$ at time 5. Figure 3.1(c) shows τ_2's lag values with respect to time.

Similarly, the LAG of the task system Γ at time 5 is 1.0. This one unit of LAG is due to the one idle processor in $\mathcal{S}$ during the interval $[2, 3)$. ◀

3.2 Response-Time Bound

In this section, we derive a polynomial-time computable response-time bound for periodic task systems under GEL schedulers. We derive the bound for any periodic task system and then show its tightness for periodic systems with pseudo-harmonic periods. We assume $N > M$; otherwise, every job starts execution upon its release, causing a response time of at most its task's WCET. We initially assume the following, which we relax later.

Assumption 3.1. Each job of any task τ_i executes for its WCET C_i.

To prove our bound, we consider an arbitrary GEL schedule $\mathcal{S}$ and an ideal schedule $\mathcal{I}$ of task system Γ under Assumption 3.1. Under this assumption, by the definition of $\mathcal{I}$, we have

$$\forall t \geq \Phi_i : \mathsf{A}(\tau_i, 0, t, \mathcal{I}) = (t - \Phi_i)u_i. \tag{3.7}$$

We derive our response-time bound (Theorem 3.1) by giving an upper bound on per-task lag (Lemma 3.20) using a lag-*monotonicity* property (Lemma 3.13).[2] Informally, the lag-monotonicity property states that no task τ_i receives more allocation in $\mathcal{S}$ than $\mathcal{I}$, *i.e.,* lag does not decrease, over any interval of length H beginning at or after Φ_i. This can be observed in Figure 3.1(c) by checking lag values of time instants separated by $H = 6$ time units, *e.g.,* τ_2's lag at times 3, 9, and 15 are 0, 1, and 1, respectively. We first establish the lag-monotonicity property using a series of properties of lag proved in Section 3.2.1. We then use the lag-monotonicity property to derive our response-time bound in Section 3.2.2.

3.2.1 Properties of lag

We begin by proving some properties of lag. Since the lag-monotonicity property compares lag values between two time instants, we first establish several properties concerning such comparisons between a pair of lag values (Lemmas 3.8–3.12) based on the simpler properties of lag (Lemmas 3.1–3.7).

The following lemma is based on the fact that the lag of a task τ_i decreases at a rate of $(1 - u_i)$ when it is scheduled. This can be seen in Figure 3.1(c). Since τ_2 has a utilization of $2/3$, its lag decreases at a rate of $1 - 2/3 = 1/3$ whenever it is scheduled. In contrast, a task's lag always increases at a rate of u_i when it is not scheduled.

Lemma 3.1. *For any task τ_i and interval $[t, t')$ with $t \geq \Phi_i$, the following hold.*

(a) If τ_i continuously executes during $[t, t')$ in $\mathcal{S}$, then $\mathsf{lag}(\tau_i, t', \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S}) - (t' - t)(1 - u_i)$.

(b) If τ_i does not execute during $[t, t')$ in $\mathcal{S}$, then $\mathsf{lag}(\tau_i, t', \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S}) + (t' - t)u_i$.

Proof. Since $t \geq \Phi_i$, by the definition of $\mathcal{I}$, we have $\mathsf{A}(\tau_i, t, t', \mathcal{I}) = (t' - t)u_i$.

[2] In Chapter 4, we will show that the same property (Lemma 4.25) holds when tasks have arbitrary parallelization levels, *i.e.,* P_i (defined in Section 2.1) successive jobs of a task can execute concurrently. Thus, the proof in Chapter 4 also works for the case considered in this chapter. However, some lemmas (*e.g.,* Lemma 3.6) proved in this chapter do not generalize to arbitrary parallelization levels.

(a) Since τ_i continuously executes throughout $[t, t')$ in $\mathcal{S}$, $\mathsf{A}(\tau_i, t, t', \mathcal{S}) = (t' - t)$ holds. Substituting $\mathsf{A}(\tau_i, t, t', \mathcal{I})$ and $\mathsf{A}(\tau_i, t, t', \mathcal{S})$ in (3.4), we have $\mathsf{lag}(\tau_i, t', \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S}) + (t' - t)u_i - (t' - t) = \mathsf{lag}(\tau_i, t, \mathcal{S}) - (t' - t)(1 - u_i)$.

(b) Since τ_i does not execute during $[t, t')$ in $\mathcal{S}$, we have $\mathsf{A}(\tau_i, t, t', \mathcal{S}) = 0$. Substituting $\mathsf{A}(\tau_i, t, t', \mathcal{I})$ and $\mathsf{A}(\tau_i, t, t', \mathcal{S})$ in (3.4), we have $\mathsf{lag}(\tau_i, t', \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S}) + (t' - t)u_i - 0 = \mathsf{lag}(\tau_i, t, \mathcal{S}) + (t' - t)u_i$. □

When the lag of a task is positive, some of its jobs are released but not finished. The following lemma shows this.

Lemma 3.2 ([Yang and Anderson, 2017], Lemma 1)**.** *If $\mathsf{lag}(\tau_i, t, \mathcal{S}) > 0$, then τ_i has a pending job at t in $\mathcal{S}$.*

The following lemma considers lag values of a task at period boundaries in $\mathcal{S}$. In $\mathcal{I}$, every job completes execution at a period boundary when the next job of the task is released. Therefore, the lag of any task must be non-negative at period boundaries; otherwise, a not-yet-released job would have to execute at some point in $\mathcal{S}$, which is impossible. This property can be seen at times $\{0, 3, 6, \ldots\}$ for τ_2 in Figure 3.1(c).

Lemma 3.3. *For any task τ_i and non-negative integer c, $\mathsf{lag}(\tau_i, \Phi_i + cT_i, \mathcal{S}) \geq 0$.*

Proof. The lemma trivially holds for $c = 0$. Assume that there is a task τ_i and an integer $c \geq 1$ such that $\mathsf{lag}(\tau_i, \Phi_i + cT_i, \mathcal{S}) < 0$ holds. Then, by (3.3), $\mathsf{A}(\tau_i, 0, \Phi_i + cT_i, \mathcal{S}) > \mathsf{A}(\tau_i, 0, \Phi_i + cT_i, \mathcal{I})$ holds. Since τ_i releases jobs periodically, jobs $\tau_{i,c}$ and $\tau_{i,c+1}$ are released at times $\Phi_i + (c-1)T_i$ and $\Phi_i + cT_i$, respectively. By Property 3.1, job $\tau_{i,c}$ completes execution at time $\Phi_i + cT_i$ in $\mathcal{I}$. Therefore, all jobs of τ_i released before time $\Phi_i + cT_i$ complete execution by time $\Phi_i + cT_i$ in $\mathcal{I}$. Since $\tau_{i,c}$ cannot execute before its release, $\mathsf{A}(\tau_i, 0, \Phi_i + cT_i, \mathcal{S})$ cannot be larger than $\mathsf{A}(\tau_i, 0, \Phi_i + cT_i, \mathcal{I})$, a contradiction. □

Lemmas 3.4–3.7 give relationships among a task τ_i's lag at time t, its utilization, and the release time of a job of τ_i. We prove these lemmas by expressing τ_i's allocation by time t in terms of τ_i's utilization and the release time of a job of τ_i.

Lemma 3.4. *If τ_i has no pending job at time $t \geq \Phi_i$ in $\mathcal{S}$ and $r(\tau_{i,k}) \leq t < r(\tau_{i,k+1})$ holds, then $\mathsf{lag}(\tau_i, t, \mathcal{S}) = (t - r(\tau_{i,k+1}))u_i$.*

Proof. Since τ_i has no pending job at time t, $\tau_{i,k}$ and all prior jobs of τ_i complete execution at or before time t. Since $\tau_{i,k+1}$ is released at time $r(\tau_{i,k+1}) = r(\tau_{i,k}) + T_i > t$, no job released after $r(\tau_{i,k})$ executes before time t. Hence, $\mathsf{A}(\tau_i, 0, t, \mathcal{S}) = \sum_{j=1}^{k} C_i = \sum_{j=1}^{k} T_i u_i = \sum_{j=1}^{k} (r(\tau_{i,j+1}) - r(\tau_{i,j}))u_i = (r(\tau_{i,k+1}) - r(\tau_{i,1}))u_i =$

$(r(\tau_{i,k+1}) - \Phi_i)u_i$. By (3.7), we have $\mathsf{A}(\tau_i, 0, t, \mathcal{I}) = (t - \Phi_i)u_i$. Substituting $\mathsf{A}(\tau_i, 0, t, \mathcal{I})$ and $\mathsf{A}(\tau_i, 0, t, \mathcal{S})$ in (3.3), we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) = (t - \Phi_i)u_i - (r(\tau_{i,k+1}) - \Phi_i)u_i = (t - r(\tau_{i,k+1}))u_i$. □

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(b), τ_2 has no pending job at time 2 in $\mathcal{S}$ and $r(\tau_{2,1}) \leq 2 < r(\tau_{2,2})$ holds. Since $r(\tau_{2,2}) = 3$, for $t = 2$, we have $(t - r(\tau_{2,2}))u_2 = (2 - 3)\frac{2}{3} = -2/3$, which is also the value of $\mathsf{lag}(\tau_2, 2, \mathcal{S})$ (see Figure 3.1(c)).

Lemma 3.5. *If $\tau_{i,k}$ completes execution at or before $t \geq \Phi_i$ in $\mathcal{S}$, then $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (t - r(\tau_{i,k+1}))u_i$.*

Proof. Since $\tau_{i,k}$ completes execution at or before t, all jobs of τ_i released at or before $r(\tau_{i,k})$ complete execution at or before t. Hence, $\mathsf{A}(\tau_i, 0, t, \mathcal{S}) \geq \sum_{j=1}^{k} C_i = \sum_{j=1}^{k} T_i u_i = \sum_{j=1}^{k} (r(\tau_{i,j+1}) - r(\tau_{i,j}))u_i = (r(\tau_{i,k+1}) - r(\tau_{i,1}))u_i$. By (3.7), we have $\mathsf{A}(\tau_i, 0, t, \mathcal{I}) = (t - \Phi_i)u_i$. Substituting $\mathsf{A}(\tau_i, 0, t, \mathcal{I})$ and $\mathsf{A}(\tau_i, 0, t, \mathcal{S})$ in (3.3), we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{A}(\tau_i, 0, t, \mathcal{I}) - \mathsf{A}(\tau_i, 0, t, \mathcal{S}) \leq (t - \Phi_i)u_i - (r(\tau_{i,k+1}) - \Phi_i)u_i = (t - r(\tau_{i,k+1}))u_i$. □

Lemma 3.6. *If τ_i has a pending job $\tau_{i,k}$ at time $t \geq \Phi_i$ in $\mathcal{S}$, then $\mathsf{lag}(\tau_i, t, \mathcal{S}) > (t - r(\tau_{i,k+1}))u_i$.*

Proof. Since $\tau_{i,k}$ is pending at time t, we have $\mathsf{A}(\tau_i, 0, t, \mathcal{S}) < \sum_{j=1}^{k} C_i = \sum_{j=1}^{k} T_i u_i = \sum_{j=1}^{k} (r(\tau_{i,j+1}) - r(\tau_{i,j}))u_i = (r(\tau_{i,k+1}) - r(\tau_{i,1}))u_i$. By (3.7), $\mathsf{A}(\tau_i, 0, t, \mathcal{I}) = (t - \Phi_i)u_i$ holds. Substituting $\mathsf{A}(\tau_i, 0, t, \mathcal{I})$ and $\mathsf{A}(\tau_i, 0, t, \mathcal{S})$ in (3.3), we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{A}(\tau_i, 0, t, \mathcal{I}) - \mathsf{A}(\tau_i, 0, t, \mathcal{S}) > (t - \Phi_i)u_i - (r(\tau_{i,k+1}) - \Phi_i)u_i = (t - r(\tau_{i,k+1}))u_i$. □

Lemma 3.7. *If $\tau_{i,k}$ is the ready job of τ_i at time $t \geq \Phi_i$ in $\mathcal{S}$, then $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (t - r(\tau_{i,k}))u_i$.*

Proof. Since $\tau_{i,k}$ is the ready job of τ_i at time t, all jobs of τ_i prior to $\tau_{i,k}$ complete execution at or before t. Thus, $\mathsf{A}(\tau_i, 0, t, \mathcal{S}) \geq \sum_{j=1}^{k-1} C_i = \sum_{j=1}^{k-1} T_i u_i = \sum_{j=1}^{k-1} (r(\tau_{i,j+1}) - r(\tau_{i,j}))u_i = (r(\tau_{i,k}) - r(\tau_{i,1}))u_i = (r(\tau_{i,k}) - \Phi_i)u_i$. By (3.7), $\mathsf{A}(\tau_i, 0, t, \mathcal{I}) = (t - \Phi_i)u_i$ holds. Substituting $\mathsf{A}(\tau_i, 0, t, \mathcal{I})$ and $\mathsf{A}(\tau_i, 0, t, \mathcal{S})$ in (3.3), we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (t - \Phi_i)u_i - (r(\tau_{i,k}) - \Phi_i)u_i = (t - r(\tau_{i,k}))u_i$. □

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(b), $\tau_{3,1}$ is τ_3's ready job at at time 4. Task τ_3's lag at time 4 is $4 \cdot 2/3 - 1 = 5/3$. By Lemma 3.6, $\mathsf{lag}(\tau_3, 4, \mathcal{S}) = 5/3 > (4-6) \cdot 2/3 = -4/3$. By Lemma 3.7, $\mathsf{lag}(\tau_3, 4, \mathcal{S}) = 5/3 \leq (4 - 0) \cdot 2/3 = 8/3$.

Using Lemmas 3.4–3.7, we now prove Lemmas 3.8–3.11, which pertain to the relationship between the lag of a task τ_i at a pair of time instants that are separated by an integer multiple of τ_i's period. For any integer c and any pair of time instants $t, t + cT_i \geq \Phi_i$, jobs of τ_i that are separated by cT_i time units receive

the same allocation in $\mathcal{I}$ by time t and $t + cT_i$, respectively. For example, consider task τ_2 and times 4 and 10 in Figure 3.1(a). Since τ_2's period is 3.0, times 4 and 10 are separated by two periods of τ_2. Consider jobs $\tau_{2,2}$ and $\tau_{2,4}$ that are also separated by two periods of τ_2. In $\mathcal{I}$, both jobs $\tau_{2,2}$ and $\tau_{2,4}$ execute for one time unit at speed 2/3 by times 4 and 10, respectively.

Now consider the schedule $\mathcal{S}$. Although jobs separated by cT_i time units receive the same allocation by time t and $t + cT_i$ in $\mathcal{I}$, they may not in $\mathcal{S}$. The following lemma considers the case where τ_i has no pending job at time t in $\mathcal{S}$. In such a case, no job $\tau_{i,k+c}$ can receive more allocation by time $t + cT_i$ than $\tau_{i,k}$ receives by t. Since in $\mathcal{I}$, they receive the same allocation, the lag of τ_i at time t is not larger than lag of τ_i at time $t + cT_i$. The following lemma shows this.

Lemma 3.8. *For any time t and integer c such that $\min\{t, t + cT_i\} \geq \Phi_i$, if τ_i has no pending job at time t in $\mathcal{S}$, then* $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq \mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$.

Proof. Let $\tau_{i,k}$ be the job of τ_i such that $r(\tau_{i,k}) \leq t < r(\tau_{i,k+1})$ holds. Since τ_i has no pending job at time $t \geq \Phi_i$, by Lemma 3.4 we have

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) = (t - r(\tau_{i,k+1}))u_i. \tag{3.8}$$

Since the jobs of a task are released periodically and $t + cT_i \geq \Phi_i$ holds, we have $r(\tau_{i,k+c}) = r(\tau_{i,k}) + cT_i$. Since $r(\tau_{i,k}) \leq t < r(\tau_{i,k+1})$, we have $r(\tau_{i,k+c}) \leq t + cT_i < r(\tau_{i,k+c+1})$. By Lemmas 3.4 and 3.6, we have

$$\begin{aligned}
\mathsf{lag}(\tau_i, t + cT_i, \mathcal{S}) &\geq (t + cT_i - r(\tau_{i,k+c+1}))u_i \\
&= \{\text{Since } r(\tau_{i,k+c+1}) = r(\tau_{i,k+1}) + cT_i\} \\
&\quad (t + cT_i - r(\tau_{i,k+1}) - cT_i)u_i \\
&= (t - r(\tau_{i,k+1}))u_i \\
&= \{\text{By (3.8)}\} \\
&\quad \mathsf{lag}(\tau_i, t, \mathcal{S}).
\end{aligned}$$

□

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(b), τ_2 has no pending job at time 5 but has a pending job at time 8 in $\mathcal{S}$. By Lemma 3.8, $\mathsf{lag}(\tau_2, 5, \mathcal{S}) = -2/3 \leq 4/3 = \mathsf{lag}(\tau_2, 8, \mathcal{S})$.

The following lemma considers the case where $\mathsf{lag}(\tau_i, t, \mathcal{S})$ is not larger than $\mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$ and τ_i has a pending job $\tau_{i,k}$ at time t. Consider the job $\tau_{i,k+c}$ that has a separation time of cT_i with $\tau_{i,k}$. In $\mathcal{I}$, both jobs receive the same allocation by time t and $t + cT_i$, respectively. Therefore, $\tau_{i,k+c}$ must not receive more allocation by time $t + cT_i$ than $\tau_{i,k}$ has received by time t, meaning that $\tau_{i,k+c}$ is also pending.

Lemma 3.9. *For any integer c such that $\min\{t, t + cT_i\} \geq \Phi_i$, if $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq \mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$ holds and $\tau_{i,k}$ is the ready job of τ_i at t in $\mathcal{S}$, then $\tau_{i,k+c}$ is pending at time $t + cT_i$ in $\mathcal{S}$.*

Proof. Assume for a contradiction that $\tau_{i,k+c}$ is not pending at time $t + cT_i$. Since $\tau_{i,k}$ is pending at time $t \geq \Phi_i$, by Definition 2.4, $r(\tau_{i,k}) \leq t$ holds, by Lemma 3.6 we have

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) > (t - r(\tau_{i,k+1}))u_i. \tag{3.9}$$

Since jobs are released periodically and $r(\tau_{i,k}) \leq t$ holds, we have $r(\tau_{i,k+c}) \leq t+cT_i$. Thus, by Definition 2.4, $\tau_{i,k+c}$ finishes execution at or before $t + cT_i$ (as it is not pending then). By Lemma 3.5, we have

$$\begin{aligned}
\mathsf{lag}(\tau_i, t + cT_i, \mathcal{S}) &\leq (t + cT_i - r(\tau_{i,k+c+1}))u_i \\
&= \{\text{Since } \tau_i \text{ releases periodically,} r(\tau_{i,k+c+1}) = r(\tau_{i,k+1}) + cT_i\} \\
&\quad (t + cT_i - r(\tau_{i,k+1}) - cT_i)u_i \\
&= (t - r(\tau_{i,k+1}))u_i \\
&< \{\text{By } (3.9)\} \\
&\quad \mathsf{lag}(\tau_i, t, \mathcal{S}),
\end{aligned}$$

a contradiction. □

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(b), $\mathsf{lag}(\tau_2, 4, \mathcal{S}) = -1/3 \leq 2/3 = \mathsf{lag}(\tau_2, 7, \mathcal{S})$. By Lemma 3.9, since $\tau_{2,2}$ is the ready job of τ_2 at time 4 in $\mathcal{S}$ and time 7 corresponds to $c = 1$, $\tau_{2,3}$ is pending at time 7 in $\mathcal{S}$.

Similarly, we consider the case where $\mathsf{lag}(\tau_i, t, \mathcal{S})$ is either not smaller or strictly larger than $\mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$.

Lemma 3.10. *For any integer c such that $\min\{t, t + cT_i\} \geq \Phi_i$, if $\tau_{i,k}$ is the ready job of τ_i at time t in $\mathcal{S}$, then the following hold.*

(a) If $\mathsf{lag}(\tau_i, t, \mathcal{S}) \geq \mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$*, then all jobs of* τ_i *released before* $r(\tau_{i,k+c})$ *complete execution at or before* $t + cT_i$ *in* $\mathcal{S}$*.*
(b) If $\mathsf{lag}(\tau_i, t, \mathcal{S}) > \mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$*, then all jobs of* τ_i *released before* $r(\tau_{i,k+c})$ *complete execution before* $t + cT_i$ *in* $\mathcal{S}$*.*

Proof. If $k + c = 1$, then $r(\tau_{i,k+c}) = r(\tau_{i,1}) = \Phi_i$ and the lemma trivially holds. So assume $k + c > 1$. Since $\tau_{i,k}$ is the ready job at time $t \geq \Phi_i$, by Lemma 3.7,

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (t - r(\tau_{i,k}))u_i. \tag{3.10}$$

(a) Assume for a contradiction that τ_i has a job that is released before time $r(\tau_{i,k+c})$ but does not complete execution at or before time $t + cT_i$. Therefore, $\tau_{i,k+c-1}$ does not complete execution at or before time $t + cT_i$ as the jobs of each task are sequential. Since $\tau_{i,k}$ is the ready job of τ_i at time t, we have $r(\tau_{i,k}) \leq t$. Since jobs are released periodically, we have $r(\tau_{i,k+c}) \leq t + cT_i$, which implies $r(\tau_{i,k+c-1}) \leq t + cT_i$. Therefore, by Definition 2.4, $\tau_{i,k+c-1}$ is pending at time $t + cT_i$. Thus, by Lemma 3.6,

$$\begin{aligned}
\mathsf{lag}(\tau_i, t + cT_i, \mathcal{S}) &> (t + cT_i - r(\tau_{i,k+c}))u_i \\
&= \{\text{Since } \tau_i \text{ releases periodically} r(\tau_{i,k+c}) = r(\tau_{i,k}) + cT_i\} \\
&\quad (t + cT_i - r(\tau_{i,k}) - cT_i)u_i \\
&= (t - r(\tau_{i,k}))u_i \\
&\geq \{\text{By (3.10)}\} \\
&\quad \mathsf{lag}(\tau_i, t, \mathcal{S}),
\end{aligned}$$

a contradiction.

(b) Since $\mathsf{lag}(\tau_i, t, \mathcal{S}) > \mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$, by Lemma 3.10(a), all jobs of τ_i released before time $r(\tau_{i,k+c})$ finish execution at or before time $t+cT_i$. Assume that they do not complete execution before time $t+cT_i$. Thus, they complete execution at time $t + cT_i$, and no job released at or after time $r(\tau_{i,k+c})$ executes at or before time $t + cT_i$. Thus, $\mathsf{A}(\tau_i, 0, t + cT_i, \mathcal{S}) = \sum_{j=1}^{k+c-1} C_i = \sum_{j=1}^{k+c-1} T_i u_i = \sum_{j=1}^{k+c-1} (r(\tau_{i,j+1}) - r(\tau_{i,j}))u_i = (r(\tau_{i,k+c}) - \Phi_i)u_i = (r(\tau_{i,k}) + cT_i - \Phi_i)u_i$. Thus, by (3.7) and (3.3), $\mathsf{lag}(\tau_i, t + cT_i, \mathcal{S}) = \mathsf{A}(\tau_i, 0, t +$

$cT_i, \mathcal{I}) - \mathsf{A}(\tau_i, 0, t + cT_i, \mathcal{S}) = (t + cT_i - \Phi_i)u_i - (r(\tau_{i,k}) + cT_i - \Phi_i)u_i = (t - r(\tau_{i,k}))u_i \geq \mathsf{lag}(\tau_i, t, \mathcal{S})$, a contradiction. □

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(b), $\mathsf{lag}(\tau_2, 8, \mathcal{S}) = 4/3 > 1/3 = \mathsf{lag}(\tau_2, 11, \mathcal{S})$. By Lemma 3.10(b), since $\tau_{2,3}$ is the ready job of τ_2 at time 8 in $\mathcal{S}$ and time 11 corresponds to $c = 1$, all jobs of τ_2 prior to $\tau_{2,4}$ complete execution before time 11 in $\mathcal{S}$.

We now utilize Lemmas 3.9 and 3.10(a) to establish a necessary condition for $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$ to hold. Intuitively, if $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, t + cT_i, \mathcal{S})$ holds, then in $\mathcal{S}$ any job $\tau_{i,k}$'s allocation at or before time t must equal the allocation of job $\tau_{i,k+c}$ at or before time $t + cT_i$. This is because their allocation in $\mathcal{I}$ are also the same at these time instants.

Lemma 3.11. *For any time t and integer c such that $\min\{t, t+cT_i\} \geq \Phi_i$, if $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, t+cT_i, \mathcal{S})$, then the following hold.*

(a) If there is no pending job of τ_i at time t in $\mathcal{S}$, then there is no pending job of τ_i at time $t + cT_i$ in $\mathcal{S}$.

(b) If $\tau_{i,k}$ is the ready job of τ_i at time t in $\mathcal{S}$, then $\tau_{i,k+c}$ is the ready job of τ_i at time $t + cT_i$ in $\mathcal{S}$.

Proof. **(a)** Assume that there is a pending job of τ_i at time $t + cT_i$ and let $\tau_{i,k}$ be the ready job of τ_i at time $t + cT_i$. Substituting t and c in Lemma 3.9 by $t + cT_i$ and $-c$, respectively, job $\tau_{i,k-c}$ is pending at time t, a contradiction.

(b) By Lemma 3.9, $\tau_{i,k+c}$ is pending at $t + cT_i$. By Lemma 3.10(a), all jobs of τ_i released before $r(\tau_{i,k+c})$ finish execution at or before time $t + cT_i$. Thus, $\tau_{i,k+c}$ is the ready job of τ_i at time $t + cT_i$. □

We now give a necessary condition for the lag-monotonicity property to not hold.

Lemma 3.12. *Let $t \geq \Phi_i + H$ be the first time instant (if one exists) such that $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) > \mathsf{lag}(\tau_i, t, \mathcal{S})$ holds in $\mathcal{S}$. Then, the following hold.*

(a) $t > \Phi_i + H$.

(b) τ_i executes during $[t - 1, t)$, but does not execute during $[t - H - 1, t - H)$ in $\mathcal{S}$.

Proof. **(a)** Assume that $t = \Phi_i + H$. Since $t - H = \Phi_i$, we have $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) = \mathsf{lag}(\tau_i, \Phi_i, \mathcal{S}) = 0$. Since T_i divides H, by Lemma 3.3, we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, \Phi_i + H, \mathcal{S}) \geq 0$. Therefore, $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) \leq \mathsf{lag}(\tau_i, t, \mathcal{S})$, a contradiction.

(b) By Lemma 3.12(a), $t - 1 \geq \Phi_i + H$ holds. By the definition of t, we have

$$\mathsf{lag}(\tau_i, t - H - 1, \mathcal{S}) \leq \mathsf{lag}(\tau_i, t - 1, \mathcal{S}). \tag{3.11}$$

Assume that τ_i does not execute during $[t-1, t)$ or does execute during $[t-H-1, t-H)$. Then, one of the following three cases holds.

Case 1. *τ_i executes during both $[t - H - 1, t - H)$ and $[t - 1, t)$.* By Lemma 3.1(a),

$$\mathsf{lag}(\tau_i, t - H, \mathcal{S}) = \mathsf{lag}(\tau_i, t - H - 1, \mathcal{S}) + u_i - 1, \tag{3.12}$$

and

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, t - 1, \mathcal{S}) + u_i - 1. \tag{3.13}$$

Since $\mathsf{lag}(\tau_i, t-H, \mathcal{S}) > \mathsf{lag}(\tau_i, t, \mathcal{S})$, by (3.12) and (3.13), we have $\mathsf{lag}(\tau_i, t-H-1, \mathcal{S}) > \mathsf{lag}(\tau_i, t-1, \mathcal{S})$, which contradicts (3.11).

Case 2. *τ_i does not execute during both $[t - H - 1, t - H)$ and $[t - 1, t)$.* By Lemma 3.1(b),

$$\mathsf{lag}(\tau_i, t - H, \mathcal{S}) = \mathsf{lag}(\tau_i, t - H - 1, \mathcal{S}) + u_i, \tag{3.14}$$

and

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, t - 1, \mathcal{S}) + u_i. \tag{3.15}$$

Since $\mathsf{lag}(\tau_i, t-H, \mathcal{S}) > \mathsf{lag}(\tau_i, t, \mathcal{S})$, by (3.14) and (3.15), we have $\mathsf{lag}(\tau_i, t-H-1, \mathcal{S}) > \mathsf{lag}(\tau_i, t-1, \mathcal{S})$, which contradicts (3.11).

Case 3. *τ_i executes during $[t - H - 1, t - H)$ but does not execute during $[t - 1, t)$.* Thus, (3.12) and (3.15) hold. Therefore, by (3.12), we have

$$\begin{aligned}
\mathsf{lag}(\tau_i, t - H - 1, \mathcal{S}) &= \mathsf{lag}(\tau_i, t - H, \mathcal{S}) + 1 - u_i \\
&\geq \{\text{Since } u_i \leq 1\} \\
&\quad \mathsf{lag}(\tau_i, t - H, \mathcal{S}) \\
&> \{\text{By the definition of } t\}
\end{aligned}$$

$$
\begin{aligned}
& \mathsf{lag}(\tau_i, t, \mathcal{S}) \\
\geq\ & \{\text{By (3.15) and } u_i \geq 0\} \\
& \mathsf{lag}(\tau_i, t-1, \mathcal{S}),
\end{aligned}
$$

a contradiction to (3.11). □

Definition 3.2. Let $h_i = H/T_i$. ◄

The following lemma gives a lag-monotonicity property for SRT-schedulable systems that is similar to that given previously for HRT-schedulable systems on uniprocessors under EDF scheduler [Leung and Merrill, 1980] and HRT-schedulable systems on multiprocessors under G-EDF scheduler [Cucu-Grosjean and Goossens, 2011]. Informally, we show that, using Lemmas 3.8–3.10 and 3.12, no task can receive more allocation in $\mathcal{S}$ than $\mathcal{I}$ over an interval $[t-H, t)$ because of the existence of higher-priority jobs of other tasks, *i.e.,* over-allocating a task would require under-allocating another task, violating the priority ordering of the jobs.

Lemma 3.13 (lag-monotonicity)**.** *For any task τ_i and any time $t \geq \Phi_i + H$,* $\mathsf{lag}(\tau_i, t-H, \mathcal{S}) \leq \mathsf{lag}(\tau_i, t, \mathcal{S})$.

Proof. We use Figure 3.2 to illustrate the proof. Assume for a contradiction that t is the first time instant such that $t \geq \Phi_i + H$ and there is a task τ_i with $\mathsf{lag}(\tau_i, t-H, \mathcal{S}) > \mathsf{lag}(\tau_i, t, \mathcal{S})$. By Lemma 3.12(b), τ_i executes during $[t-1, t)$. Let $\tau_{i,p}$ be the job of τ_i that executes during $[t-1, t)$. Since T_i divides H, by Lemma 3.8 (with t and c replaced by $t-H$ and h_i, respectively), there is a pending job of τ_i at time $t-H$. Let $\tau_{i,k}$ be the ready job of τ_i at time $t-H$.

Claim 3.1. $r(\tau_{i,k}) < t-H$.

Proof. Assume otherwise. Then, $r(\tau_{i,k}) = t-H$ and $\mathsf{lag}(\tau_i, t-H, \mathcal{S}) = 0$ hold. Since jobs are released periodically and $t = r(\tau_{i,k}) + H$ holds, there is a non-negative integer c such that $t = \Phi_i + cT_i$, which by Lemma 3.3 implies $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, \Phi_i + cT_i, \mathcal{S}) \geq 0$. Therefore, $\mathsf{lag}(\tau_i, t-H, \mathcal{S}) = 0 \leq \mathsf{lag}(\tau_i, t, \mathcal{S})$, and t cannot be a time instant with $\mathsf{lag}(\tau_i, t-H, \mathcal{S}) > \mathsf{lag}(\tau_i, t, \mathcal{S})$. Therefore, $r(\tau_{i,k}) < t-H$ holds. □

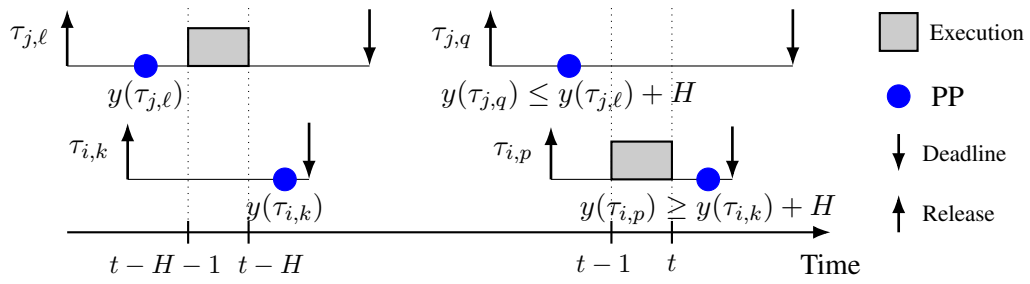


Figure 3.2: Illustration of the proof of Lemma 3.13.

By Claim 3.1, $\tau_{i,k}$ is pending at time $t - H - 1$. By Lemma 3.12(b), $\tau_{i,k}$ does not execute during $[t - H - 1, t - H)$ (see Figure 3.2). Since $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) > \mathsf{lag}(\tau_i, t, \mathcal{S})$, substituting t and c in Lemma 3.10(b) by $t - H$ and h_i (Definition 3.2), respectively, all jobs of τ_i released before $r(\tau_{i,k+h_i})$ complete execution before time t (at or before time $t - 1$). Thus, $p \ge k + h_i$ and we have

$$\begin{aligned} y(\tau_{i,p}) &\ge y(\tau_{i,k+h_i}) \\ &= \{\text{Since } \tau_i \text{ releases periodically, } y(\tau_{i,k+h_i}) = y(\tau_{i,k}) + h_i T_i \text{ holds and by Definition 3.2}\} \\ &\quad y(\tau_{i,k}) + H. \end{aligned} \tag{3.16}$$

Since $\tau_{i,k}$ is pending but does not execute during $[t - H - 1, t - H)$ and $\tau_{i,p}$ executes during $[t - 1, t)$, there must be a task τ_j that executes during $[t - H - 1, t - H)$, but does not execute during $[t - 1, t)$. Let $\tau_{j,\ell}$ executes during $[t - H - 1, t - H)$ (see Figure 3.2). By Lemma 3.12(a), $t > \Phi_i + H$, and hence, $t - 1 \ge \Phi_i + H$. Thus, by the definition of t, $\mathsf{lag}(\tau_j, t - H - 1, \mathcal{S}) \le \mathsf{lag}(\tau_j, t - 1, \mathcal{S})$ holds. Since $\tau_{j,\ell}$ executes during $[t - H - 1, t - H)$, it is the ready job of τ_j at time $t - H - 1$. Substituting $\tau_{i,k}, t$, and c in Lemma 3.9 by $\tau_{j,\ell}, t - H - 1$, and h_j, respectively, $\tau_{j,\ell+h_j}$ is pending at time $t - 1$. Therefore, τ_j has a pending job at time $t - 1$; let $\tau_{j,q}$ be the ready job of τ_j at time $t - 1$. Thus, we have

$$\begin{aligned} y(\tau_{j,q}) &\le y(\tau_{j,\ell+h_j}) \\ &= \{\text{Since } \tau_j \text{ releases periodically, } y(\tau_{j,\ell+h_j}) = y(\tau_{j,\ell}) + h_j T_j \text{ holds and by Definition 3.2}\} \\ &\quad y(\tau_{j,\ell}) + H. \end{aligned} \tag{3.17}$$

Since $\tau_{i,k}$ is the ready job of τ_i at time $t - H - 1$ but does not execute during $[t - H - 1, t - H)$, and $\tau_{j,\ell}$ executes during $[t - H - 1, t - H)$, by Rule PR, we have two cases.

Case 1. $y(\tau_{j,\ell}) < y(\tau_{i,k})$. Substituting $y(\tau_{j,\ell})$ by $y(\tau_{i,k})$ in (3.17), we have $y(\tau_{j,q}) < y(\tau_{i,k}) + H$. By (3.16), $y(\tau_{j,q}) < y(\tau_{i,k}) + H \leq y(\tau_{i,p})$. Therefore, by Rule PR, $\tau_{j,q}$ has higher priority than $\tau_{i,p}$. Hence, $\tau_{i,p}$ cannot execute during $[t-1, t)$, while $\tau_{j,\ell}$ is not executing during $[t-1, t)$, a contradiction.

Case 2. $y(\tau_{j,\ell}) = y(\tau_{i,k})$ and $j < i$. Substituting $y(\tau_{j,\ell})$ by $y(\tau_{i,k})$ in (3.17), we have $y(\tau_{j,q}) \leq y(\tau_{i,k}) + H$, which by (3.16) implies $y(\tau_{j,q}) \leq y(\tau_{i,k}) + H \leq y(\tau_{i,p})$. Therefore, by Rule PR, $\tau_{j,q}$ has higher priority than $\tau_{i,p}$. Hence, $\tau_{i,p}$ cannot execute during $[t-1, t)$, while $\tau_{j,\ell}$ is not executing during $[t-1, t)$, a contradiction.

In both cases, we reach a contradiction. □

By (3.5) and Lemma 3.13, we have the following LAG-monotonicity property.

Corollary 3.1 (LAG-monotonicity)**.** *For any time instant $t \geq \Phi_{max} + H$,* $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) \leq \mathsf{LAG}(\Gamma, t, \mathcal{S})$.

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(a), we have $H = 6$, $\mathsf{lag}(\tau_2, 4, \mathcal{S}) = -1/3 \leq 2/3 = \mathsf{lag}(\tau_2, 10, \mathcal{S})$ and $\mathsf{LAG}(\Gamma, 4, \mathcal{S}) = 1 \leq 2 = \mathsf{LAG}(\Gamma, 10, \mathcal{S})$.

Lemma 3.14. *If $\tau_{i,k}$ is the ready job of τ_i at t in $\mathcal{S}$, then $y(\tau_{i,k}) \leq t - \frac{\mathsf{lag}(\tau_i, t, \mathcal{S})}{u_i} + Y_i$.*

Proof. By Lemma 3.7, we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (t - r(\tau_{i,k}))u_i$. Applying (3.1) (*i.e.,* $y(\tau_{i,k}) = r(\tau_{i,k}) + Y_i$), we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (t - y(\tau_{i,k}) + Y_i)u_i$. The proof follows from rearranging this inequality. □

The following lemma provides a relationship between lag and response times. The proof of this lemma does not depend on the used scheduling algorithm.[3]

Lemma 3.15 ([Yang and Anderson, 2017], Corollary 1)**.** *If $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq X_i$ holds for any t, then the response time of τ_i is at most $T_i + \frac{X_i}{u_i}$.*

3.2.2 Deriving Response-Time Bounds

We now derive response-time bounds for periodic tasks using the properties of lag derived in Section 3.2.1. We derive our bound by first deriving an upper bound on the lag (Lemma 3.20) of any task τ_i, and then applying Lemma 3.15 on the derived upper bound. To derive an upper bound on per-task lag, we first give Lemmas 3.16–3.19.

[3]The lemma statement in [Yang and Anderson, 2017] refers to tardiness instead of response times.

Definition 3.3. In $\mathcal{S}$, a time instant t is called *busy* if at least $\lceil U_{tot} \rceil$ tasks have pending jobs at time t, and *non-busy* otherwise. In $\mathcal{S}$, a time interval $[t, t')$ is called *busy* (resp., *non-busy*) if each instant in the interval is busy (resp., non-busy). ◀

Lemma 3.16. *If τ_i continuously executes during $[t, t')$ in $\mathcal{S}$, then* $\mathsf{lag}(\tau_i, t', \mathcal{S}) \leq \mathsf{lag}(\tau_i, t, \mathcal{S})$.

Proof. Follows from Lemma 3.1(a) and $u_i \leq 1$. □

Lemma 3.17. *If $[t, t')$ is a busy interval in $\mathcal{S}$, then* $\mathsf{LAG}(\Gamma, t', \mathcal{S}) \leq \mathsf{LAG}(\Gamma, t, \mathcal{S})$.

Proof. By the definition of $\mathcal{I}$, $\mathsf{A}(\Gamma, t, t', \mathcal{I}) \leq U_{tot}(t' - t)$ holds. By Definition 3.3, we have $\mathsf{A}(\Gamma, t, t', \mathcal{S}) \geq \lceil U_{tot} \rceil (t' - t)$. Therefore, by (3.6) and $U_{tot} \leq \lceil U_{tot} \rceil$, $\mathsf{LAG}(\Gamma, t', \mathcal{S}) = \mathsf{LAG}(\Gamma, t, \mathcal{S}) + \mathsf{A}(\Gamma, t, t', \mathcal{I}) - \mathsf{A}(\Gamma, t, t', \mathcal{S}) \leq \mathsf{LAG}(\Gamma, t, \mathcal{S}) + U_{tot}(t' - t) - \lceil U_{tot} \rceil (t' - t) \leq \mathsf{LAG}(\Gamma, t, \mathcal{S})$. □

Lemma 3.18. *For any $t \geq \Phi_{max} + H$, if* $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t, \mathcal{S})$ *holds, then for each τ_i,* $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S})$ *holds.*

Proof. Assume that there is a task τ_i with $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) \neq \mathsf{lag}(\tau_i, t, \mathcal{S})$. Since $t \geq \Phi_{max} + H$, by Lemma 3.13, $\mathsf{lag}(\tau_j, t - H, \mathcal{S}) \leq \mathsf{lag}(\tau_j, t, \mathcal{S})$ holds for any task τ_j including τ_i. Therefore, $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) < \mathsf{lag}(\tau_i, t, \mathcal{S})$ holds. By (3.5), we have

$$\begin{aligned}
\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) &= \sum_{\tau_j \in \Gamma \setminus \{\tau_i\}} \mathsf{lag}(\tau_j, t - H, \mathcal{S}) + \mathsf{lag}(\tau_i, t - H, \mathcal{S}) \\
&< \{\text{Since } \mathsf{lag}(\tau_i, t - H, \mathcal{S}) < \mathsf{lag}(\tau_i, t, \mathcal{S}) \text{ and for all } j,\ \mathsf{lag}(\tau_j, t - H, \mathcal{S}) \leq \mathsf{lag}(\tau_j, t, \mathcal{S})\} \\
&\quad \sum_{\tau_j \in \Gamma \setminus \{\tau_i\}} \mathsf{lag}(\tau_j, t, \mathcal{S}) + \mathsf{lag}(\tau_i, t, \mathcal{S}) \\
&= \mathsf{LAG}(\Gamma, t, \mathcal{S}),
\end{aligned}$$

a contradiction. □

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(b), $\mathsf{LAG}(\Gamma, 7, \mathcal{S}) = 2 = \mathsf{LAG}(\Gamma, 13, \mathcal{S})$ holds. By Lemma 3.18, we have $\mathsf{lag}(\tau_1, 7, \mathcal{S}) = \mathsf{lag}(\tau_1, 13, \mathcal{S}) = -1/3$, $\mathsf{lag}(\tau_2, 7, \mathcal{S}) = \mathsf{lag}(\tau_2, 13, \mathcal{S}) = 2/3$, and $\mathsf{lag}(\tau_3, 7, \mathcal{S}) = \mathsf{lag}(\tau_3, 13, \mathcal{S}) = 5/3$.

Lemma 3.19. *For any $X_i > 0$, if t is the first time instant such that* $\mathsf{lag}(\tau_i, t, \mathcal{S}) > X_i$, *then τ_i does not execute during $[t - 1, t)$.*

Proof. Since $X_i > 0$ and for any $t' \leq \Phi_i$, $\mathsf{lag}(\tau_i, t', \mathcal{S}) = 0$ holds, we have $t > \Phi_i$. Therefore, $\mathsf{lag}(\tau_i, t-1, \mathcal{S}) \leq X_i$ holds. Assume that τ_i executes during $[t-1, t)$. By Lemma 3.16, $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq \mathsf{lag}(\tau_i, t-1, \mathcal{S}) \leq X_i$, a contradiction. □

We now show that each task τ_i's lag cannot exceed $(H + Y_i - Y_{min})u_i$. Informally, assume that t is the first time instant where a task τ_i's lag exceeds $(H + Y_i - Y_{min})u_i$ in $\mathcal{S}$. If $[t - H, t)$ is a busy-interval, then by Lemma 3.17 (LAG does not increase over a busy interval) and Corollary 3.1 (LAG-monotonicity), LAG at $t - H$ and t must be the same in $\mathcal{S}$, which by Lemma 3.18 implies τ_i's lag at $t - H$ and t is also same. Otherwise, if there is a non-busy instant t_b in $[t - H, t)$, then by Lemma 3.14, τ_i's ready job's priority must be higher than any job released at or after t_b throughout $[t_b, t)$. Therefore, τ_i would execute continuously throughout $[t_b, t)$, violating Lemma 3.19. We now give the formal proof.

Lemma 3.20. *For any task τ_i and any time instant t in $\mathcal{S}$, $\mathsf{lag}(\tau_i, t, \mathcal{S}) \leq (H + Y_i - Y_{min})u_i$.*

Proof. We use Figure 3.3 to illustrate the proof. Assume that there is a time instant t such that there is a task τ_i with $\mathsf{lag}(\tau_i, t, \mathcal{S}) > (H + Y_i - Y_{min})u_i$ and let t be the first such time instant. Since $\mathcal{I}$ executes τ_i at the rate of u_i, $\mathsf{lag}(\tau_i, t', \mathcal{S}) \leq Hu_i$ holds for any $t' \leq \Phi_i + H$. Therefore, $t > \Phi_i + H \geq H$ holds.

We first prove that $[t - H, t)$ is a busy interval. Since $t > H$, $[t - H, t)$ is a valid time interval. By Lemma 3.2, there is a pending job of τ_i at time t because $\mathsf{lag}(\tau_i, t, \mathcal{S}) > 0$. Let $\tau_{i,k}$ be the ready job of τ_i at time t. By Lemma 3.14, we have

$$\begin{aligned} y(\tau_{i,k}) &\leq t - \frac{\mathsf{lag}(\tau_i, t, \mathcal{S})}{u_i} + Y_i \\ &< \{\text{Since } \mathsf{lag}(\tau_i, t, \mathcal{S}) > (H + Y_i - Y_{min})u_i\} \\ &\quad t - \frac{(H + Y_i - Y_{min})u_i}{u_i} + Y_i \\ &= t - H + Y_{min}. \end{aligned} \tag{3.18}$$

By (3.1), we have

$$\begin{aligned} r(\tau_{i,k}) &= y(\tau_{i,k}) - Y_i \\ &< \{\text{By (3.18)}\} \\ &\quad t - H + Y_{min} - Y_i \end{aligned}$$

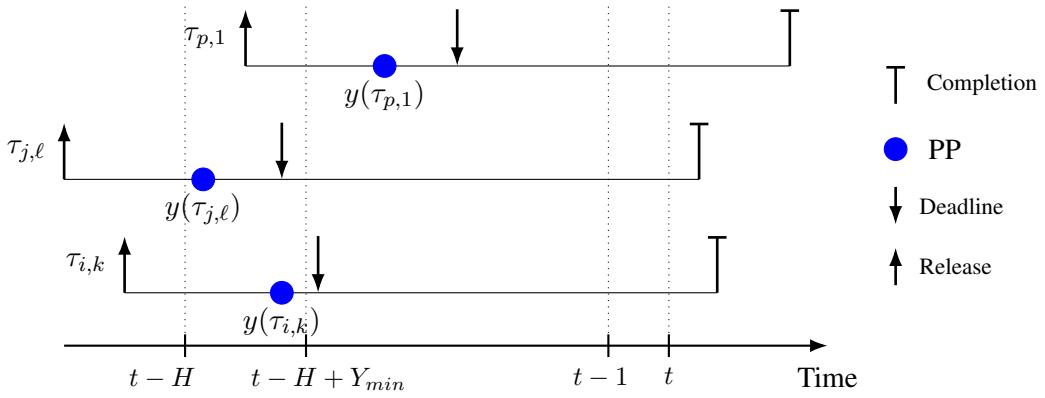


Figure 3.3: Illustration of the proof of Lemma 3.20.

$$\leq \{\text{Since } Y_{min} \leq Y_i\}$$
$$t - H. \tag{3.19}$$

Thus, by Definition 2.4, $\tau_{i,k}$ is pending throughout $[t-H, t)$. Since t is the first time instant with $\mathsf{lag}(\tau_i, t, \mathcal{S}) > (H + Y_i - Y_{min})u_i$, by Lemma 3.19, $\tau_{i,k}$ does not execute during $[t-1, t)$. Thus, there are at least M tasks with higher priority jobs than $\tau_{i,k}$ at time $t-1$. Let Γ^h be the set of tasks having higher priority jobs than $\tau_{i,k}$ at time $t-1$. Then, $|\Gamma^h| \geq M$ holds. By the definition of Γ^h, for any task $\tau_j \in \Gamma^h$, $y(\tau_{j,\ell}) \leq y(\tau_{i,k})$ holds where $\tau_{j,\ell}$ is the ready job of τ_j at time $t-1$ (see Figure 3.3). By a calculation similar to that yielding (3.19), $r(\tau_{j,\ell}) < t - H$ holds, which implies $\tau_{j,\ell}$ is pending throughout $[t-H, t)$. Thus, by (3.18) we have the following property.

Property 3.2. *Each task in $\Gamma^h \cup \{\tau_i\}$ has pending jobs with PPs less than $t - H + Y_{min}$ throughout $[t-H, t)$.*

By Property 3.2, $[t-H, t)$ is a busy interval. By Lemma 3.17, we therefore have

$$\mathsf{LAG}(\Gamma, t, \mathcal{S}) \leq \mathsf{LAG}(\Gamma, t - H, \mathcal{S}). \tag{3.20}$$

We now consider two cases.

Case 1. $t \geq \Phi_{max} + H$. By Corollary 3.1, we have

$$\mathsf{LAG}(\Gamma, t, \mathcal{S}) \geq \mathsf{LAG}(\Gamma, t - H, \mathcal{S}). \tag{3.21}$$

By (3.20) and (3.21), we have

$$\mathsf{LAG}(\Gamma, t, \mathcal{S}) = \mathsf{LAG}(\Gamma, t - H, \mathcal{S}). \tag{3.22}$$

Since $t \geq \Phi_{max} + H$ and (3.22) holds, by Lemma 3.18, $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, t - H, \mathcal{S})$ holds. Therefore, t cannot be the first time instant with $\mathsf{lag}(\tau_i, t, \mathcal{S}) > (H + Y_i - Y_{min})u_i$.

Case 2. $t < \Phi_{max} + H$. Let Γ^s be the set of tasks such that for each $\tau_p \in \Gamma^s$, $t - H < \Phi_p \leq \Phi_{max}$ holds. Since each task $\tau_p \in \Gamma^s$ releases its first job after $t - H$, $r(\tau_{p,1}) > t - H$ and $\mathsf{lag}(\tau_p, t - H, \mathcal{S}) = 0$ hold (see Figure 3.3). Thus, by (3.1) and $Y_p \geq Y_{min}$, we have

$$\forall \tau_p \in \Gamma^s : y(\tau_{p,1}) > t - H + Y_{min}. \tag{3.23}$$

By Property 3.2 and (3.23), no task $\tau_p \in \Gamma^s$ executes during $[t - H, t)$. Therefore, we have

$$\forall \tau_p \in \Gamma^s : \mathsf{lag}(\tau_p, t, \mathcal{S}) \geq 0 = \mathsf{lag}(\tau_p, t - H, \mathcal{S}). \tag{3.24}$$

By the definition of Γ^s, for any task $\tau_q \in \Gamma \setminus \Gamma^s$, $t - H \geq \Phi_q$ holds, which implies $t \geq \Phi_q + H$. Therefore, by Lemma 3.13, we have

$$\forall \tau_q \in \Gamma \setminus \Gamma^s : \mathsf{lag}(\tau_q, t, \mathcal{S}) \geq \mathsf{lag}(\tau_q, t - H, \mathcal{S}). \tag{3.25}$$

Since t is the first time instant with $\mathsf{lag}(\tau_i, t, \mathcal{S}) > (H + Y_i - Y_{min})u_i > 0$, $\mathsf{lag}(\tau_i, t', \mathcal{S}) \leq (H + Y_i - Y_{min})u_i$ holds for any $t' < t$. Thus, we have

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) > \mathsf{lag}(\tau_i, t - H, \mathcal{S}). \tag{3.26}$$

By (3.5), we have

$$\begin{aligned}
\mathsf{LAG}(\Gamma, t, \mathcal{S}) &= \sum_{\tau_j \in \Gamma} \mathsf{lag}(\tau_j, t, \mathcal{S}) \\
&= \sum_{\tau_j \in \Gamma^s} \mathsf{lag}(\tau_j, t, \mathcal{S}) + \sum_{\tau_j \in \Gamma \setminus (\Gamma^s \cup \{\tau_i\})} \mathsf{lag}(\tau_j, t, \mathcal{S}) + \mathsf{lag}(\tau_i, t, \mathcal{S}) \\
&> \{\text{By (3.24), (3.25), and (3.26)}\}
\end{aligned}$$

$$\sum_{\tau_j \in \Gamma^s} \mathsf{lag}(\tau_j, t-H, \mathcal{S}) + \sum_{\tau_j \in \Gamma \setminus (\Gamma^s \cup \{\tau_i\})} \mathsf{lag}(\tau_j, t-H, \mathcal{S}) + \mathsf{lag}(\tau_i, t-H, \mathcal{S})$$
$$= \mathsf{LAG}(\Gamma, t-H, \mathcal{S}),$$

a contradiction to (3.20). □

We now give our tardiness bound in the following Theorem.

Theorem 3.1. *The response time of task τ_i is at most $T_i + H + Y_i - Y_{min}$ in $\mathcal{S}$.*

Proof. The theorem follows from Lemmas 3.15 and 3.20. □

Therefore, for pseudo-harmonic periodic systems, we have the following theorem.

Theorem 3.2. *For a pseudo-harmonic task system Γ, the response time of a task τ_i is at most $T_i + T_{max} + Y_i - Y_{min}$ in $\mathcal{S}$.*

By Theorem 3.2, we have following response-time bounds under G-EDF and G-FIFO. Note that G-EDF scheduling assumes $D_i = T_i$.

Theorem 3.3. *For a pseudo-harmonic task system Γ, the response time of a task τ_i in a G-EDF and G-FIFO schedule is at most $T_i + T_{max} + T_i - T_{min}$ and $T_i + T_{max}$, respectively.*

Removing Assumption 3.1. Prior work has shown that removing Assumption 3.1 does not invalidate G-EDF response-time bounds because its removal cannot cause work to shift later [Yang and Anderson, 2017], *i.e.,* G-EDF is sustainable. It can be similarly removed for any GEL scheduler.

Theorem 3.4. *Let Γ be a periodic task set, $\mathcal{S}$ be a GEL schedule of Γ satisfying Assumption 3.1, and $\mathcal{S}'$ be a GEL schedule with the same PP for each job of Γ without satisfying Assumption 3.1. Then, no job in $\mathcal{S}'$ finishes later than in $\mathcal{S}$.*

Tightness. The following example shows the tightness of the tardiness bound in Theorem 3.2.

Example 3.2. Consider a task system Γ with $M+1$ tasks where $\tau_i = (0, M, M+1)$. For any JLFP scheduler, the maximum response time among all tasks is $T_i + M - 1 = T_i + T_{max} - 2$. For both G-FIFO and G-EDF, the response-time bound of a task in Γ by Theorem 3.1 is $T_i + T_{max}$. A G-EDF/G-FIFO schedule corresponding to $M = 5$ is shown in Figure 3.4. Jobs $\tau_{6,1}, \tau_{5,2}$, and $\tau_{4,3}$ have response times of 10.0, 9.0, and 8.0 time units, respectively. Similarly, $\tau_{3,4}$ has response time of 7.0 time units (not shown in Figure 3.4). τ_1 and τ_2 have no tardy job. The schedule repeats after 30.0 time units. ◀

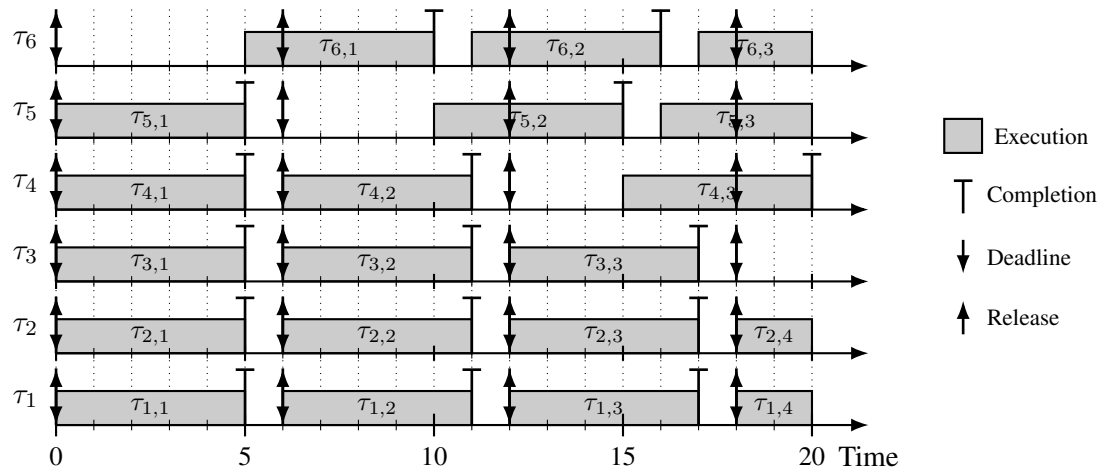


Figure 3.4: Schedule corresponding to Example 3.2

.

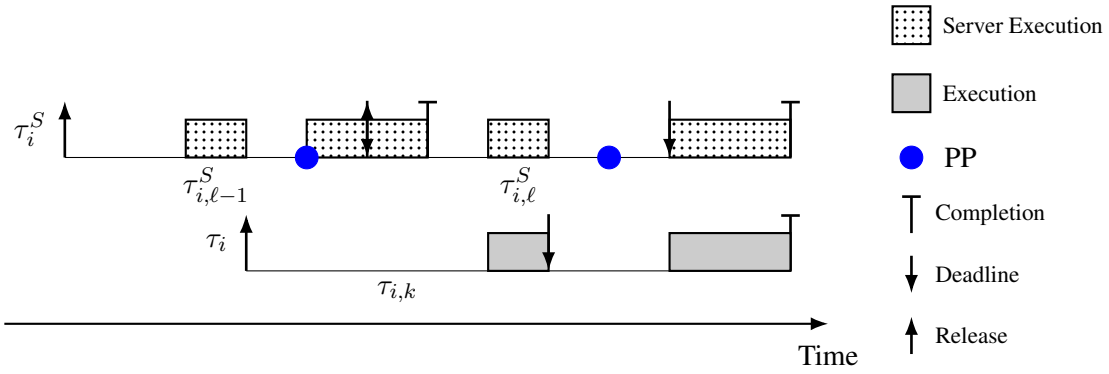


Figure 3.5: Scheduling sporadic tasks by GEL-scheduled periodic servers.

Sporadic tasks. We can enable similar response-time bounds for sporadic tasks using GEL-scheduled periodic servers. For each task τ_i, we create a server task $\tau_i^s = (0, C_i, T_i)$. We schedule the server tasks by a GEL scheduler where each server job of τ_i^s receives an allocation of exactly C_i time units. We schedule job $\tau_{i,k}$ on server job $\tau_{i,\ell}^s$ where $d(\tau_{i,k}) \in (r(\tau_{i,\ell})^s, d(\tau_{i,\ell})^s]$ (see Figure 3.5). Since both τ_i and τ_i^s have the same period, no other job of τ_i is scheduled on $\tau_{i,\ell}^s$. Since $\tau_{i,\ell}^s$ receives allocation of C_i time units, $\tau_{i,k}$ finishes execution at or before $\tau_{i,\ell}^s$ completes. Since $d(\tau_{i,\ell})^s - d(\tau_{i,k}) \leq T_i$, we have the following theorem.

Theorem 3.5. *A pseudo-harmonic sporadic task system Γ can be scheduled using periodic servers scheduled by a GEL scheduler such that each task τ_i's response time is at most $2T_i + T_{max} + Y_i - Y_{min}$.*

Discussion. While the response-time bound given in Theorem 3.1 is tight in general, the response-time bound is not tight for task systems with a smaller total utilization than M. For instance, an HRT-schedulable task system also has the response-time bound specified in Theorem 3.1. Although the response-time bounds

in [Devi and Anderson, 2008; Erickson et al., 2014; Valente, 2016] can have smaller bounds when the total utilization is less than M compared to systems with full utilization, they also have similar issues, *e.g.,* response-time bounds larger than relative deadlines for HRT-schedulable task systems.

Although the response-time bound given in Theorem 3.2 is $T_i + T_{max}$ under G-FIFO, the response-time bound under G-EDF can be larger than $T_i + T_{max}$. The response time of a task can actually exceed $T_i + T_{max}$ under G-EDF as illustrated in the following example.

Example 3.3. Consider a task system with five tasks $\tau_1 = (1, 4, 5)$, $\tau_2 = (3, 3, 4)$, $\tau_3 = (9, 19, 25)$, $\tau_4 = (20, 99, 100)$, $\tau_5 = (75, 70, 100)$ scheduled on four processors by G-EDF. It can be shown that the tardiness of the job $\tau_{4,48}$ is 204 time units, which is $T_i + T_{max} + 4$. ◀

3.2.3 An Alternate Response-Time Bound

Although Theorem 3.1 provides a response-time bound that is tight within a constant factor for pseudo-harmonic task systems, the bound can be overly pessimistic for systems with low total utilizations. To mitigate this pessimism, we now derive a response-time bound based on the per-task lag upper bound established in Lemma 3.20. Using a technique similar to that in [Devi and Anderson, 2005], we prove Theorem 3.6.

Definition 3.4. For any non-negative integer ℓ, let $\mathcal{H}_\ell$ denote the sum of the ℓ largest values of $(H + Y_i - Y_{min})u_i$. Thus, $\mathcal{H}_\ell = \sum_{\ell \text{ highest}} (H + Y_i - Y_{min})u_i$. ◀

Theorem 3.6. *The response time of task τ_i is at most $Y_i + x + C_i$ in $\mathcal{S}$, where*

$$x \geq \frac{\mathcal{H}_{\lceil U_{tot} \rceil - 1} + \sum_{j=1}^{N} \max\{(T_j - Y_j)u_j, 0\} - C_i}{M}. \tag{3.27}$$

We prove Theorem 3.6 by induction on job priorities. We consider an arbitrary job $\tau_{i,k}$ and inductively prove that its response time is no more than $Y_i + x + C_i$ in $\mathcal{S}$. Thus, we assume the following.

Assumption 3.2. The response time of each job with higher priority than $\tau_{i,k}$ is at most $Y_i + x + C_i$ in $\mathcal{S}$.

We now denote the jobs considered in Assumption 3.2 as follows.

Definition 3.5. Let ψ be the set of jobs that have higher priorities than $\tau_{i,k}$. Let $\Psi = \psi \cup \{\tau_{i,k}\}$. Let $\mathcal{S}'$ (resp., $\mathcal{I}'$) be the schedule of $\mathcal{S}$ (resp., $\mathcal{I}$) corresponding to the jobs Ψ. ◀

In schedule $\mathcal{S}$, job $\tau_{i,k}$ can only be delayed by the jobs in $\psi = \Psi \setminus \tau_{i,k}$. Note that jobs not in Ψ may be scheduled before $\tau_{i,k}$ when fewer than M jobs in Ψ are ready; however, such jobs will be preempted once new jobs in Ψ are released. Therefore, we only need to consider schedule $\mathcal{S}'$ to derive a response-time bound for $\tau_{i,k}$ in $\mathcal{S}$. The following lemma formally proves this.

Lemma 3.21. *The response time of each job $\tau_{j,\ell} \in \Psi$ in $\mathcal{S}$ is at most its response time in $\mathcal{S}'$.*

Proof. Assume otherwise. Let t be the earliest time instant at which a job, call it $\tau_{j,\ell}$, in Ψ completes execution in $\mathcal{S}'$ but not in $\mathcal{S}$. Then, there exists a time instant $t' < t$ such that $\tau_{j,\ell}$ executes in $\mathcal{S}'$ but not in $\mathcal{S}$. Since $\tau_{j,\ell}$ executes at t' in $\mathcal{S}'$, $\tau_{j,\ell-1}$ (if any) must have completed execution at or before t' in $\mathcal{S}'$. By the definition of t, $\tau_{j,\ell-1}$ (if any) also completes execution at or before t' in $\mathcal{S}$. Therefore, by Definition 2.5 and $P_i = 1$, $\tau_{j,\ell}$ is ready at time t' in $\mathcal{S}$. Since $\tau_{j,\ell}$ does not execute at time t' in $\mathcal{S}$, M tasks other than τ_j must have higher-priority jobs executing at t' in $\mathcal{S}$. By Definition 3.5, these jobs are in Ψ. By the definition of t', the jobs that execute at time t' in $\mathcal{S}$ have not completed execution by time t' in $\mathcal{S}'$. Therefore, there are M tasks with higher-priority jobs than $\tau_{j,\ell}$ at t' in $\mathcal{S}'$, implying that $\tau_{j,\ell}$ cannot execute at t' in $\mathcal{S}'$, a contradiction. □

To derive a response-time bound for $\tau_{i,k}$ in $\mathcal{S}$, we first upper bound LAG in $\mathcal{S}'$ at time $y(\tau_{i,k})$. For notational convenience, let $t_y = y(\tau_{i,k})$ and $t_f = f(\tau_{i,k})$. We assume that $t_f > t_y$; otherwise, the response time of $\tau_{i,k}$ is at most $Y_i \leq Y_i + x + C_i$, completing the proof.

Definition 3.6. Let t_0 be the earliest time instant such that $[t_0, t_y)$ is a busy interval (see Definition 3.3). ◄

Lemma 3.22. $\mathsf{LAG}(\Gamma, t_y, \mathcal{S}) \leq \mathcal{H}_{\lceil U_{tot} \rceil - 1}$.

Proof. We first upper bound $\mathsf{LAG}(\Psi, t_0, \mathcal{S})$. By Definitions 3.6 and 3.3, there are at most $\lceil U_{tot} \rceil - 1$ tasks with pending jobs at time $t_0 - 1$. Therefore, any task τ_j with no pending job at time $t_0 - 1$ either releases a new job at time t_0 or has no pending job at time t_0. If τ_j releases a new job at t_0, then $\mathsf{lag}(\tau_j, t_0, \mathcal{S}') = 0$, as all jobs of τ_j that are released before t_0 complete execution by t_0. If τ_j has no pending job at t_0, the by Lemma 3.4, $\mathsf{lag}(\tau_j, t_0, \mathcal{S}') < 0$ holds. Thus, only tasks that have pending jobs at time $t_0 - 1$ can have positive lag at time t_0. Since there are at most $\lceil U_{tot} \rceil - 1$ tasks with pending jobs at time $t_0 - 1$, by Lemma 3.20, Definition 3.4 and (3.5), we have

$$\mathsf{LAG}(\Gamma, t_0, \mathcal{S}') \leq \mathcal{H}_{\lceil U_{tot} \rceil - 1}.$$

Since $[t_0, t_y)$ is a busy interval, by Lemma 3.17, we have $\mathsf{LAG}(\Gamma, t_y, \mathcal{S}') \leq \mathcal{H}_{\lceil U_{tot} \rceil - 1}$. □

Let W be the total remaining workload of Γ at time t_y in $\mathcal{S}'$. The following lemma upper bounds W.

Lemma 3.23. $W \leq Mx + C_i$.

Proof. Let $W_1, W_2, \ldots, W_N$ denote the remaining workloads of tasks $\tau_1, \tau_2, \ldots, \tau_N$, respectively, at time t_y in $\mathcal{S}'$. Thus, we have

$$W = \sum_{j=1}^{N} W_j. \tag{3.28}$$

We now upper bound W by upper bounding each W_j. Let $\tau_{j,1}, \tau_{j,2}, \ldots, \tau_{j,\ell}$ be the jobs of τ_j that are in Ψ. Since only jobs in Ψ are in Γ, the total workload of τ_j in $\mathcal{S}'$ is $\sum_{x=1}^{\ell} C_j$. Thus, in $\mathcal{S}'$, the remaining workload of τ_j at time t_y is

$$W_j = \sum_{x=1}^{\ell} C_j - \mathsf{A}(\tau_j, 0, t_y, \mathcal{S}'). \tag{3.29}$$

Since $Y_j \geq 0$, no jobs released after t_y are in Ψ. Therefore, $r(\tau_{j,\ell}) \leq t_y$. We now consider two cases.

Case 1. $r(\tau_{j,\ell}) + T_j \leq t_y$. By the definition of $\mathcal{I}'$, jobs $\tau_{j,1}, \tau_{j,2}, \ldots, \tau_{j,\ell}$ complete execution by time t_y in $\mathcal{I}'$. Since only jobs in Ψ are in Γ, $\mathsf{A}(\tau_j, 0, t_y, \mathcal{I}') = \sum_{x=1}^{\ell} C_j$. Thus, by (3.29), we have $W_j = \mathsf{A}(\tau_j, 0, t_y, \mathcal{I}') - \mathsf{A}(\tau_j, 0, t_y, \mathcal{S}') = \mathsf{LAG}(\tau_j, t_y, \mathcal{S}') \leq \mathsf{LAG}(\tau_j, t_y, \mathcal{S}') + \min\{(T_j - Y_j)u_j, 0\}$.

Case 2. $r(\tau_{j,\ell}) + T_j > t_y$. In this case, all jobs prior to $\tau_{j,\ell}$ complete execution by time t_y in $\mathcal{I}'$. Since $\tau_{j,\ell}$ starts execution at time $r(\tau_{j,\ell})$ in $\mathcal{I}'$, it executes for $(t_y - r(\tau_{j,\ell}))u_j$ units by time t_y in $\mathcal{I}'$. Thus, $\mathsf{A}(\tau_j, 0, t_y, \mathcal{I}') = \sum_{x=1}^{\ell-1} C_j + (t_y - r(\tau_{j,\ell}))u_j$. By (3.29) and $T_j = C_j u_j$, we have

$$\begin{aligned}
W_j &= \sum_{x=1}^{\ell-1} C_j + T_j u_j - \mathsf{A}(\tau_j, 0, t_y, \mathcal{S}') \\
&= \sum_{x=1}^{\ell-1} C_j + (r(\tau_{j,\ell}) + T_j - t_y + t_y - r(\tau_{j,\ell}))u_j - \mathsf{A}(\tau_j, 0, t_y, \mathcal{S}') \\
&= \sum_{x=1}^{\ell-1} C_j + (t_y - r(\tau_{j,\ell}))u_j + (r(\tau_{j,\ell}) + T_j - t_y)u_j - \mathsf{A}(\tau_j, 0, t_y, \mathcal{S}') \\
&= \mathsf{A}(\tau_j, 0, t_y, \mathcal{I}') + (r(\tau_{j,\ell}) + T_j - t_y)u_j - \mathsf{A}(\tau_j, 0, t_y, \mathcal{S}') \\
&= \{\text{By (3.5)}\} \\
&\quad \mathsf{lag}(\tau_j, t_y, \mathcal{S}') + (r(\tau_{j,\ell}) + T_j - t_y)u_j \\
&\leq \{\text{Since } \tau_{j,\ell} \in \Psi, \text{ we have } y(\tau_{j,\ell}) = r(\tau_{j,\ell}) + Y_j \leq t_y\} \\
&\quad \mathsf{lag}(\tau_j, t_y, \mathcal{S}') + (T_j - Y_j)u_j
\end{aligned}$$

$$\leq \mathsf{lag}(\tau_j, t_y, \mathcal{S}') + \min\{(T_j - Y_j)u_j, 0\}.$$

Therefore, in both cases, $W_j \leq \mathsf{lag}(\tau_j, t_y, \mathcal{S}') + \min\{(T_j - Y_j)u_j, 0\}$. By (3.28), we have

$$\begin{aligned}
W &= \sum_{j=1}^{N} W_j \\
&\leq \sum_{j=1}^{N} \left(\mathsf{lag}(\tau_j, t_y, \mathcal{S}') + \min\{(T_j - Y_j)u_j, 0\}\right) \\
&= \{\text{By (3.6)}\} \\
&\quad \mathsf{LAG}(\Gamma, t_y, \mathcal{S}') + \sum_{j=1}^{N} \min\{(T_j - Y_j)u_j, 0\} \\
&\leq \{\text{By Lemma 3.22}\} \\
&\quad \mathcal{H}_{\lceil U_{tot} \rceil - 1} + \sum_{j=1}^{N} \min\{(T_j - Y_j)u_j, 0\} \\
&= M \cdot \frac{\mathcal{H}_{\lceil U_{tot} \rceil - 1} + \sum_{j=1}^{N} \min\{(T_j - Y_j)u_j, 0\} - C_i}{M} + C_i \\
&\leq \{\text{By (3.27)}\} \\
&\quad Mx + C_i.
\end{aligned}$$

Thus, the lemma holds. □

Using the upper bound on remaining workload at time t_y, we now prove Theorem 3.6.

Proof of Theorem 3.6. Assume that $\tau_{i,j}$ executes for e_i time units by time t_y. Thus, the remaining execution time of $\tau_{i,j}$ at time t_y is at most $C_i - e_i$. Let t_s be the first time instant at or after t_y when $\tau_{i,j}$ is scheduled in $\mathcal{S}'$. We prove the lemma by showing that $t_s \leq t_y + x + e_i$. We first show that $\tau_{i,k}$ becomes ready at or before time $t_y + x + e_i$. By Assumption 3.2, $\tau_{i,k-1}$ completes execution by time $r(\tau_{i,k-1}) + Y_i + x + C_i = r(\tau_{i,k}) - T_i + Y_i + x + C_i = t_y + x + C_i - T_i \leq t_y + x$ (since $C_i \leq T_i$). Thus, $\tau_{i,k-1}$ is ready by time $t_y + x \leq t_x + x + e_i$.

We now show that $t_s \leq t_y + x + e_i$. Assume otherwise. Then, M tasks other than τ_i have pending jobs at time $t_y + x + e_i$. Since only jobs in Ψ are present in $\mathcal{S}'$, all M processors must be busy executing jobs in Ψ during $[t_y, t_y + x + e_i + 1)$. Thus, the total work performed during $[t_y, t_y + x + e_i + 1)$ in $\mathcal{S}'$ is $M(x + e_i + 1)$.

Since $\tau_{i,j}$ executes for e_i time units by t_y, the total remaining workload of Ψ at t_y due to jobs other than $\tau_{i,j}$ is $W - C_i + e_i$. Since only jobs in Ψ executes during $[t_y, t_y + x + e_i)$, we have $M(x + e_i + 1) \leq W - C_i + e_i$, which implies that $W \geq M(x + e_i + 1) + C_i - e_i > Mx + C_i$, a contradiction to Lemma 3.23.

Thus, $t_s \leq t_y + x + e_i$. Since only jobs in Ψ are present in $\mathcal{S}'$, no jobs are released after time t_y. Therefore, $\tau_{i,j}$ continuously executes at or after time t_s until it finishes. Thus, $\tau_{i,k}$ finishes by time $t_y + x + e_i + C_i - e_i = t_y + x + C_i$ in $\mathcal{S}'$. Thus, by Lemma 3.21, the theorem holds. □

3.3 Exact Response-Time Bound

Having derived a response-time bound for periodic tasks that is tight in general for the pseudo-harmonic case, we now show how to derive an exact response-time bound. We do so by deriving an upper bound on the length of the prefix of a schedule during which tasks may experience increasing response times (afterwards, they do not). We show, in Lemma 3.27, that if there is a time instant $t \geq \Phi_{max}$ when LAG has the same values at t and $t - H$, then for any $t' \geq t$, the LAG values at t' and $t' - H$ are also equal. Intuitively, this implies that the schedule in the interval $[t - H, t)$ repeats after t. Moreover, since the lag of each task is bounded (Lemma 3.20), we can derive an upper bound on LAG (Lemma 3.26). Therefore, since LAG does not decrease over any interval of length H starting after Φ_{max} (LAG-monotonicity), there must be a finite interval $[\Phi_{max}, t')$ such that LAG strictly increases over any interval of length H in $[\Phi_{max}, t')$. We derive an upper bound on such an interval in Lemma 3.31. Intuitively, for each task, a job with the maximum response time of the task must complete at or before the schedule starts to repeat. We first consider task systems satisfying Assumption 3.1. We define a *response-time-increasing interval* as follows.

Definition 3.7. Given a periodic task system Γ, a *response-time-increasing interval* in a schedule $\mathcal{S}$ is a finite interval of time $[0, t]$ such that for each task $\tau_i \in \Gamma$, if the maximum response time of τ_i's jobs that complete execution at or before t is x_i in $\mathcal{S}$, then the response time of τ_i is x_i in $\mathcal{S}$. ◀

We now derive an upper bound on the response-time-increasing interval of a periodic task system Γ satisfying Assumption 3.1.

Definition 3.8. Let F be the sum of the $N - 1$ largest values of $C_i(1 - u_i)$; *i.e.,*

$$F = \sum_{N-1 \text{ largest}} C_i(1 - u_i).$$

Let G be the sum of the $\lceil U_{tot} \rceil - 1$ largest values of $(H + Y_i - Y_{min})u_i$; *i.e.,*

$$G = \sum_{\lceil U_{tot} \rceil - 1 \text{ largest}} (H + Y_i - Y_{min})u_i.$$

Finally, let $E = \lceil F + G + 1 \rceil$. ◀

The following lemma gives a trivial lower bound on the lag of a task at any time t in $\mathcal{S}$. A task's lag is minimum when its active job finishes execution as early as possible in $\mathcal{S}$, *i.e.,* C_i time units after its release.

Lemma 3.24. *For any task τ_i and time instant t,* $\mathsf{lag}(\tau_i, t, \mathcal{S}) \geq -C_i(1 - u_i)$.

Proof. If $t \leq \Phi_i$, then $\mathsf{lag}(\tau_i, t, \mathcal{S}) = 0$, so assume $t > \Phi_i$. Let $\tau_{i,k}$ be the job of τ_i with $r(\tau_{i,k}) \leq t < r(\tau_{i,k+1})$ and e_i be the total duration for which $\tau_{i,k}$ is scheduled at or before time t in $\mathcal{S}$. Therefore, $\mathsf{A}(\Gamma, \tau_i, 0, t, \mathcal{S}) = \sum_{j=1}^{k-1} C_i + e_i$. By the definition of $\mathcal{I}$, all jobs of τ_i prior to $\tau_{i,k}$ complete execution by time t in $\mathcal{I}$. Since jobs can only execute after their release, by the time $\tau_{i,k}$ executes for e_i units in $\mathcal{S}$, $\tau_{i,k}$ completed execution of at least $e_i u_i$ units in $\mathcal{I}$. Therefore, $\mathsf{A}(\Gamma, \tau_i, 0, t, \mathcal{I}) \geq \sum_{j=1}^{k-1} C_i + e_i u_i$. Substituting $\mathsf{A}(\Gamma, \tau_i, 0, t, \mathcal{I})$ and $\mathsf{A}(\Gamma, \tau_i, 0, t, \mathcal{S})$ in (3.3), we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) \geq \sum_{j=1}^{k-1} C_i + e_i u_i - \sum_{j=1}^{k-1} C_i - e_i = -e_i(1 - u_i)$. Since $e_i \leq C_i$, we have $\mathsf{lag}(\tau_i, t, \mathcal{S}) \geq -C_i(1 - u_i)$. □

We now give a lower bound on LAG at Φ_{max} in $\mathcal{S}$. By the definition of Φ_{max}, there is at least one task with lag that equals 0 at Φ_{max}.

Lemma 3.25. $\mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{S}) \geq -F$.

Proof. Let Γ' be the set of tasks such that for any $\tau_i \in \Gamma'$, $\Phi_i = \Phi_{max}$ holds. Therefore, $\mathsf{lag}(\tau_i, \Phi_{max}, \mathcal{S}) = 0$ holds for any $\tau_i \in \Gamma'$. Thus, $\sum_{\tau_i \in \Gamma'} \mathsf{lag}(\tau_i, \Phi_{max}, \mathcal{S}) = 0$. Hence, by (3.5), we have $\mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{S}) = \sum_{\tau_i \in \Gamma} \mathsf{lag}(\tau_i, \Phi_{max}, \mathcal{S}) = \sum_{\tau_i \in \Gamma \setminus \Gamma'} \mathsf{lag}(\tau_i, \Phi_{max}, \mathcal{S})$, which by Lemma 3.24 implies, $\mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{S}) \geq \sum_{\tau_i \in \Gamma \setminus \Gamma'} -C_i(1 - u_i)$. By the definition of Φ_{max}, $|\Gamma'| \geq 1$ holds. Therefore, by Definition 3.8, we have $\mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{S}) \geq -\sum_{N-1 \text{ largest}} C_i(1 - u_i) = -F$. □

We now derive an upper bound on LAG at any time t in $\mathcal{S}$ by determining an upper bound on LAG at the latest non-busy time instant at or before t.

Lemma 3.26. *For any t,* $\mathsf{LAG}(\Gamma, t, \mathcal{S}) \leq G$.

Proof. Let t_b be the latest non-busy time instant at or before t, otherwise let $t_b = 0$. We first derive an upper bound on $\mathsf{LAG}(\Gamma, t_b, \mathcal{S})$. If $t_b = 0$, then $\mathsf{LAG}(\Gamma, t_b, \mathcal{S}) = 0$. Otherwise, let $\Gamma' \subseteq \Gamma$ be the tasks with pending jobs at t_b. By (3.5),

$$\begin{aligned}
\mathsf{LAG}(\Gamma, t_b, \mathcal{S}) &= \sum_{\tau_i \in \Gamma'} \mathsf{lag}(\tau_i, t_b, \mathcal{S}) + \sum_{\tau_i \notin \Gamma'} \mathsf{lag}(\tau_i, t_b, \mathcal{S}) \\
&\leq \{\text{By Lemma 3.2, } (\forall \tau_i \notin \Gamma' : \mathsf{lag}(\tau_i, t_b, \mathcal{S}) \leq 0) \text{ holds}\} \\
&\quad \sum_{\tau_i \in \Gamma'} \mathsf{lag}(\tau_i, t_b, \mathcal{S}) \\
&\leq \{\text{By Lemma 3.20}\} \\
&\quad \sum_{\tau_i \in \Gamma'} (H + Y_i - Y_{min}) u_i \\
&\leq \{\text{By Definition 3.3, } |\Gamma'| < \lceil U_{tot} \rceil\} \\
&\quad \sum_{\lceil U_{tot} \rceil - 1 \text{ largest}} (H + Y_i - Y_{min}) u_i \\
&= \{\text{By Definition 3.8}\} \\
&\quad G.
\end{aligned}$$

By Lemma 3.17, $\mathsf{LAG}(\Gamma, t, \mathcal{S}) \leq \mathsf{LAG}(\Gamma, t_b, \mathcal{S}) \leq G$ holds. □

Lemmas 3.25 and 3.26 imply that LAG cannot increase more than $F + G$ units over any interval $[\Phi_{max}, t)$. We use this fact later in Lemma 3.31. We now show that once $\mathsf{LAG}(\Gamma, t, \mathcal{S}) = \mathsf{LAG}(\Gamma, t - H, \mathcal{S})$ holds for some t, the equality also holds for all time instances after t. Informally, by Lemma 3.18, if $\mathsf{LAG}(\Gamma, t, \mathcal{S}) = \mathsf{LAG}(\Gamma, t - H, \mathcal{S})$ holds, then for any task τ_i, $\mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{lag}(\tau_i, t - H, \mathcal{S})$ also holds. This implies that the scheduling decisions at t are the same as at $t - H$. Therefore, the schedule in $[t - H, t)$ repeats in $[t, t + H)$.

Lemma 3.27. *If there is a time instant $t' \geq \Phi_{max} + H$ such that $\mathsf{LAG}(\Gamma, t' - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t', \mathcal{S})$ holds, then for any $t \geq t'$, $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t, \mathcal{S})$ holds.*

Proof. Assume for a contradiction that there exists a $t \geq t'$ such that $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) \neq \mathsf{LAG}(\Gamma, t, \mathcal{S})$ and let t be the first such time instant. By the definition of t and t', $t > t' \geq \Phi_{max} + H$ and $t - 1 \geq \Phi_{max} + H$

hold. Therefore, $\mathsf{LAG}(\Gamma, t-H-1, \mathcal{S}) = \mathsf{LAG}(\Gamma, t-1, \mathcal{S})$ holds. Thus, by Lemma 3.18, we have

$$\forall \tau_i : \mathsf{lag}(\tau_i, t-H-1, \mathcal{S}) = \mathsf{lag}(\tau_i, t-1, \mathcal{S}). \tag{3.30}$$

Since T_i divides H, by (3.30) and Lemma 3.11(a) (with t and c replaced by $t-H-1$ and h_i, respectively), we have the following property.

Property 3.3. *Any task with no pending job at $t-H-1$ has no pending job at $t-1$.*

Let $\Gamma' \subseteq \Gamma$ be the set of tasks with pending jobs at $t-H-1$. Let $\tau_{i,k}$ be the ready job of $\tau_i \in \Gamma'$ at $t-H-1$. By Definition 3.2, (3.30) and Lemma 3.11(b) (with t and c replaced by $t-H-1$ and h_i, respectively), $\tau_{i,k+h_i}$ is the ready job of τ_i at $t-1$. Since τ_i releases jobs periodically, we have $y(\tau_{i,k+h_i}) = y(\tau_{i,k}) + h_i T_i = y(\tau_{i,k}) + H$. Thus, the tasks in Γ' have the same priority ordering at both $t-H-1$ and $t-1$, which along with Property 3.3 implies that the same set of tasks execute during both $[t-H-1)$ and $[t-1, t)$. Hence, $\mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{S}) = \mathsf{A}(\Gamma, t-1, t, \mathcal{S})$. Since $t-H-1 \geq \Phi_{max}$, we have $\mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{I}) = \mathsf{A}(\Gamma, t-1, t, \mathcal{I})$. Thus, by (3.6) we have

$$\begin{aligned}
\mathsf{LAG}(\Gamma, t, \mathcal{S}) &= \mathsf{LAG}(\Gamma, t-1, \mathcal{S}) + \mathsf{A}(\Gamma, t-1, t, \mathcal{I}) - \mathsf{A}(\Gamma, t-1, t, \mathcal{S}) \\
&= \{\text{Since } \mathsf{LAG}(\Gamma, t-1, \mathcal{S}) = \mathsf{LAG}(\Gamma, t+H-1)\} \\
&\quad \mathsf{LAG}(\Gamma, t-H-1, \mathcal{S}) + \mathsf{A}(\Gamma, t-1, t, \mathcal{I}) - \mathsf{A}(\Gamma, t-1, t, \mathcal{S}) \\
&= \{\text{Since } \mathsf{A}(\Gamma, t-1, t, \mathcal{S}) = \mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{S}) \text{ and} \\
&\qquad\qquad \mathsf{A}(\Gamma, t-1, t, \mathcal{I}) = \mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{I})\} \\
&= \mathsf{LAG}(\Gamma, t-H-1, \mathcal{S}) + \mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{I}) - \mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{S}) \\
&= \{\text{By (3.6)}\} \\
&\quad \mathsf{LAG}(\Gamma, t-H, \mathcal{S}),
\end{aligned}$$

a contradiction. □

Lemma 3.28. *If there is a time instant $t' \geq \Phi_{max} + H$ such that $\mathsf{LAG}(\Gamma, t'-H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t', \mathcal{S})$ holds, then $\mathsf{lag}(\tau_i, t-H, \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S})$ holds for any $t \geq t'$ and $\tau_i \in \Gamma$.*

Proof. Follows from Lemmas 3.27 and 3.18. □

Using the above lemma, we now show that whenever LAG becomes equal at hyperperiod boundaries, $\mathcal{S}$ starts to repeat.

Lemma 3.29. *If there is a time instant $t' \geq \Phi_{max} + H$ such that $\mathsf{LAG}(\Gamma, t' - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t', \mathcal{S})$ holds, then for any $t \geq t'$ and $\tau_i \in \Gamma$, $\tau_{i,k-h_i}$ is ready at time $t - H$ if and only if $\tau_{i,k}$ is ready at time t.*

Proof. By Lemma 3.28, $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S})$ holds. Assume that $\tau_{i,k-h_i}$ is ready at time $t - H$. Then, by Lemma 3.11(b), $\tau_{i,k}$ is ready at time t. Similarly, assuming $\tau_{i,k}$ is ready at time t and applying Lemma 3.11(b), $\tau_{i,k-h_i}$ is ready at time $t - H$. □

Lemma 3.30. *If there is a time instant $t' \geq \Phi_{max} + H$ such that $\mathsf{LAG}(\Gamma, t' - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t', \mathcal{S})$ holds, then for any $t \geq t'$ and $\tau_i \in \Gamma$, $\tau_{i,k-h_i}$ is scheduled at time $t - H$ if and only if $\tau_{i,k}$ is scheduled at time t.*

Proof. Follows from Lemma 3.30, (3.1), and prioritization Rule PR. □

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(b), $\mathsf{LAG}(\Gamma, 6, \mathcal{S}) = \mathsf{LAG}(\Gamma, 12, \mathcal{S})$ holds. Therefore, for all tasks τ_i and $t \geq 12$, $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t, \mathcal{S}) = 2$ and $\mathsf{lag}(\tau_i, t - H, \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S})$ hold. We now show that there is a time instant t after $\Phi_{max} + H$ and at or before $\Phi_{max} + EH$ where LAG has the same value at t and $t - H$. Therefore, the schedule starts to repeat at or before $\Phi_{max} + EH$. Intuitively, LAG must increase by at least 1.0 execution unit, if not equal, over each interval $[\Phi_{max} + iH, \Phi_{max} + (i + 1)H)$ where $0 \leq i < E$. Therefore, since $E = \lceil F + G + 1 \rceil$, LAG at $\Phi_{max} + EH$ must be more than G, contradicting Lemma 3.26.

Lemma 3.31. *There is a time instant $t \in [\Phi_{max}+H, \Phi_{max}+EH]$ such that $\mathsf{LAG}(\Gamma, t-H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t, \mathcal{S})$ holds.*

Proof. Assume $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) \neq \mathsf{LAG}(\Gamma, t, \mathcal{S})$ holds for all $t \in [\Phi_{max} + H, \Phi_{max} + EH]$. Let t be any arbitrary time instant in $[\Phi_{max} + H, \Phi_{max} + EH]$. Since $t \geq \Phi_{max} + H$, by Corollary 3.1, we have $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) \leq \mathsf{LAG}(\Gamma, t, \mathcal{S})$. Thus, $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) < \mathsf{LAG}(\Gamma, t, \mathcal{S})$ holds. Since tasks release jobs periodically and $t - H \geq \Phi_{max}$ holds, we have

$$\mathsf{A}(\Gamma, t - H, t, \mathcal{I}) = U_{tot} H. \tag{3.31}$$

Since $U_{tot} = \sum_{i=1}^{N} \frac{C_i}{T_i}$ and $h_i = H/T_i$, we have $U_{tot} = \frac{\sum_{i=1}^{N} h_i C_i}{H}$. Therefore, $U_{tot} H = \sum_{i=1}^{N} h_i C_i$. Since both h_i and C_i are integers, $U_{tot} H$ is also an integer. By (3.6), we have

$$\begin{aligned}
\mathsf{A}(\Gamma, t-H, t, \mathcal{S}) &= \mathsf{A}(\Gamma, t-H, t, \mathcal{I}) + \mathsf{LAG}(\Gamma, t-H, \mathcal{S}) - \mathsf{LAG}(\Gamma, t, \mathcal{S}) \\
&< \{\text{Since } \mathsf{LAG}(\Gamma, t-H, \mathcal{S}) < \mathsf{LAG}(\Gamma, t, \mathcal{S})\} \\
&\quad \mathsf{A}(\Gamma, t-H, t, \mathcal{I}) \\
&= \{\text{Since } \mathsf{A}(\Gamma, t-H, t, \mathcal{I}) = U_{tot} H\} \\
&\quad U_{tot} H. \qquad (3.32)
\end{aligned}$$

Since $U_{tot} H$ and $\mathsf{A}(\Gamma, t-H, t, \mathcal{S})$ are integers, by (3.32) we have

$$\mathsf{A}(\Gamma, t-H, t, \mathcal{S}) \leq U_{tot} H - 1. \qquad (3.33)$$

Now, by (3.6), we have

$$\begin{aligned}
\mathsf{LAG}(\Gamma, \Phi_{max} + EH, \mathcal{S}) &= \mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{S}) + \mathsf{A}(\Gamma, \Phi_{max}, \Phi_{max} + EH, \mathcal{I}) - \mathsf{A}(\Gamma, \Phi_{max}, \Phi_{max} + EH, \mathcal{S}) \\
&= \{\text{Since } [\Phi_{max}, \Phi_{max} + EH) = \bigcup_{i=0}^{E-1} [\Phi_{max} + iH, \Phi_{max} + (i+1)H)\} \\
&\quad \mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{S}) + \sum_{i=0}^{E-1} (\mathsf{A}(\Gamma, \Phi_{max} + iH, \Phi_{max} + (i+1)H, \mathcal{I}) \\
&\qquad - \mathsf{A}(\Gamma, \Phi_{max} + iH, \Phi_{max} + (i+1)H, \mathcal{S})) \\
&\geq \{\text{Substituting } t = \Phi_{max} + (i+1)H \text{ in (3.31) and (3.33)}\} \\
&\quad \mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{S}) + \sum_{i=0}^{E-1} (U_{tot} H - U_{tot} H + 1) \\
&= \mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{S}) + \sum_{i=0}^{E-1} 1 \\
&\geq \{\text{By Lemma 3.25 and Definition 3.8}\} \\
&\quad -F + F + G + 1 \\
&> G,
\end{aligned}$$

a contradiction to Lemma 3.26. □

Therefore, Lemma 3.31 establishes a *simulation interval* of $\Phi_{max} + EH$ for $\mathcal{S}$ (by Lemma 3.30). Note that, by Definition 3.8, E depends on task parameters, task count, and processor count. Thus, the simulation interval contains pseudo-polynomial number of hyperperiods.

We now show that a job with the maximum response time must complete execution at or before $\Phi_{max} + EH$ by Lemma 3.32 and Theorem 3.7.

Lemma 3.32. *If there is a time instant $t' \geq \Phi_{max} + H$ such that $\mathsf{LAG}(\Gamma, t' - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t', \mathcal{S})$ holds and x_i is the maximum response time of any of task τ_i's jobs that complete execution at or before t' in $\mathcal{S}$, then the response time of τ_i is x_i in $\mathcal{S}$.*

Proof. Assume that the response time of τ_i is more than x_i and let $\tau_{i,k}$ be the first job with response time exceeding x_i. Let t be the time instant when $\tau_{i,k}$ finishes execution. Then, $t > t'$ holds. Since $\mathsf{LAG}(\Gamma, t' - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t', \mathcal{S})$ and $t - 1 \geq t'$ hold, by Corollary 3.28, we have $\mathsf{lag}(\tau_i, t - H - 1, \mathcal{S}) = \mathsf{lag}(\tau_i, t - 1, \mathcal{S})$. Since $h_i = H/T_i$ and $\tau_{i,k}$ is pending at $t - 1$, substituting t and c in Lemma 3.11(b) by $t - 1$ and $-h_i$, respectively, $\tau_{i,k-h_i}$ is pending at $t - 1 - H$. Therefore, $\tau_{i,k-h_i}$ finishes execution at or after $t - H$. Hence, we have

$$
\begin{aligned}
f(\tau_{i,k-h_i}) - d(\tau_{i,k-h_i}) &\geq t - H - d(\tau_{i,k-h_i}) \\
&= \{\text{Since } \tau_i \text{ releases periodically, } d(\tau_{i,k-h_i}) = d(\tau_{i,k}) - h_i T_i\} \\
&\quad t - H - d(\tau_{i,k}) + h_i T_i \\
&= \{\text{By the definition of } t \text{ and Definition 3.2}\} \\
&\quad f(\tau_{i,k}) - d(\tau_{i,k}).
\end{aligned}
$$

Therefore, $\max\{0, f(\tau_{i,k-h_i}) - d(\tau_{i,k-h_i})\} \geq \max\{0, f(\tau_{i,k}) - d(\tau_{i,k})\}$ holds and $\tau_{i,k}$'s response time cannot exceed $\tau_{i,k-h_i}$'s response time. □

For the task system in Example 3.1 and its G-EDF schedule in Figure 3.1(b), $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t, \mathcal{S})$ holds for the first time at time 12. Job $\tau_{3,1}$ has the maximum response time in $\mathcal{S}$.

Theorem 3.7. *If the maximum response time of a task τ_i's jobs that completes at or before $\Phi_{max} + EH$ is x_i in $\mathcal{S}$, then the response time of τ_i is x_i in $\mathcal{S}$.*

Proof. By Lemma 3.31, there is a time instant $t \in [\Phi_{max} + H, \Phi_{max} + EH]$ such that $\mathsf{LAG}(\Gamma, t - H, \mathcal{S}) = \mathsf{LAG}(\Gamma, t, \mathcal{S})$ holds. Let z_i be the maximum response time of τ_i's jobs that complete execution at or before t in $\mathcal{S}$. By Lemma 3.32, the response time of τ_i is z_i. Since $t \in [\Phi_{max} + H, \Phi_{max} + EH]$, by the definition of x_i, $z_i \leq x_i$ holds. Since the response time of τ_i in $\mathcal{S}$ is z_i, $z_i \geq x_i$ holds. Therefore, $x_i = z_i$. □

By Theorems 3.4 and 3.7, if the maximum response time of τ_i's jobs that complete at or before $\Phi_{max} + EH$ is x_i in a GEL schedule $\mathcal{S}$ satisfying Assumption 3.1, then τ_i's response time is at most x_i in a GEL schedule $\mathcal{S}'$ not satisfying Assumption 3.1.

Deriving response times. By Theorem 3.7, we can determine an exact response-time bound of each task by simulating a schedule up to time $\Phi_{max} + EH$. By Definition 3.8, we have $F = \sum_{N-1 \text{ largest}} C_i(1 - u_i) \leq \sum_{i=1}^{N} C_i = \sum_{i=1}^{N} T_i u_i \leq H \sum_{i=1}^{N} u_i \leq mH$. By Definition 3.8, $G = \sum_{\lceil U_{tot} \rceil - 1 \text{ largest}} (H + Y_i - Y_{min}) u_i \leq \sum_{M-1 \text{ largest}} (H + Y_{max}) = (M - 1)(H + Y_{max})$ holds. Therefore, we have $E = \lceil F + G + 1 \rceil \leq \lceil MH + (M - 1)(H + Y_{max}) + 1 \rceil$. For pseudo-harmonic systems, since scheduling decisions at each time instant are determined in polynomial time and $H = T_{max}$, simulating a schedule up to time $\Phi_{max} + ET_{max}$ takes pseudo-polynomial time. By Lemma 3.32, we can terminate the simulation early at time $t \geq \Phi_{max} + H$ by checking whether $\mathsf{LAG}(\Gamma, t, \mathcal{S}) = \mathsf{LAG}(\Gamma, t - H, \mathcal{S})$ holds. This would require storing the last H values of LAG at any time. We can also store one value of LAG at any time, *e.g.,* the last time instant that is multiple of H, and check for LAG-equality H time after the last-stored instance. This would require simulating for at most H time units more than that required when H values of LAG are stored. We note that this method can be adapted for non-pseudo-harmonic systems with H and G replaced with H and a corresponding upper bound on LAG, respectively.

3.4 Experimental Evaluation

We now present the results of simulation experiments we conducted to evaluate our response-time bounds and the effectiveness of our approach to derive exact response-time bounds.

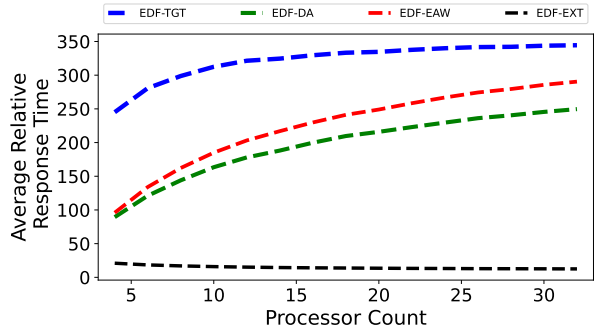
We generated task systems randomly for systems with 4 to 32 processors. We chose *light*, *medium*, *heavy*, or *wide* task utilizations, for which task utilizations were uniformly distributed in $[0.01, 0.3]$, $[0.3, 0.7]$, $[0.7, 1]$, and $[0.01, 1]$, respectively. We chose the maximum task period from 100ms to 1000ms with a step size of 100ms. Each task period was chosen randomly from all factors of the maximum task period. In case there was no task with the maximum period, we randomly chose a task and scaled its parameters to set its period to

the maximum period. We rounded down each execution cost to its nearest integer and disregarded any task if its execution cost became zero. We chose the offset of each task randomly between 0 and its period. For each utilization cap M and utilization distribution, we generated 1,000 task systems by adding tasks until five attempts to add a next task without exceeding the utilization cap failed.

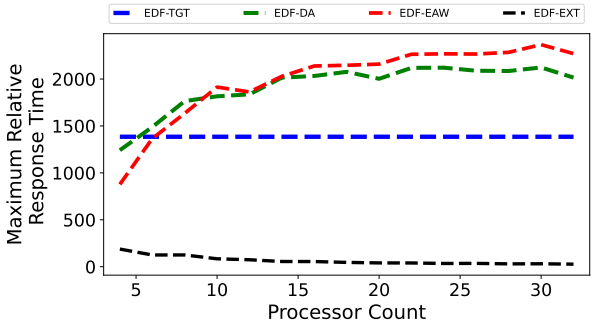
We used *relative response-time bounds* as our evaluation metric, where a task's relative response-time bound is computed by dividing its response time by its period. For each task system, we computed exact response-time bounds using Theorem 3.7 and response-time bounds using Theorems 3.1 and 3.6 (taking the minimum among these two bounds) under G-EDF (EDF-EXT and EDF-TGT, respectively) and G-FIFO (FIFO-EXT and FIFO-TGT, respectively). We also computed response-time bounds under G-EDF (resp., G-FIFO) using methods by Devi and Anderson [Devi and Anderson, 2008] (EDF-DA) and Erickson *et al.* [Erickson et al., 2014] (EDF-EAW) (resp., Leontyev and Anderson [Leontyev and Anderson, 2007] (FIFO-LA) and Erickson *et al.* [Erickson et al., 2014] (FIFO-EAW)). We did not compare against the bound under G-EDF from [Valente, 2016] as it is computationally expensive to compute and has trends similar to EDF-DA [Valente, 2016] (In our attempt to compute response-time bounds from [Valente, 2016] using the most efficient implementation from [Leoncini et al., 2019], we found that computing tardiness bounds for a task system on 16 or more processors can take several hours to complete). We measured the time taken to compute EDF-EXT and FIFO-EXT for each task system. We present a representative selection of our results in Figures 3.6–3.11.

Observation 3.1. *For heavy utilizations, the average relative response-time bound for EDF-TGT was 64.77% and 43.31% larger than for EDF-DA and EDF-EAW, respectively. The maximum relative response-time bound for EDF-TGT was 42.06% and 43.17% smaller than for EDF-DA and EDF-EAW, respectively. The average and maximum relative response-time bounds for FIFO-TGT were 35.01% and 76.20% smaller (resp., 42.93% larger and 45.75% smaller) than for FIFO-LA (resp., FIFO-EAW), respectively.*

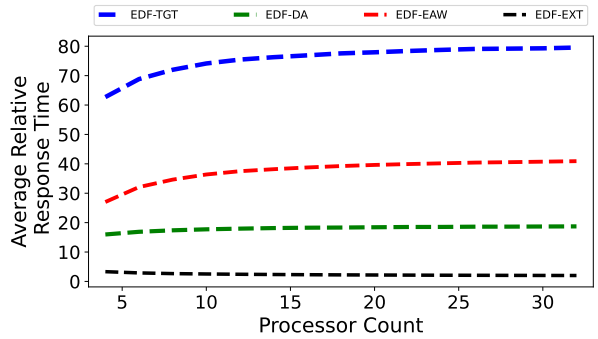
This can be seen in insets (a) and (b) of Figures 3.6 and 3.7. For heavy task utilizations, the mean for EDF-DA and EDF-EAW (resp., FIFO-EAW) were smaller than EDF-TGT (resp., FIFO-TGT). This is because the relative response-time bounds for EDF-TGT and FIFO-TGT are similar for every task in a system and contain T_{max} causing large relative response-time bounds for tasks with small periods. However, the maximum for EDF-DA and EDF-EAW (resp., FIFO-EAW) are generally larger than EDF-TGT (resp., FIFO-TGT).



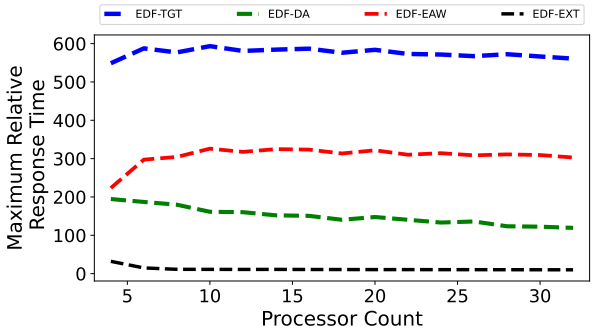
(a) Average relative response-time bound for heavy task utilizations with respect to processor count.



(b) Maximum relative response-time bound for heavy task utilizations with respect to processor count.



(c) Average relative response-time bound for light task utilizations with respect to processor count.

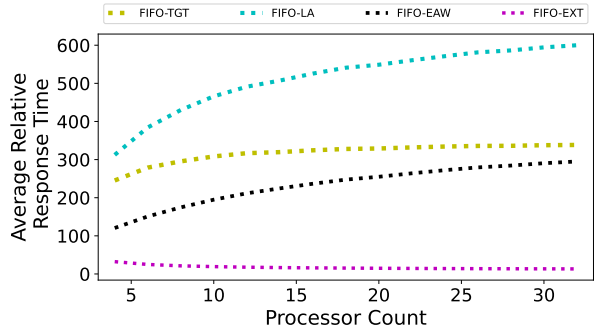


(d) Maximum relative response-time bound for light task utilizations with respect to processor count.

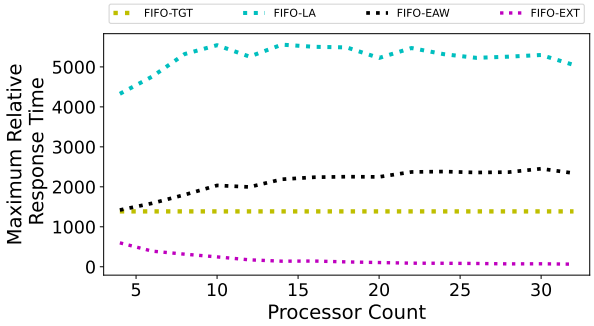
Figure 3.6: Average and maximum response-time bound under G-EDF with respect to the number of processors.

Observation 3.2. *For light utilizations, the average and maximum relative response-time bounds for EDF-TGT were 1091.6% and 561.14% (resp., 470.23% and 276.43%) larger than for EDF-DA (resp., EDF-EAW), respectively. The average and maximum relative response-time bounds for FIFO-TGT were 201.58% and 69.69% (resp., 288.16% and 113.52%) larger than for FIFO-LA (resp., FIFO-EAW), respectively.*

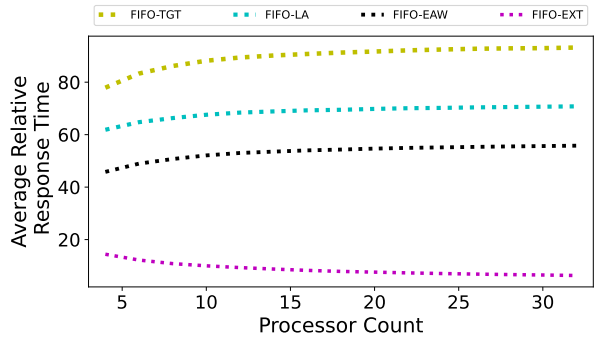
This can be seen in insets (c) and (d) of Figures 3.6 and 3.7. EDF-DA (resp., FIFO-LA) is tighter than EDF-TGT (resp., FIFO-TGT) for light per-task utilizations. This is because EDF-DA and FIFO-LA are functions of the sum of largest $\lceil U_{tot} \rceil - 1$ task utilizations and task WCETs. When task utilizations are small, task WCETs are also smaller, leading to tighter response-time bounds for EDF-DA, EDF-EAW, FIFO-LA, and FIFO-EAW.



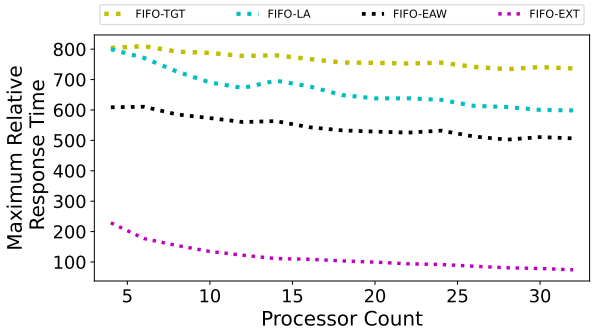
(a) Average relative response-time bound for heavy task utilizations with respect to processor count.



(b) Maximum relative response-time bound for heavy task utilizations with respect to processor count.



(c) Average relative response-time bound for light task utilizations with respect to processor count.



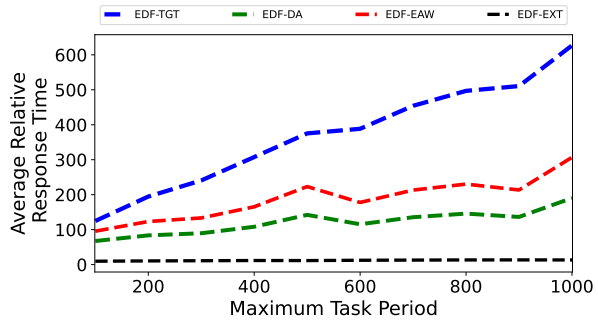
(d) Maximum relative response-time bound for light task utilizations with respect to processor count.

Figure 3.7: Average and maximum response-time bound under G-FIFO with respect to the number of processors.

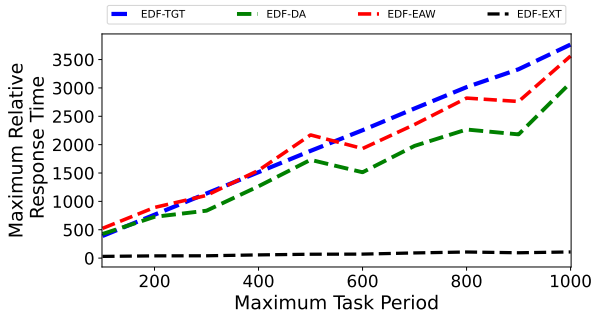
Observation 3.3. *Across all task systems, the average relative response time for EDF-EXT and FIFO-EXT was 0.83 and 1.47, respectively. The maximum relative response time for EDF-EXT and FIFO-EXT was 37.5 and 101.2, respectively.*

This can be seen in Figures 3.6 and 3.7. Average and maximum relative response time are usually smaller under G-EDF than G-FIFO. Note that relative response times are normalized bounds with respect to task periods. Under G-FIFO, tasks with small periods can have large response times if they their execution is delayed by earlier released "long" jobs. Such jobs have large response times. In contrast, under G-EDF, typically tasks with large periods are preempted multiple times, leading to their large response times. However, average and maximum response time can be larger under G-EDF than G-FIFO (see Example 3.3).

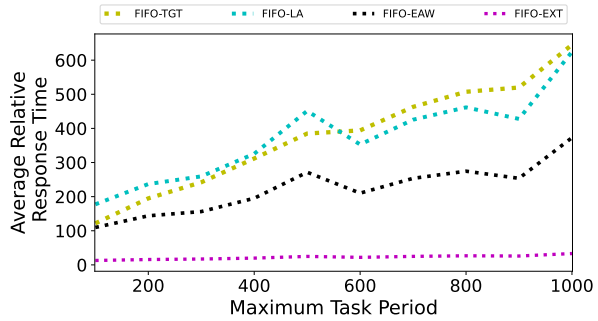
Observation 3.4. *EDF-TGT, EDF-DA, EDF-EAW, FIFO-TGT, FIFO-LA, and FIFO-EAW increased with respected to the maximum task period.*



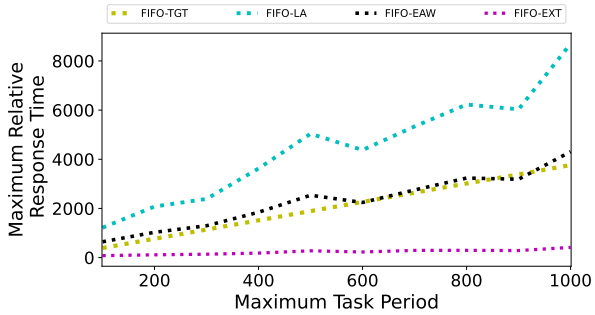
(a) Average relative response-time bound for wide task utilizations with respect to task periods.



(b) Maximum relative response-time bound for wide task utilizations with respect to task periods.



(c) Average relative response-time bound for wide task utilizations with respect to task periods.

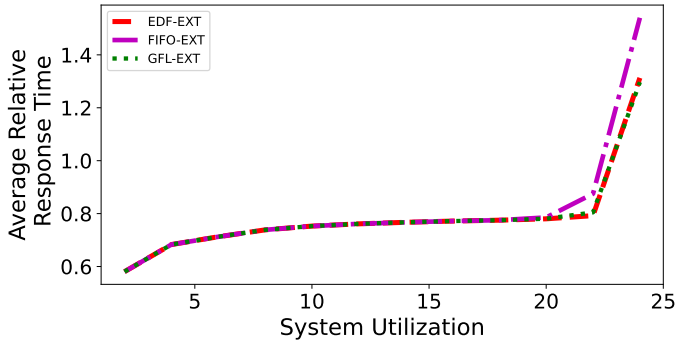


(d) Maximum relative response-time bound for wide task utilizations with respect to task periods.

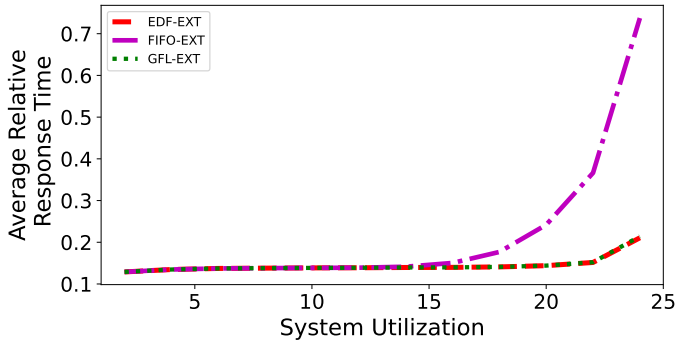
Figure 3.8: Average and maximum response-time bound under G-FIFO with respect to the number of processors.

This can be seen in insets (a)–(d) of Figure 3.8. Under G-EDF scheduling, EDF-TGT includes both the maximum task period and task deadlines, resulting in a greater increase in average response-time bounds with respect to the maximum period compared to EDF-DA and EDF-EAW. In contrast, for G-FIFO scheduling, the average and maximum response-time bounds for FIFO-TGT increase at similar rates compared to FIFO-LA and FIFO-EAW as the maximum task period increases.

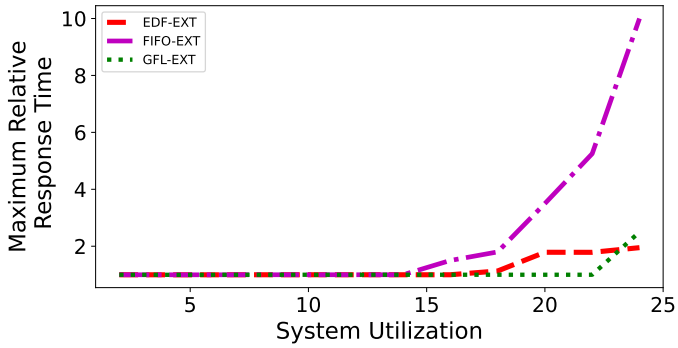
To compare exact response-time bounds under G-FIFO and G-EDF with respect to system utilization, we considered a 24-processor platform and generated 1,000 task systems for each utilization cap within $[2, 24]$ with a step size of 2. In addition to G-FIFO and G-EDF, we also derived exact response-time bound (GFL-EXT) under *global fair-lateness* (G-FL) scheduler, which achieves smallest response-time bounds among all GEL schedulers under analysis techniques of Erickson *et al.* [Erickson et al., 2014]. Insets (a), (b), and (c) of Figure 3.9 present the results.



(a) Average exact relative response-time bound for heavy task utilizations with respect to system utilization.



(b) Average exact relative response-time bound for light task utilizations with respect to system utilization.

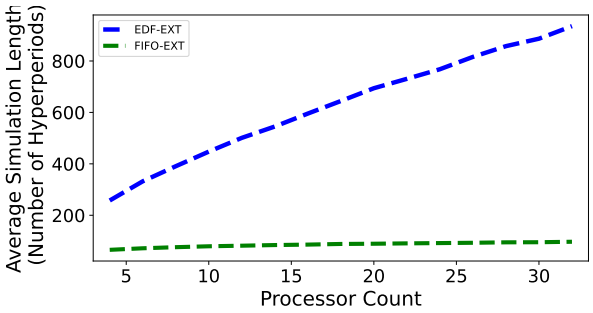


(c) Maximum exact relative response-time bound for wide task utilizations with respect to system utilization.

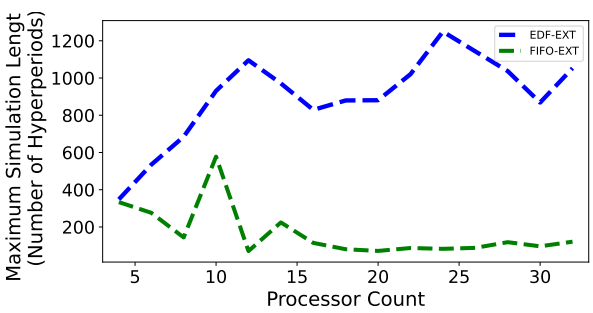
Figure 3.9: Average and maximum response-time bound with respect to system utilizations.

Observation 3.5. *The average and maximum relative response times for EDF-EXT and GFL-EXT were smaller than FIFO-EXT.*

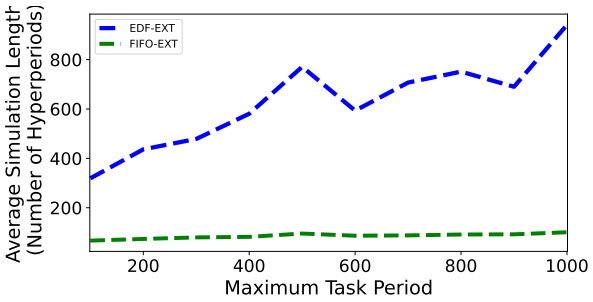
Figure 3.10 presents average and maximum simulation length for EDF-EXT and FIFO-EXT. In general, simulation interval lengths for G-EDF were larger than G-FIFO.



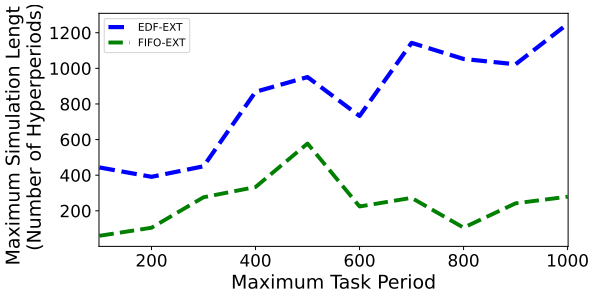
(a) Average simulation length vs. processor count.



(b) Maximum simulation length vs. processor count.



(c) Average simulation length vs. maximum task period.



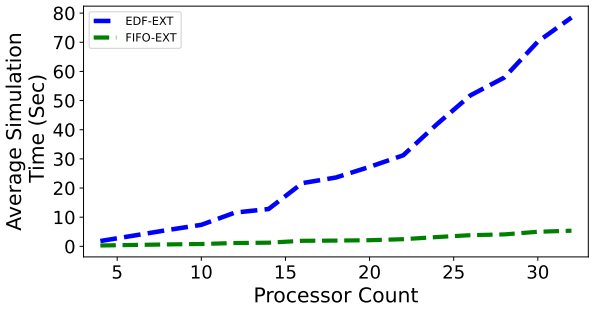
(d) Maximum simulation length vs. maximum task period.

Figure 3.10: Average and maximum simulation lengths.

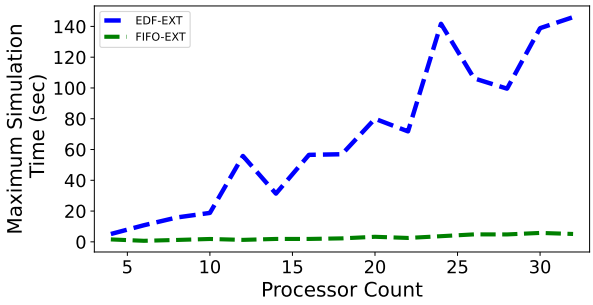
Observation 3.6. *Across all task systems, the average simulation length for EDF-EXT and FIFO-EXT was 167.47 and 23.32 hyperperiods. The maximum simulation length for EDF-EXT and FIFO-EXT was 1249.36 and 577.84 hyperperiods.*

Figure 3.11 presents computation time for EDF-EXT and FIFO-EXT. These plots imply that exact response-time bounds can often be efficiently computed. The average and maximum simulation time for EDF-EXT are larger than FIFO-EXT due to larger simulation lengths and more preemptions. The running time increases when the number of processors is large. Note that the running time may increase significantly when T_{max} is large.

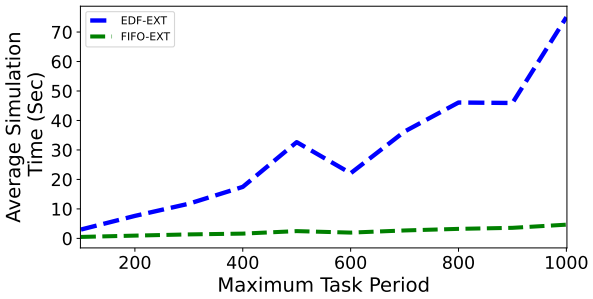
Observation 3.7. *Across all task systems, the average time to compute EDF-EXT and FIFO-EXT was 6.7 and 0.55 sec, respectively. The maximum time to compute EDF-EXT and FIFO-EXT was 146.03 and 5.79 sec, respectively. (These computations were done on 2.50 GHz Intel processors with 30M cache and 32GB RAM.)*



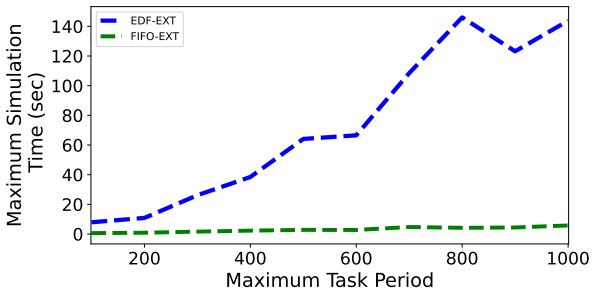
(a) Average simulation time vs. processor count.



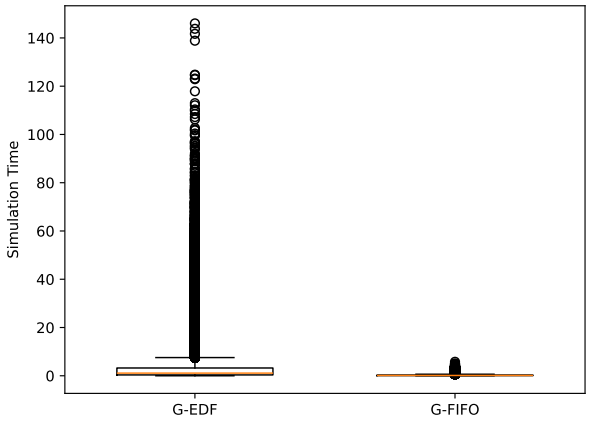
(b) Maximum simulation time vs. processor count.



(c) Average simulation time vs. maximum task period.



(d) Maximum simulation time vs. maximum task period.



(e) Simulation time for G-EDF and G-FIFO.

Figure 3.11: Average and maximum simulation time.

3.5 Chapter Summary

In this chapter, we have presented a response-time bound for periodic tasks under GEL schedulers. This is the first response-time bound for pseudo-harmonic systems under GEL scheduler that does not increase with respect to the number of tasks or processors. We have shown the tightness of our bound for pseudo-harmonic periodic tasks and provided a method to determine similar response-time bounds for sporadic tasks. We have

also provided a method to compute exact response-time bounds for pseudo-harmonic periodic tasks under GEL schedulers.

CHAPTER 4: SERVER-BASED SCHEDULING OF DAG TASKS[1]

In this chapter, we consider the scheduling of SRT DAG tasks on an identical multiprocessor. Since the concurrent execution of the same task is common in many applications involving DAG tasks, *e.g.,* computer vision applications, we consider each task to be specified under the rp model. Therefore, we must deal with all sources of interference and dependencies mentioned in Table 2.3.

To deal with such challenges, we give a *server*-based scheduling technique and show that this scheduler is SRT-optimal. The design of server-based scheduling is motivated by the goal of applying simulation-based response-time analysis techniques, presented in Chapter 3, to derive exact response-time bounds of DAG tasks under the rp model. Recall that a simulation-based approach involves upper bounding both the time to reach schedule repetition (transient state) and the period of schedule repetition (steady state). This is difficult for DAG tasks, mainly due to variations in node activation times caused by precedence constraints. Using per-node reservation servers allows us to provide an upper bound on the processor capacity allocated to a node over a hyperperiod, a property that we exploit to ensure schedule repetition of DAG tasks.

Organization. After covering needed background (Section 4.1), we describe our server-based approach (Section 4.2), explain how to obtain exact DAG response-time bounds (Sections 4.3 and 4.4), discuss our experiments (Section 4.5), and provide a summary (Section 4.6).

4.1 System Model

In this section, we present the considered DAG task model. Table 4.1 summarizes the notation given here.

Task model. We consider a task system Γ consisting of N *periodic* DAG tasks to be scheduled on M identical processors. Each *DAG task* G^v has n^v nodes that represent sequential tasks $\{\tau_1^v, \tau_2^v, \ldots, \tau_{n^v}^v\}$. We use the term *node* and *task* interchangeably. A directed edge from τ_i^v to τ_k^v represents a precedence constraint

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

Ahmed, S. and Anderson, J. (2022), Exact Response-Time Bounds of Periodic DAG Tasks under Server-Based Global Scheduling, *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 349—358.

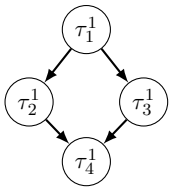


Figure 4.1: A DAG G^1.

between the *predecessor* task τ_i^v and the *successor* task τ_i^v. The set of predecessors of τ_i^v is denoted by $pred(\tau_i^v)$. Each DAG task G^v has a unique *source* task τ_1^v with no incoming edge and a unique *sink* task $\tau_{n^v}^v$ with no outgoing edge. This assumption is made for simplicity; the results presented in this chapter are also applicable to DAG tasks with multiple source and sink tasks.

Each DAG task G^v has a *period* T^v. DAG task G^v releases a *DAG job* every T^v time units. The j^{th} DAG job of G^v is denoted by G_j^v. DAG task G^v has an *offset* Φ^v, which is the time when G_1^v is released. The DAG job G_j^v consists of a *job* $\tau_{i,j}^v$ for each task τ_i^v in that DAG. Each job $\tau_{i,j}^v$ is preemptive. The *release time* and *finish time* of $\tau_{i,j}^v$ are denoted by $r(\tau_{i,j}^v)$ and $f(\tau_{i,j}^v)$, respectively. The source task τ_i^v releases its j^{th} job when G_j^v is released. The j^{th} job of each non-source task is released once the j^{th} job of all of its predecessor tasks finish, *i.e.,* $r(\tau_{i,j}^v) = \max_{\tau_k^v \in pred(\tau_i^v)}\{f(\tau_{k,j}^v)\}$.

The *response time* of job $\tau_{i,j}^v$ is $f(\tau_{i,j}^v) - r(\tau_{1,j}^v)$, *i.e.,* the time duration between the source task's job release and $\tau_{i,j}^v$'s completion. The response time of task τ_i^v is $\sup_j\{f(\tau_{i,j}^v) - r(\tau_{1,j}^v)\}$. A DAG job G_j^v completes when $\tau_{n^v,j}^v$ finishes, *i.e.,* when all jobs associated with the DAG job complete. G_j^v's response time equals the response time of corresponding job $\tau_{n^v,j}^v$ of the sink task. G^v's response time equals $\tau_{n^v}^v$'s response time.

Example 4.1. Figure 4.1 depicts a DAG G^1. The source and sink tasks of G^1 are τ_1^1 and τ_1^4, respectively. The first DAG job G_1^1 of G^1 consists of jobs $\tau_{1,1}^1, \tau_{2,1}^1, \tau_{3,1}^1$, and $\tau_{4,1}^1$. DAG job G_1^1's completes when all these jobs complete execution. G_1^1's response time is the time difference between $\tau_{4,1}^1$'s completion time and $\tau_{1,1}^1$'s release time. ◀

The WCET of τ_i^v is denoted by C_i^v. The *utilization* of τ_i^v is $u_i^v = C_i^v/T^v$. The utilization of G^v is $U^v = \sum_{i=1}^{n^v} u_i^v$. The total utilization of all DAG tasks is $U_{tot} = \sum_{v=1}^{N} U^v$. We let $T_{max} = \max_v\{T^v\}$, $C_{max} = \max_{v,i}\{C_i^v\}$, and $\Phi_{max} = \max_v\{\Phi^v\}$. The *hyperperiod* H is the LCM of all periods. A task system is *pseudo-harmonic* if each period divides T_{max}, *i.e.,* $H = T_{max}$ holds.

Each DAG task can have a *relative deadline*. We do not explicitly specify DAG deadlines, as our technique does not rely on it. We consider multiple DAG jobs of a DAG task can be present at the same time; this happens for SRT DAG tasks or HRT DAG tasks with deadlines larger than periods. We assume that concurrent execution of two jobs of a task is specified according to the rp model (recall from Section 2.1). Thus, each task τ_i^v has an arbitrary *parallelization level* P_i^v, meaning that P_i^v successive jobs of task τ_i^v can execute concurrently. We assume that different tasks of the same DAG task can have different parallelization levels. Under the rp model, pending and ready jobs are defined as follows. Finally, we assume that the task system Γ satisfies the following condition.

$$U_{tot} \leq M \wedge (\forall v, i : u_i^v \leq P_i^v)$$

Note that the above condition is an exact condition for SRT-feasibility of DAG task on identical multiprocessors (Theorem 2.1).

4.2 Server-Based Scheduling of DAG Tasks

To schedule DAG tasks, we adopt a server-based policy where a global scheduler allocates time to per-node *reservation servers*, upon which task jobs are scheduled. To illustrate this scheduling, we first introduce *reservation servers*.

Reservation servers. For each task τ_i^v, we define a periodic *reservation server* S_i^v. We denote the set of all servers as Γ_s. Each server S_i^v has a period T^v and a budget C_i^v. Note that S_i^v's period and budget are the same as G^v's period and τ_i^v's WCET, respectively. Each server S_i^v releases a (potentially infinite) sequence of *server jobs* $S_{i,1}^v, S_{i,2}^v, \ldots$. The *release time* and *finish time* of $S_{i,j}^v$ are denoted by $r(S_{i,j}^v)$ and $f(S_{i,j}^v)$, respectively. Server S_i^v releases its first job $S_{i,1}^v$ at time Φ^v. Its subsequent jobs are released periodically, *i.e.,* $S_{i,j}^v$ is released at time $\Phi^v + (j-1)T^v$. Therefore, we have

$$\forall v, i, j : r(S_{i,j}^v) = r(\tau_{1,j}^v).$$

We do not require deadlines to be hard, *i.e.,* server jobs can miss their deadlines. Server S_i^v has a parallelization level of P_i^v matching that of τ_i^v.

Table 4.1: Notation summary for Chapter 4.

Symbol	Meaning
Γ	Task system
N	Number of DAG tasks
M	Number of processors
G^v	v^{th} DAG
T^v	Period of G^v
Φ^v	Offset of G^v
τ_i^v	i^{th} task of G^v
C_i^v	WCET of τ_i^v
u_i^v	Utilization of τ_i^v
S_i^v	Server of τ_i^v
Y_i^v	RPP of S_i^v
$R(\cdot)$	Response-time bound of task or server
U_{tot}	Utilization of Γ
T_{max}	$\max_v\{T^v\}$
Φ_{max}	$\max_v\{\Phi^v\}$
H	Hyperperiod
h^v	H/T^v
$\tau_{i,j}^v$	j^{th} job of τ_i^v
$S_{i,j}^v$	j^{th} job of S_i^v
$r(\cdot)$	Release time of job or server job
$f(\cdot)$	Completion time of $\tau_{i,j}^v$
$y(S_{i,j}^v)$	PP of $S_{i,j}^v$
$\mathcal{S}$	An arbitrary schedule
$\mathcal{I}$	Ideal schedule
$\mathsf{A}(\cdot)$	Allocation of job, task, or system (Definition 3.1)
$\mathsf{lag}(\cdot)$	lag of job or task
$\mathsf{LAG}(\cdot)$	LAG of Γ or Γ_s
Δ	Definition 4.5
E, F, G	Definition 4.6

Server jobs are scheduled according to a GEL scheduler. Thus, each server S_i^v has a *relative PP*, denoted by Y_i^v. Each server job has a PP $y(S_{i,j}^v)$, which is determined as

$$y(S_{i,j}^v) = r(S_{i,j}^v) + Y_i^v. \tag{4.1}$$

Before elaborating on the scheduling of servers, we first give budget consumption and replenishment rules.

Replenishment Rule. The budget of $S_{i,j}^v$ is replenished to C_i^v when it is released.

Consumption Rule. $S_{i,j}^v$ consumes budget at the rate of one execution unit per unit of time when it is scheduled until its budget is exhausted.

Similar to task jobs, we define pending and ready server jobs below.

Definition 4.1. A server job $S_{i,j}^v$ is *complete* after its budget is exhausted. $S_{i,j}^v$ is *pending* at time t if $r(S_{i,j}^v) \leq t < f(S_{i,j}^v)$. $S_{i,j}^v$ is *ready* if it is pending and $S_{i,j-P_i^v}^v$ (if $j > P_i^v$) is complete. ◀

Under GEL scheduling, the (up to) M ready server jobs with earliest PPs are scheduled. We assume ties are broken arbitrarily but consistently by DAG and task indices. Thus, we modify the Rule PR in Chapter 2.2 as follows.

PR. Server job $S_{i,j}^v$ has higher priority than server job $S_{k,\ell}^w$ if and only if

$$y(S_{i,j}^v) < y(S_{k,\ell}^w) \vee \left(y(S_{i,j}^v) = y(S_{k,\ell}^w) \wedge (v < w \vee (v = w \wedge i < j))\right).$$

Thus, if $y(S_{i,j}^v) = y(S_{k,\ell}^w)$ and $y(S_{i,p}^v) = y(S_{k,q}^w)$, then $S_{i,j}^v$ has higher priority than $S_{k,\ell}^w$ if and only if $S_{i,p}^v$ has higher priority than $S_{k,q}^w$. The response time of $S_{i,j}^v$ (resp., S_i^v) is $f(S_{i,j}^v) - r(S_{i,j}^v)$ (resp., $\sup_j\{f(S_{i,j}^v) - r(S_{i,j}^v)\}$).

Example 4.2. Consider the DAG G^1 shown in Figure 4.1. Assume that the period of G^1 is 5.0 time units, and $C_1^1 = 2.0, C_2^1 = 3.0, C_3^1 = 2.0$, and $C_4^1 = 3.0$. The parallelization level of each task is one. There are four servers each corresponding to a task. Figure 4.2 illustrates a G-EDF schedule of these servers. Each server has a period of 5.0 time units and releases is first server job at time 0 when τ_1^1 releases its first job. The server S_2^1 corresponding to the task τ_2^1 has a budget of 3.0 units. Server job $S_{2,1}^1$'s budget is replenished

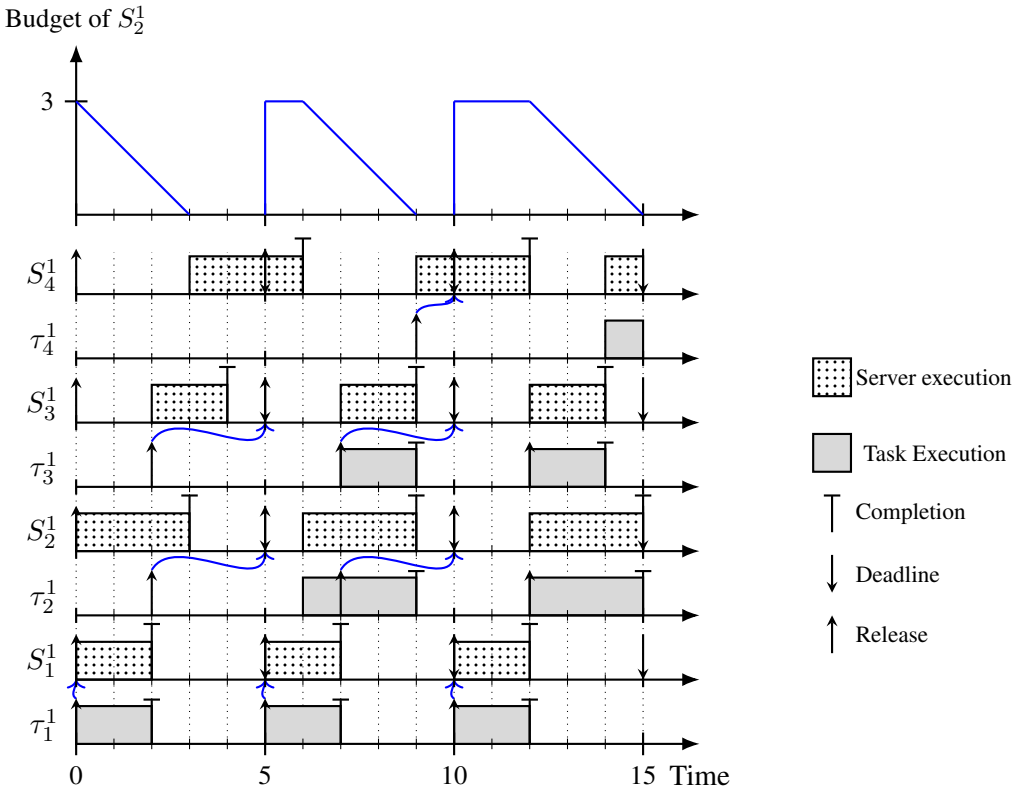


Figure 4.2: Illustration of server-based scheduling for the DAG G^1 in Figure 4.1. Blue arrows between job and server job releases represent job linking.

to 3.0 units when $S_{2,1}^1$ is released at time 0. It consumes its budget by one unit per unit of time when it is scheduled. $S_{2,1}^1$ completes at time 3 when its budget is exhausted. ◀

Scheduling tasks on servers. Jobs are scheduled on servers via the following rules.

R1. Jobs of τ_i^v are scheduled on server jobs of S_i^v. A job $\tau_{i,j}^v$ is *linked* to a single server job $S_{i,\ell}^v$ (via Rule R2 given below) and at most one job can be linked to a server job.

R2. Assume that a server job $S_{i,\ell}^v$ is released at time t. If $r(\tau_{i,1}^v) \leq t$ holds and $\tau_{i,1}^v$ is not linked to any server job at time t, then $\tau_{i,1}^v$ is linked to $S_{i,\ell}^v$. Otherwise, if $\tau_{i,j}^v$ is the last job of τ_i^v that is linked to some server job and $r(\tau_{i,j+1}^v) \leq t$ holds, then $\tau_{i,j+1}^v$ is linked to $S_{i,\ell}^v$.

R3. If $\tau_{i,j}^v$ is linked to $S_{i,\ell}^v$, then $\tau_{i,j}^v$ executes whenever $S_{i,\ell}^v$ is scheduled until $\tau_{i,j}^v$ completes.

Here, Rule R2 prohibits any out-of-order linking that can happen if a job $\tau_{i,j+1}^v$ is released before its prior job $\tau_{i,j}^v$. Since job releases are determined by precedence constraints, such a scenario can happen for DAG tasks.

For example, considering the DAG in Figure 4.1, if the server jobs of $\tau^v_{1,j}$ and $\tau^v_{1,j+1}$ are scheduled at the same time and the actual execution time of $\tau^v_{1,j}$ is smaller, then $\tau^v_{2,j+1}$ is released earlier than $\tau^v_{2,j}$. Rule R2 ensures that $\tau^v_{2,j}$ is linked before $\tau^v_{2,j+1}$ even if it is released later. Without such enforcement, $\tau^v_{2,j+1}$ may execute at a higher priority than $\tau^v_{2,j}$ due to their out-of-order linking.

Example 4.2 (Continued)**.** Figure 4.2 depicts a schedule of G^1. At time 0, the first job of each server is released. Since $\tau^1_{1,1}$ is released at time 0, by Rule R2, it is linked to $S^1_{1,1}$. By Rule R3, $\tau^1_{1,1}$ executes when $S^1_{1,1}$ is scheduled during $[0, 2)$. At time 2, $\tau^1_{2,1}$ and $\tau^1_{3,1}$ are released. At time 5, when $S^1_{2,1}$ (resp., $S^1_{3,1}$) is released, $\tau^1_{2,1}$ (resp., $S^1_{3,1}$) is linked to it. ◀

As seen in Figure 4.2(b), an unlinked server job has unused budget. In Section 4.4, we give slack-reallocation rules to utilize such unused budgets without violating response-time bounds.

4.3 Basic Response-Time Bound

In this section, we give non-exact response-time bounds for DAG tasks under the server-based scheduling given in Section 4.2. These bounds establish the SRT-optimality of server-based scheduling. Moreover, recall that, in Chapter 3, we have used an upper bound on LAG to determine the simulation length needed for computing exact response-time bounds. Similarly, we will use the bounds derived in this section to derive a bound on the simulation length in Section 4.4.

Server response-time bounds. As server tasks are periodic and have restricted parallelism, previously derived response-time bounds apply to them [Amert et al., 2019].

Definition 4.2. Let $\mathcal{U}_b = \sum_{b \text{ largest values of } \tau^v_i \text{ with } P^v_i < M} u^v_i$ and $\mathcal{C}_b = \sum_{b \text{ largest values of } \tau^v_i \text{ with } P^v_i < M} u^v_i$. ◀

From [Amert et al., 2019], S^v_i has a response-time bound $R(S^v_i)$, where

$$R(S^v_i) = T^v + \frac{(M-1)C_{max} + 2\mathcal{C}_{M-1}}{m - \mathcal{U}_{M-1}} + C^v_i. \tag{4.2}$$

Response-time bounds of DAG tasks. Using the response-time bounds of servers given in (4.2), we now derive response-time bounds of DAG tasks under server-based scheduling.

Lemma 4.1. *If $S^v_{i,j}$ and $S^v_{i,k}$ are ready at time t where $j < k$ and $S^v_{i,k}$ is scheduled at time t, then $S^v_{i,j}$ is also scheduled at time t.*

Proof. Since $j < k$ and S_i^v releases periodically, $r(S_{i,j}^v) < r(S_{i,k}^v)$ holds. Thus, $y(\tau_{i,j}^v) < y(\tau_{i,k}^v)$ holds. Having higher priority and being ready, $S_{i,j}^v$ is thus scheduled if $S_{i,k}^v$ is. □

Due to Lemma 4.1, an earlier server job of a server completes at or before the later server jobs of the server.

Lemma 4.2. *For any j and k with $j \leq k$, $f(S_{i,j}^v) \leq f(S_{i,k}^v)$.*

Proof. Assume for a contradiction that the lemma does not hold, in which case $j < k$ clearly holds. Let t be the first time instant such that there are server jobs $S_{i,j}^v$ and $S_{i,k}^v$ such that $j < k$, $t < f(S_{i,j}^v)$, and $t = f(S_{i,k}^v)$. Since, by the budget Consumption Rule, $S_{i,j}^v$ and $S_{i,k}^v$ are scheduled for C_i^v time units before their completion and $f(S_{i,j}^v) > f(S_{i,k}^v)$, there is a time instant $t' \leq t$ such that $S_{i,j}^v$ is not scheduled at t', but $S_{i,k}^v$ is scheduled at t'.

We now prove that $S_{i,j}^v$ is ready at time t', which by Lemma 4.1 implies that it is scheduled at time t', a contradiction. Since $S_{i,k}^v$ is scheduled at t' and $j < k$, $r(S_{i,j}^v) < r(S_{i,k}^v) \leq t'$ holds. By the definition of t and t', $t' < f(S_{i,j}^v)$. If $j < P_i^v$, then $S_{i,j}^v$ is ready at time t' as claimed, so assume $j \geq P_i^v$. Since $S_{i,k}^v$ is scheduled (hence ready) at time t', $S_{i,k-P_i^v}^v$ completes by time t'. As $j < k$, by the definition of t, $f(S_{i,j-P_i^v}^v) \leq f(S_{i,k-P_i^v}^v)$. Thus, $S_{i,j-P_i^v}^v$ completes by time t' and $S_{i,j}^v$ is ready then. □

Lemma 4.3. *If a job $\tau_{i,j}^v$ is ready when its linked server job $S_{i,k}^v$ is first scheduled, then $f(\tau_{i,j}^v) \leq f(S_{i,k}^v)$.*

Proof. By the budget Consumption Rule, $S_{i,k}^v$ is scheduled for C_i^v time units. Since $\tau_{i,j}^v$ is ready when $S_{i,k}^v$ is first scheduled, by Rule R3, $\tau_{i,j}^v$ completes execution at or before $S_{i,k}^v$'s budget is exhausted. Therefore, $f(\tau_{i,j}^v) \leq f(S_{i,k}^v)$ holds. □

From Rule R2, we have the following three lemmas that relate a job's index with its linked server job's index.

Lemma 4.4. *If $\tau_{i,j}^v$ is linked to a server job $S_{i,k}^v$, then $j \leq k$.*

Proof. Follows from Rule R2. □

Lemma 4.5. *If $\tau_{i,j}^v$ and $\tau_{i,j+1}^v$ are linked to $S_{i,k}^v$ and $S_{i,\ell}^v$, respectively, then $\ell > k$ holds.*

Proof. By Rule R2, $\tau_{i,j+1}^v$ is linked to $S_{i,\ell}^v$ if $\tau_{i,j}^v$ is the last job of τ_i^v that is linked to a server job when $S_{i,\ell}^v$ is released. Therefore, $S_{i,k}^v$ is released before $S_{i,\ell}^v$. Thus, $\ell > k$ holds. □

Lemma 4.6. *If $\tau_{i,j}^v$ and $\tau_{i,j+c}^v$ are linked to $S_{i,k}^v$ and $S_{i,\ell}^v$, respectively, then $\ell - k \geq c$ holds.*

Proof. Follows from Lemma 4.5. □

We now define a response-time bound $R(\tau_i^v)$ for each τ_i^v. $R(\tau_i^v)$ is recursively computed according to τ_i^v's predecessors' response-time bounds. Let

$$R(\tau_i^v) = O_i^v + R(S_i^v) + T^v, \tag{4.3}$$

$$\text{where} \quad O_i^v = \begin{cases} 0 & i = 1 \\ \max\limits_{\tau_j^v \in pred(\tau_i^v)} \{R(\tau_j^v)\} & \text{otherwise.} \end{cases} \tag{4.4}$$

In Theorem 4.1, we show that $R(\tau_i^v)$ is a response-time bound of τ_i^v using Lemmas 4.7–4.10 given below.

Lemma 4.7. *For any job $\tau_{i,j}^v$, $\tau_{i,j}^v$ is ready at or before its linked server job $S_{i,k}^v$ starts execution.*

Proof. Assume otherwise. Let t be the first time instant such that there is a job $\tau_{i,j}^v$ that is not ready, but its linked server job $S_{i,k}^v$ starts execution at time t. By Rule R2, $r(\tau_{i,j}^v) \leq t$. Since $\tau_{i,j}^v$ is not ready at time t, $j > P_i^v$ holds and $\tau_{i,j-P_i^v}^v$ does not complete execution at or before time t. By Lemma 4.4, $k \geq j > P_i^v$ holds.

We now prove that $\tau_{i,j-P_i^v}^v$ completes by time t, *i.e.,* $f(\tau_{i,j-P_i^v}^v) \leq t$, thereby reaching a contradiction. By Lemma 4.6, $\tau_{i,j-P_i^v}^v$ is linked to $S_{i,\ell}^v$ with $\ell \leq k - P_i^v$. Let t' be the first time instant when $S_{i,\ell}^v$ is scheduled. Since $S_{i,k}^v$ is scheduled at time t, $f(S_{i,k-P_i^v}^v) \leq t$. Thus, by Lemma 4.2, $f(S_{i,\ell}^v) \leq t$. Since $C_i^v > 0$, we have $t' < f(S_{i,\ell}^v) \leq t$. Hence, by the definition of t, $\tau_{i,j-P_i^v}^v$ is ready when $S_{i,\ell}^v$ is first scheduled. By Lemma 4.3, $f(\tau_{i,j-P_i^v}^v) \leq f(S_{i,\ell}^v) \leq t$. Thus, $\tau_{i,j}^v$ is ready at time t. □

By Lemmas 4.7 and 4.3, we have the following lemma.

Lemma 4.8. *For any job $\tau_{i,j}^v$, $\tau_{i,j}^v$ completes execution at or before its linked server job $S_{i,k}^v$ completes.*

Proof. Follows from Lemmas 4.7 and 4.3. □

Using (4.3) and (4.4), we have the following lemma.

Lemma 4.9. *For any non-source task τ_i^v, $O_i^v \geq O_k^v + R(S_k^v) + T^v$ holds, where $\tau_k^v \in pred(\tau_i^v)$.*

Proof. By (4.3) and (4.4), we have $O_i^v \geq R(\tau_k^v) = O_k^v + R(S_k^v) + T^v$. □

Lemma 4.10. *For any job $\tau_{i,j}^v$, $\tau_{i,j}^v$ is linked to a server job at or before time $r(\tau_{1,j}^v) + O_i^v + T^v$.*

Proof. Assume for a contradiction that t is the first time instant such that a job $\tau_{i,j}^v$ is not linked to any server job and $t = r(\tau_{1,j}^v) + O_i^v + T^v$ holds. Let $S_{i,k}^v$ be the latest server job of S_i^v released at or before time t. We will show that $\tau_{i,j}^v$ is linked to $S_{i,k}^v$, thereby reaching a contradiction. Since S_i^v releases jobs periodically, $r(S_{i,k}^v) \geq t - T^v = r(\tau_{1,j}^v) + O_i^v$. By Rule R2, it suffices to prove that $r(\tau_{i,j}^v) \leq r(\tau_{1,j}^v) + O_i^v$ holds and $\tau_{i,j-1}^v$ (if $j > 1$) is linked to a server job by time $r(\tau_{1,j}^v) + O_i^v$.

Claim 4.1. *$\tau_{i,j}^v$ is released at or before $r(\tau_{1,j}^v) + O_i^v$.*

Proof. Assume otherwise. Since $\tau_{1,j}^v$ is released at time $r(\tau_{1,j}^v)$ and by (4.4), $O_1^v = 0$, we have $i \neq 1$. Thus, τ_i^v is a non-source task. Since τ_i^v releases $\tau_{i,j}^v$ once the j^{th} job of each of its predecessors completes, there is a job $\tau_{p,j}^v$ such that $\tau_p^v \in pred(\tau_i^v)$ and $f(\tau_{p,j}^v) > r(\tau_{1,j}^v) + O_i^v$ hold. By Lemma 4.9, we have $O_p^v < O_i^v$. Thus, $r(\tau_{1,j}^v) + O_p^v + T^v < r(\tau_{1,j}^v) + O_i^v + T^v = t$. Therefore, by the definition of t, $\tau_{p,j}^v$ is linked to a server job at or before time $r(\tau_{1,j}^v) + O_p^v + T^v$. Assume that $\tau_{p,j}^v$ is linked to $S_{p,\ell}^v$. Then, by Rule R2, $r(S_{p,\ell}^v) \leq r(\tau_{1,j}^v) + O_p^v + T^v$. Since the response time of $S_{p,\ell}^v$ is at most $R(S_p^v)$, we have

$$\begin{aligned} f(S_{p,\ell}^v) &\leq r(S_{p,\ell}^v) + R(S_p^v) \\ &\leq \{\text{Since } r(S_{p,\ell}^v) \leq r(\tau_{1,j}^v) + O_p^v + T^v\} \\ &\quad r(\tau_{1,j}^v) + O_p^v + T^v + R(S_p^v) \\ &\leq \{\text{By Lemma 4.9 and since } \tau_p^v \in pred(\tau_i^v)\} \\ &\quad r(\tau_{1,j}^v) + O_i^v. \end{aligned} \tag{4.5}$$

By Lemma 4.8 and (4.5), we have $f(\tau_{p,j}^v) \leq f(S_{p,\ell}^v) \leq r(\tau_{1,j}^v) + O_i^v$, a contradiction. □

Claim 4.2. *If $j > 1$, then $\tau_{i,j-1}^v$ is linked to a server job at or before time $r(\tau_{1,j}^v) + O_i^v$.*

Proof. Since source task τ_1^v releases jobs periodically, we have $r(\tau_{1,j}^v) = r(\tau_{1,j-1}^v) + T^v$. Thus, $r(\tau_{1,j-1}^v) + O_i^v + T^v = r(\tau_{1,j}^v) + O_i^v < r(\tau_{1,j}^v) + O_i^v + T^v$ holds. Therefore, by the definition of t, $\tau_{i,j-1}^v$ is linked to a server job at or before time $r(\tau_{1,j-1}^v) + O_i^v + T^v = r(\tau_{1,j}^v) + O_i^v$. □

These two claims yield a contradiction, as noted above. □

By (4.3) and Lemmas 4.10 and 4.8, we have the following theorem.

Theorem 4.1. *The response time of any job $\tau_{i,j}^v$ is at most $R(\tau_i^v)$.*

Proof. By Lemma 4.10, $\tau_{i,j}^v$ is linked to a server job $S_{i,k}^v$ released by time $r(\tau_{1,j}^v) + O_i^v + T^v$. Since $S_{i,k}^v$'s response time is at most $R(S_i^v)$, $S_{i,k}^v$ finishes execution by time $r(\tau_{1,j}^v) + O_i^v + T^v + R(S_i^v)$. By Lemma 4.8, $\tau_{i,j}^v$ finishes execution by time $r(\tau_{1,j}^v) + O_i^v + T^v + R(S_i^v)$. Thus, the response time of $\tau_{i,j}^v$ is at most $O_i^v + T^v + R(S_i^v)$. By (4.3), $R(\tau_i^v) = O_i^v + T^v + R(S_i^v)$ holds. □

4.4 Exact Response-Time Bound

In this section, we give a simulation-based method to compute exact response-time bounds of DAG tasks under the server-based scheduling policy given in Section 4.2. We leverage the lag-monotonicity (Lemma 3.13) and LAG-monotonicity (Corollary 3.1) properties used in Chapter 3 for deriving exact bounds for sequential tasks. However, the presence of concurrent ready jobs of the same task complicates establishing and using such properties. We initially assume the following, which we relax later.

Assumption 4.1. Each job of any task τ_i^v executes for its WCET C_i^v.

Note that the response-time bounds (and associated lemmas and theorems) given in Section 4.3 do not rely on Assumption 4.1. We begin by defining lag and LAG for DAG tasks and servers.

4.4.1 Definitions and Notation

We denote a GEL schedule of Γ_s as $\mathcal{S}$. We denote a schedule of Γ on the server schedule $\mathcal{S}$ as $\mathcal{G}$. Similar to the response-time bound proof of Chapter 3, we will use the concept of lag and LAG. Similar to Chapter 3, we first define a "hypothetical" ideal schedule for servers and DAG tasks.

Ideal schedule. Let $\{\widehat{\pi}_1^1, \widehat{\pi}_2^1, \ldots, \widehat{\pi}_{n^N}^N\}$ be $\sum_{v=1}^N n^v$ processors, where $\widehat{\pi}_i^v$ has speed of u_i^v. In an *ideal schedule* $\mathcal{I}$, each task τ_i^v and corresponding server S_i^v is partitioned to be scheduled on processor $\widehat{\pi}_i^v$. Each server's budget is replenished according to the budget Replenishment Rule given in Section 4.3. However, its budget is consumed via the following rule.

> **Ideal Consumption Rule.** $S_{i,j}^v$ consumes budget at the rate of u_i^v execution unit per unit of time when it is scheduled until its budget is exhausted.

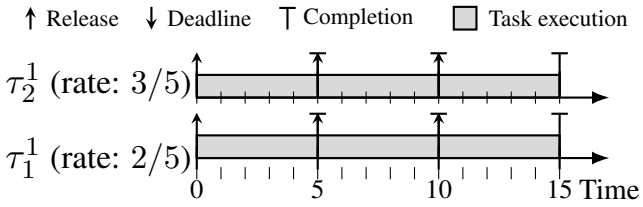


Figure 4.3: An ideal schedule of τ_1^v and τ_2^v of G^1 in Figure 4.1. Server jobs are not shown as they have the same schedule.

Each server job $S_{i,j}^v$ is scheduled at time $r(S_{i,j}^v) = r(\tau_{1,j}^v)$ and remains scheduled until its budget is exhausted. Therefore, $S_{i,j}^v$ completes at time $r(S_{i,j}^v) + C_i^v/u_i^v = r(S_{i,j}^v) + T^v$.

Each job $\tau_{i,j}^v$ executes at a rate of u_i^v whenever $S_{i,j}^v$ is scheduled. Thus, $\tau_{i,j}^v$ begins execution at time $r(\tau_{1,j}^v)$ and completes execution at time $r(\tau_{1,j}^v) + T^v$. Therefore, all jobs corresponding to a DAG job G_j^v start execution upon G_j^v's release and complete execution at time $r(\tau_{1,j}^v) + T^v$ when G_{j+1}^v is released. Note that precedence constraints among tasks are not maintained in $\mathcal{I}$.

Example 4.2 (Continued)**.** Figure 4.3 depicts an ideal schedule $\mathcal{I}$ corresponding to the tasks τ_1^1 and τ_2^1 of DAG task G^1 in Figure 4.2. Although job $\tau_{1,1}^1$ is not complete at time 0, $\tau_{2,1}^1$ starts execution at time 0 at the rate of $3/5$ execution units per unit of time. ◀

We now define the term *allocation*. To avoid repetition, we use the notation $J_{i,j}^v$ to denote $\tau_{i,j}^v$ or $S_{i,j}^v$. Similarly, we use J_i^v (resp., Ψ) to denote τ_i^v or S_i^v (resp., Γ or Γ_s). Finally, we use $\mathcal{H}$ to denote a schedule that can be either $\mathcal{G}$ or $\mathcal{S}$ or $\mathcal{I}$.

Allocation. The *cumulative processor capacity* (as defined by Definition 3.1) allocated to job $J_{i,j}^v$, task J_i^v, and system Ψ in a schedule $\mathcal{H}$ over an interval $[t, t')$ is denoted by $\mathsf{A}(J_{i,j}^v, t, t', \mathcal{H})$, $\mathsf{A}(J_i^v, t, t', \mathcal{H})$, and $\mathsf{A}(\Psi, t, t', \mathcal{H})$, respectively. The allocation of a task (resp., system) over interval $[t, t')$ is the sum of all of its jobs' (resp., tasks') allocation over interval $[t, t')$. Therefore, we have the following equations.

$$\mathsf{A}(J_i^v, t, t', \mathcal{H}) = \sum_j \mathsf{A}(J_{i,j}^v, t, t', \mathcal{H}) \tag{4.6}$$

$$\mathsf{A}(\Psi, t, t', \mathcal{H}) = \sum_{v,i} \mathsf{A}(J_i^v, t, t', \mathcal{H}) \tag{4.7}$$

Since the processor capacity allocated to a job or server job over an interval is non-negative, for any intervals $[t, t')$ and $[t, t'')$ with $t'' \geq t' \geq t$, we have the following properties.

$$\mathsf{A}(J_{i,j}^v, t, t', \mathcal{H}) \leq \mathsf{A}(J_{i,j}^v, t, t'', \mathcal{H}) \tag{4.8}$$

$$\mathsf{A}(J_{i,j}^v, t, t', \mathcal{H}) = \mathsf{A}(J_{i,j}^v, 0, t', \mathcal{H}) - \mathsf{A}(J_{i,j}^v, 0, t, \mathcal{H}) \tag{4.9}$$

In $\mathcal{I}$, server job $S_{i,j}^v$ and job $\tau_{i,j}^v$ are scheduled when G_j^v is released, *i.e.,* at time $r(\tau_{1,j}^v)$. $S_{i,j}^v$ (resp., $\tau_{i,j}^v$) consumes budget (resp., executes) at a rate of u_i^v until its completion. Therefore, for any interval $[r(\tau_{1,j}^v), t)$, we have

$$\mathsf{A}(J_{i,j}^v, r(\tau_{1,j}^v), t, \mathcal{I}) = \min\{u_i^v(t - r(\tau_{1,j}^v)), C_i^v\}. \tag{4.10}$$

Similarly, for interval $[t, t')$ with $t \geq \Phi^v$ (resp., $t \geq \Phi_{max}$).

$$\mathsf{A}(J_i^v, t, t', \mathcal{I}) = u_i^v(t' - t) \tag{4.11}$$

$$\mathsf{A}(\Psi, t, t', \mathcal{I}) = U_{tot}(t' - t) \tag{4.12}$$

lag and LAG. The lag of job $J_{i,j}^v$ in schedule $\mathcal{H}$ is defined as

$$\mathsf{lag}(J_{i,j}^v, t, \mathcal{H}) = \mathsf{A}(J_{i,j}^v, 0, t, \mathcal{I}) - \mathsf{A}(J_{i,j}^v, 0, t, \mathcal{H}). \tag{4.13}$$

The lag (resp., LAG) of task J_i^v (resp., system Ψ) at time t in $\mathcal{H}$ is given by

$$\begin{aligned} \mathsf{lag}(J_i^v, t, \mathcal{H}) &= \sum_j \mathsf{lag}(J_{i,j}^v, t, \mathcal{H}) \\ &= \mathsf{A}(J_i^v, 0, t, \mathcal{I}) - \mathsf{A}(J_i^v, 0, t, \mathcal{H}) \end{aligned} \tag{4.14}$$

$$\mathsf{LAG}(\Psi, t, \mathcal{H}) = \sum_{v,i} \mathsf{lag}(J_i^v, t, \mathcal{H})$$

$$= \mathsf{A}(\Psi, 0, t, \mathcal{I}) - \mathsf{A}(\Psi, 0, t, \mathcal{H}) \tag{4.15}$$

Since $\mathsf{lag}(J_i^v, 0, \mathcal{H}) = 0$ and $\mathsf{LAG}(\Psi, 0, \mathcal{H}) = 0$, for $t' \geq t$ we have the following equations.

$$\begin{aligned}\mathsf{lag}(J_i^v, t', \mathcal{H}) = \mathsf{lag}(J_i^v, t, \mathcal{H}) + \mathsf{A}(J_i^v, t, t', \mathcal{I}) \\ - \mathsf{A}(J_i^v, t, t', \mathcal{H})\end{aligned} \tag{4.16}$$

$$\mathsf{LAG}(\Psi, t', \mathcal{H}) = \mathsf{LAG}(\Psi, t, \mathcal{H}) + \mathsf{A}(\Psi, t, t', \mathcal{I}) - \mathsf{A}(\Psi, t, t', \mathcal{H}) \tag{4.17}$$

Definition 4.3. Let $h^v = H/T^v$. ◀

Proof overview. Similar to the method we gave to compute exact response-time bounds of sequential tasks in Chapter 3, we aim to derive an upper bound on the length of the prefix of a schedule of DAG tasks after which the response times of DAG tasks do not increase (Theorem 4.2). We do so by showing that if the LAG of Γ remains the same at hyperperiod boundaries for a sufficiently long interval of time, then it continues to remain the same at hyperperiod boundaries at any time in the future (Lemma 4.50). Furthermore, when this happens, response times do not increase afterwards (Lemma 4.51). Our proof is based on the following four key steps, among which the first two pertains to the server schedule and the remaining to the DAG schedule.

Step 1. The amount of time a server job $S_{i,j+h^v}^v$ is scheduled by time $t + H$ is at most the amount of time $S_{i,j}^v$ is scheduled by time t (Lemma 4.19). The amount of time S_i^v (resp., Γ_s) is scheduled over an an interval $[t, t + H)$ is at most Hu_i^v (resp., HU_{tot}) (Lemmas 4.25 and 4.26). This step essentially establishes the lag-monotonicity and LAG-monotonicity properties for servers.

Step 2. If the time allocated to Γ_s over an interval $[t, t + H)$ equals HU_{tot}, then scheduling decisions and the state of Γ_s are identical at times t and $t + H$ (Lemma 4.33).

Step 3. If the time allocated to Γ over H-sized intervals remains HU_{tot} for sufficiently long, then Γ continues to be scheduled for HU_{tot} time units in any future H-sized interval (Lemma 4.50) and each DAG task's maximum response time has stabilized (Lemma 4.51).

Step 4. There is a time instant when the condition mentioned in Step 3 holds (Lemma 4.54).

We cover Steps 1 and 2 in Section 4.4.2 and Steps 3 and 4 in Section 4.4.3.

4.4.2 Analysis of Servers

We begin by addressing Steps 1 and 2. As mentioned earlier, Step 1 establishes the monotonicity properties of lag and LAG. However, as multiple jobs per task can be ready concurrently, we need job-level reasoning, in contrast to the task-level reasoning we used in Chapter 3 where at most one job per task can be ready concurrently.

To complete Step 1, we will first show, in Lemma 4.19, that the amount of time allocated to a server job $S^v_{i,j+h^v}$ by time $t + H$ is at most the amount of time allocated to $S^v_{i,j}$ by time t. Note that server jobs $S^v_{i,j}$ and $S^v_{i,j+h^v}$ are separated by a hyperperiod. We will prove the existence of higher-priority ready jobs that cause $S^v_{i,j+h^v}$ to maintain this property under GEL scheduling. We begin by proving Lemmas 4.11–4.17, which establish the existence of such ready higher-priority jobs. The lemma below holds as servers release jobs periodically.

Lemma 4.11. *For any integer c, $r(S^v_{i,j+ch^v}) = r(S^v_{i,j}) + cH$ and $y(S^v_{i,j+ch^v}) = y(S^v_{i,j}) + cH$ hold.*

The following two lemmas show that the pending jobs of a task are consecutive at any time instant. Informally, this is because, by Lemma 4.2, a server job cannot finish before a prior server job of the same server finishes.

Lemma 4.12. *If $S^v_{i,j}$ and $S^v_{i,\ell}$ with $j \leq \ell$ are pending at time t, then each server job $S^v_{i,k}$ with $j \leq k \leq \ell$ is pending at time t.*

Proof. By Definition 4.1, we have $r(S^v_{i,\ell}) \leq t$. Since $k \leq \ell$, we have $r(S^v_{i,k}) \leq r(S^v_{i,\ell}) \leq t$. By Definition 4.1, we have $f(S^v_{i,j}) > t$. Since $k \geq j$, by Lemma 4.2, we have $f(S^v_{i,k}) \geq f(S^v_{i,j}) > t$. Thus, by Definition 4.1, $S^v_{i,k}$ is pending at time t. □

By Lemma 4.12, we have the following lemma.

Lemma 4.13. *If at least p server jobs of S^v_i are pending at time t and $S^v_{i,j}$ is the highest-priority server job of S^v_i that is pending at time t, then $S^v_{i,j+1}, S^v_{i,j+2}, \ldots, S^v_{i,j+p-1}$ are pending at time t.*

Proof. Assume that $S^v_{i,j+k}$ such that $1 \leq k \leq p-1$ is not pending at time t. Since $S^v_{i,j}$ is the highest-priority pending server job of S^v_i at time t, no server job of S^v_i prior to $S^v_{i,j}$ is pending at time t. Since S^v_i has at least p pending server jobs at time t, there must be a pending server job $S^v_{i,j+\ell}$ at time t such that $\ell > p-1$ holds. By Lemma 4.12, each server job $S^v_{i,j+k}$ with $1 \leq k \leq \ell$ is pending at time t, a contradiction. □

Using the above lemmas about pending server jobs, we now give the following lemmas about ready server jobs. We will use them to show the existence of higher-priority ready jobs at certain time instances, which subsequently will be needed to prove Lemma 4.19.

Lemma 4.14. *If at least p server jobs of S_i^v are ready at time t and $S_{i,j}^v$ is the highest-priority server job of S_i^v that is pending at time t, then $S_{i,j}^v, S_{i,j+1}^v, \ldots, S_{i,j+p-1}^v$ are ready at time t.*

Proof. Since $P_i^v \geq 1$ and no server job of S_i^v with higher priority than $S_{i,j}^v$ is pending, $S_{i,j}^v$ is ready at time t. By Lemma 4.13, server jobs $S_{i,j+1}^v, S_{i,j+2}^v, \ldots, S_{i,j+p-1}^v$ are pending at time t. Assume that $S_{i,j+k}^v$ such that $1 \leq k \leq p-1$ is not ready at time t. Then, $j+k > P_i^v$, and $S_{i,j+k-P_i^v}^v$ does not finish execution by time t, *i.e.,* $f(S_{i,j+k-P_i^v}^v) > t$. As there are at least p ready jobs at time t, there is a ready job $S_{i,j+\ell}^v$ at time t such that $\ell > k$ (thus, $j+\ell > P_i^v$). By Lemma 4.2, $f(S_{i,j+\ell-P_i^v}^v) \geq f(S_{i,j+k-P_i^v}^v) > t$. Thus, $S_{i,j+\ell}^v$ is not ready at time t, a contradiction. □

Lemma 4.15. *If at least p server jobs of S_i^v are pending at time t, then at least $\min\{p, P_i^v\}$ server jobs are ready at time t.*

Proof. If $p = 0$, then the lemma holds trivially, so assume $p \geq 1$. Let $S_{i,j}^v$ be the highest-priority server job of S_i^v that is pending at time t. By Lemma 4.13, $S_{i,j+1}^v, S_{i,j+2}^v, \ldots, S_{i,j+p-1}^v$ are pending at time t. Let $\ell = \min\{p, P_i^v\}$. Then, $\ell \leq P_i^v$ holds. For any $0 \leq k \leq \ell-1$, $j+k-P_i^v \leq j+k-\ell < j$ holds. Therefore, for any $0 \leq k \leq \ell-1$ and $j+k-P_i^v > 0$, $S_{i,j+k-P_i^v}^v$ is complete at time t. By Definition 4.1, for each $0 \leq k \leq \ell-1$, $S_{i,j+k}^v$ is ready at time t and the lemma holds. □

Lemma 4.16. *For any integer c and job index j such that $j + ch^v \geq 1$, if $S_{i,j}^v$ is pending at time t and $\mathsf{A}(S_{i,j}^v, 0, t, \mathcal{S}) \geq \mathsf{A}(S_{i,j+ch^v}^v, 0, t+cH, \mathcal{S})$ holds, then $S_{i,j+ch^v}^v$ is pending at time $t+cH$.*

Proof. By Definition 4.1, $r(\tau_{i,j}^v) \leq t$. By Lemma 4.11, $r(S_{i,j+ch^v}^v) = r(S_{i,j}^v) + cH \leq t + cH$. Thus, $S_{i,j+ch^v}^v$ is released at or before time $t+cH$. Since $S_{i,j}^v$ is ready at time t, by Definition 4.1, $f(S_{i,j}^v) > t$. Thus, $\mathsf{A}(S_{i,j}^v, 0, t, \mathcal{S}) < C_i^v$. Hence, $\mathsf{A}(S_{i,j+ch^v}^v, 0, t+cH, \mathcal{S}) \leq \mathsf{A}(S_{i,j}^v, 0, t, \mathcal{S}) < C_i^v$. Thus, $S_{i,j}^v$ completes execution after time $t+cH$. Therefore, by Definition 4.1, it is pending at time $t+cH$. □

Definition 4.4. Let $hp(S_{k,\ell}^w, S_i^v, t)$ denote the number of ready jobs of S_i^v at time t that have higher priorities than $S_{k,\ell}^w$. ◀

Using Lemmas 4.11–4.16, the following lemma shows that the number of ready jobs at time $t + H$ with higher priorities than $S^w_{k,\ell+h^w}$ is no smaller than the number of ready jobs at time t with higher priorities than $S^w_{k,\ell}$, when $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t + H, \mathcal{S})$ holds for each server job. Informally, this is because, for any ready server job at time t, there exists a unique ready server job at time $t + H$.

Lemma 4.17. *Assume that, for each server job $S^v_{i,j}$ of a server S^v_i, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t + H, \mathcal{S})$ holds at time $t \geq \Phi^v$. Then, for any server job $S^w_{k,\ell}$, $hp(S^w_{k,\ell}, S^v_i, t) \leq hp(S^w_{k,\ell+h^w}, S^v_i, t + H)$ holds.*

Proof. Let $p = hp(S^w_{k,\ell}, S^v_i, t)$. If $p = 0$, then the lemma trivially holds, so assume $p \geq 1$. By Definitions 4.1 and 4.4, $p \leq P^v_i$. Let $S^v_{i,j}$ be the highest-priority server job of S^v_i that is pending at time t. By Lemma 4.14, $S^v_{i,j}, S^v_{i,j+1}, \ldots, S^v_{i,j+p-1}$ are ready at time t. Therefore, by Lemma 4.16 (replacing c by 1), server jobs $S^v_{i,j+h^v}, S^v_{i,j+1+h^v}, \ldots, S^v_{i,j+p-1+h^v}$ are pending at time $t + H$. Let $S^v_{i,x}$ be the highest-priority server job of S^v_i that is pending at time $t + H$. Then, $x \leq j + h^v$ holds. By Lemma 4.12, each server job $S^v_{i,z}$ such that $x \leq z \leq j+p-1+h^v$ is pending at time $t+H$. Therefore, there are at least $j+p-1+h^v-x+1 = j+p+h^v-x$ pending server jobs of S^v_i at time $t + H$. By Lemma 4.15, at least $\min\{j + p + h^v - x, P^v_i\}$ server jobs of S^v_i are ready at time $t + H$. Since $x \leq j + h^v$, we have $j + p + h^v - x \geq p$. Since by Definitions 4.1 and 4.4, $p \leq P^v_i$ holds, we have $p \leq \min\{j + p + h^v - x, P^v_i\}$. Thus, there are at least p ready jobs of S^v_i at time $t + H$. By Lemma 4.14, $S^v_{i,x}, S^v_{i,x+1}, \ldots, S^v_{i,x+p-1}$ are ready at time t.

We now prove that each server job $S^v_{i,x+b}$ with $0 \leq b \leq p - 1$ has higher priority than $S^w_{k,\ell+h^w}$. Since $x \leq j + h^v$, we have $x + p - 1 \leq j + p - 1 + h^v$. Thus, each $S^v_{i,x+b}$ with $0 \leq b \leq p - 1$ has higher or equal priority than $S^v_{i,j+p-1+h^v}$. So, it suffices to prove that $S^v_{i,j+p-1+h^v}$ has higher priority than $S^w_{k,\ell+h^w}$. As $S^v_{i,j}$ is the highest-priority ready job of S^v_i at t, by Definition 4.4, each of $S^v_{i,j}, S^v_{i,j+1}, \ldots, S^v_{i,j+p-1}$ has higher priority than $S^w_{k,\ell}$. Thus, $y(S^v_{i,j+p-1}) \leq y(S^w_{k,\ell})$ holds, and by Lemma 4.11,

$$\begin{aligned} y(S^w_{k,\ell+h^w}) &= y(S^w_{k,\ell}) + H \\ &\geq \{\text{Since } y(S^v_{i,j+p-1}) \leq y(S^w_{k,\ell})\} \\ &\quad y(S^v_{i,j+p-1}) + H \\ &= \{\text{By Lemma 4.11}\} \\ &\quad y(S^v_{i,j+p-1+h^v}). \end{aligned} \tag{4.18}$$

Since ties are broken consistently, $S^v_{i,j+p-1+h^v}$ has higher priority than $S^w_{k,\ell+h^w}$ if $S^v_{i,j+p-1}$ has higher priority than $S^w_{k,\ell}$. Thus, there are at least p ready jobs of S^v_i at time $t+H$ that have higher priorities than $S^w_{k,\ell+h^w}$. □

The following lemma gives necessary conditions for $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S})$ to not hold for the first time in $\mathcal{S}$. The lemma and its proof is similar to Lemma 3.12 proved in Chapter 3. Informally, Lemma 4.18(a) holds as, by Definition 4.3 and Lemma 4.11, $S^v_{i,j+h^v}$ is released after time $\Phi^v + H$. By the definition of time t, $\mathsf{A}(S^v_{i,j}, 0, t-1, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t+H-1, \mathcal{S})$ holds. Thus, Lemma 4.18(b) must hold to satisfy the lemma assumptions.

Lemma 4.18. *Let $t \geq \Phi^v$ be the first time instant (if any) such that for a job $S^v_{i,j}$, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) < \mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S})$ holds. Then, the following hold.*

(a) $t > \Phi^v$.

(b) $S^v_{i,j}$ is not scheduled during $[t-1, t)$, but $S^v_{i,j+h^v}$ is scheduled during $[t+H-1, t+H)$.

Proof. **(a)** Assume that $t = \Phi^v$ holds. Since $r(S^v_{i,1}) = \Phi^v$, by Lemma 4.11, S^v_{i,h^v+1} is released at time $t+H$. Since no server job is scheduled before its release, for any $k \geq 1$, $\mathsf{A}(S^v_{i,k+h^v}, 0, t+H, \mathcal{S}) = 0$ holds. Thus, for any $k \geq 1$, $\mathsf{A}(S^v_{i,k}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,k+h^v}, 0, t+H, \mathcal{S})$, a contradiction.
(b) By Lemma 4.18(a), $t > \Phi^v$ holds. By the definition of t,

$$\mathsf{A}(S^v_{i,j}, 0, t-1, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t+H-1, \mathcal{S}). \tag{4.19}$$

Assume that $S^v_{i,j}$ is scheduled during $[t-1, t)$ or $S^v_{i,j+h^v}$ is not scheduled during $[t+H-1, t+H)$. Then, one of the following three cases holds.

Case 1. $S^v_{i,j}$ and $S^v_{i,j+h^v}$ are scheduled during $[t-1, t)$ and $[t+H-1, t+H)$, respectively. Therefore, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) = \mathsf{A}(S^v_{i,j}, 0, t-1, \mathcal{S}) + 1$ and $\mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S}) = \mathsf{A}(S^v_{i,j+h^v}, 0, t+H-1, \mathcal{S}) + 1$ hold. By (4.19), $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S})$ holds

Case 2. $S^v_{i,j}$ and $S^v_{i,j+h^v}$ are not scheduled during $[t-1, t)$ and $[t+H-1, t+H)$, respectively. Therefore, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) = \mathsf{A}(S^v_{i,j}, 0, t-1, \mathcal{S})$ and $\mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S}) = \mathsf{A}(S^v_{i,j+h^v}, 0, t+H-1, \mathcal{S})$ hold. By (4.19), $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S})$ holds.

Case 3. $S^v_{i,j}$ is scheduled during $[t-1, t)$ and $S^v_{i,j+h^v}$ is not scheduled during $[t+H-1, t+H)$. Therefore, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) = \mathsf{A}(S^v_{i,j}, 0, t-1, \mathcal{S}) + 1$ and $\mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S}) = \mathsf{A}(S^v_{i,j+h^v}, 0, t+H-1, \mathcal{S})$ hold. By (4.19), $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S})$ holds.

In each case, we have a contradiction. □

Using Lemmas 4.17 and 4.18, we now prove the following lemma. This lemma is a "job-level" analogue of the lag-monotonicity property (Lemma 3.13) shown in Chapter 3. Note that we disallowed the concurrent execution of jobs of the same task in Chapter 3. However, with arbitrary parallelism levels, we need to consider multiple ready jobs per task in the proof of the following lemma (hence, we need Lemma 4.17).

Lemma 4.19. *For any server job $S^v_{i,j}$ and time instant $t \geq \Phi^v$, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t + H, \mathcal{S})$ holds.*

Proof. Assume otherwise. Let t be the first time instant such that $t \geq \Phi^v$ and there is a job $S^v_{i,j}$ satisfying the following.

$$\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) < \mathsf{A}(S^v_{i,j+h^v}, 0, t + H, \mathcal{S}) \tag{4.20}$$

By Lemma 4.18(a), $t > \Phi^v$ holds. Thus, by the definition of time t,

$$\forall w, k, \ell : t - 1 \geq \Phi^w :: \mathsf{A}(S^w_{k,\ell}, 0, t - 1, \mathcal{S}) \geq \mathsf{A}(S^w_{k,\ell+h^w}, 0, t + H - 1, \mathcal{S}). \tag{4.21}$$

Since $S^v_{i,j+h^v}$ is scheduled for C^v_i time units in total, we have

$$\mathsf{A}(S^v_{i,j+h^v}, 0, t + H, \mathcal{S}) \leq C^v_i. \tag{4.22}$$

By Lemma 4.18(b), $S^v_{i,j+h^v}$ is scheduled during $[t + H - 1, t + H)$, so $r(S^v_{i,j+h^v}) \leq t + H - 1$ holds, and by Lemma 4.11,

$$r(S^v_{i,j}) = r(S^v_{i,j+h^v}) - H \leq t - 1. \tag{4.23}$$

We now prove two claims.

Claim 4.3. *$S^v_{i,j}$ is pending at time $t - 1$.*

Proof. By (4.23), $S^v_{i,j}$ is released at or before time $t - 1$. Thus, it suffices to prove that $f(S^v_{i,j}) > t - 1$. Assume to the contrary that $f(S^v_{i,j}) \leq t - 1$. Thus, $\mathsf{A}(S^v_{i,j}, 0, t - 1, \mathcal{S}) = C^v_i$. By (4.8), we have $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq C^v_i$. By (4.20), $\mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S}) > \mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq C^v_i$, contradicting (4.22). □

Claim 4.4. *$S^v_{i,j}$ is ready at time $t - 1$.*

Proof. By Claim 4.3, $S^v_{i,j}$ is pending at time $t-1$. If $j \leq P^v_i$, then $S^v_{i,j}$ is also ready at time $t-1$, so assume $j > P^v_i$. Since $S^v_{i,j+h^v}$ is scheduled (hence, ready) at time $t+H-1$, $S^v_{i,j+h^v-P^v_i}$ completes by time $t+H-1$. Hence, $\mathsf{A}(S^v_{i,j+h^v-P^v_i}, 0, t+H-1, \mathcal{S}) = C^v_i$. By (4.21), we have $\mathsf{A}(S^v_{i,j-P^v_i}, 0, t-1, \mathcal{S}) \geq C^v_i$. Thus, $S^v_{i,j-P^v_i}$ completes by time $t-1$ and $S^v_{i,j}$ is ready at time $t-1$. □

By Claim 4.4 and Lemma 4.18(b), $S^v_{i,j}$ is ready but not scheduled at time $t-1$. Therefore, at time $t-1$, there are at least M ready server jobs that have higher priorities than $S^v_{i,j}$. By (4.21) and Lemma 4.17, each server S^w_k with p ready server jobs of higher priority than $S^v_{i,j}$ at time $t-1$ has at least p ready server jobs of higher priority than $S^v_{i,j+h^v}$ at time $t+H-1$. Thus, there are at least M ready server jobs of higher priority than $S^v_{i,j+h^v}$ at time $t+H-1$. Therefore, $S^v_{i,j+h^v}$ cannot be scheduled at time $t+H-1$, which contradicts Lemma 4.18(b). □

The following lemma generalizes Lemma 4.19 for the case of multiple hyperperiods.

Lemma 4.20. *For any server job $S^v_{i,j}$, positive integer c, and time instant $t \geq \Phi^v$, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S})$.*

Proof. By Lemma 4.19, for any $k \geq 0$, we have $\mathsf{A}(S^v_{i,j}, 0, t+kH, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+h^v}, 0, t+(k+1)H, \mathcal{S})$. Therefore, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S})$. □

We now give Lemmas 4.21–4.26, which complete Step 1. Since server job $S^v_{i,j}$ consumes budget at the rate of u^v_i in $\mathcal{I}$ during $[r(S^v_{i,j}), r(S^v_{i,j}) + T^v)$, we have the following lemma.

Lemma 4.21. *For any job $S^v_{i,j}$, positive integer c, and time instant $t \geq \Phi^v$, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{I}) = \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{I})$.*

Proof. We consider two cases.

Case 1. $t < r(S^v_{i,j})$. By Lemma 4.11, $t + cH < r(S^v_{i,j+ch^v})$. In $\mathcal{I}$, $S^v_{i,j}$ and $S^v_{i,j+ch^v}$ do not consume budget before $r(S^v_{i,j})$ and $r(S^v_{i,j+ch^v})$, respectively. Thus, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{I}) = \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{I}) = 0$ holds.

Case 2. $t \geq r(S^v_{i,j})$. Since $S^v_{i,j}$ is released at time $r(S^v_{i,j})$,

$$\begin{aligned}\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{I}) &= \mathsf{A}(S^v_{i,j}, r(S^v_{i,j}), t, \mathcal{I}) \\ &= \{\text{By (4.10)}\}\end{aligned}$$

$$\min\{u_i^v(t - r(S_{i,j}^v)), C_i^v\}. \tag{4.24}$$

Similarly, for $S_{i,j+h^v}^v$, we have

$$\begin{aligned}
\mathsf{A}(S_{i,j+h^v}^v, 0, t+cH, \mathcal{I}) &= \min\{u_i^v(t+cH-r(S_{i,j+ch^v}^v)), C_i^v\} \\
&= \{\text{By Lemma 4.11}\} \\
&\quad \min\{u_i^v(t - r(S_{i,j}^v)), C_i^v\} \\
&= \{\text{By (4.24)}\} \\
&\quad \mathsf{A}(S_{i,j}^v, 0, t, \mathcal{I}).
\end{aligned}$$

Thus, the lemma holds. □

Lemma 4.22. *For any server job $S_{i,j}^v$, positive integer c, and time instant $t \geq \Phi^v$, $\mathsf{lag}(S_{i,j}^v, t, \mathcal{S}) \leq \mathsf{lag}(S_{i,j+ch^v}^v, t + cH, \mathcal{S})$.*

Proof. By Lemmas 4.20 and 4.21, $\mathsf{A}(S_{i,j}^v, 0, t, \mathcal{I}) - \mathsf{A}(S_{i,j}^v, 0, t, \mathcal{S}) \leq \mathsf{A}(S_{i,j+ch^v}^v, 0, t+cH, \mathcal{I}) - \mathsf{A}(S_{i,j+ch^v}^v, 0, t + cH, \mathcal{S})$. Thus, by (4.13), the lemma holds. □

In $\mathcal{I}$, each server job completes execution when the next server job is released. Thus, a server job's lag cannot be negative at or after its next server job is released. This is shown in the following lemma.

Lemma 4.23. *For any server job $S_{i,j}^v$ and time instant $t \geq r(S_{i,j}^v) + T^v$, $\mathsf{lag}(S_{i,j}^v, t, \mathcal{S}) \geq 0$ holds.*

Proof. Since $S_{i,j}^v$ is scheduled after its release in $\mathcal{I}$, we have

$$\begin{aligned}
\mathsf{A}(S_{i,j}^v, 0, t, \mathcal{I}) &= \mathsf{A}(S_{i,j}^v, r(S_{i,j}^v), t, \mathcal{I}) \\
&= \{\text{By (4.10)}\} \\
&\quad \min\{u_i^v(t - r(S_{i,j}^v)), C_i^v\} \\
&= \{\text{Since } t \geq r(S_{i,j}^v) + T^v \text{ and } u_i^v T^v = C_i^v\} \\
&\quad C_i^v.
\end{aligned}$$

Since $S_{i,j}^v$ does not consume budget more than C_i^v in $\mathcal{S}$, we have $\mathsf{A}(S_i^v j, 0, t, \mathcal{S}) \leq C_i^v$. Thus, we have $\mathsf{lag}(S_{i,j}^v, t, \mathcal{S}) = \mathsf{A}(S_{i,j}^v, 0, t, \mathcal{I}) - \mathsf{A}(S_{i,j}^v, 0, t, \mathcal{S}) \geq C_i^v - C_i^v \geq 0$. □

Since each server job $S^v_{i,j}$ completes at time $r(S^v_{i,j}) + T^v = r(S^v_{i,j+1})$ in $\mathcal{I}$, and S^v_{i,h^v+1} is released at time $\Phi^v + H$, each server job $S^v_{i,j}$ with $j \leq h^v$ is complete by time $\Phi^v + H$ in $\mathcal{I}$. Using this fact, we have the following lemma.

Lemma 4.24. *For any integer $c \geq 1$, server job $S^v_{i,j}$ with $1 \leq j \leq ch^v$, and time $t \geq \Phi^v + cH$, $\mathsf{lag}(S^v_{i,j}, t, \mathcal{S}) \geq 0$ holds.*

Proof. Since S^v_i releases periodically, we have $r(S^v_{i,j}) + T^v = \Phi^v + (j-1)T^v + T^v = \Phi^v + jT^v$. Since $j \leq ch^v$, we have $r(S^v_{i,j}) + T^v = \Phi^v + jT^v \leq \Phi^v + ch^v T^v = \Phi^v + cH$. Thus, $t \geq r(S^v_{i,j}) + T^v$ holds. By Lemma 4.23, $\mathsf{lag}(S^v_{i,j}, t, \mathcal{S}) \geq 0$. □

We now prove lag-monotonicity for servers using server jobs' lag-monotonicity.

Lemma 4.25. *For any server S^v_i, positive integer c, and time instant $t \geq \Phi^v$, the following hold.*

(a) $\mathsf{lag}(S^v_i, t, \mathcal{S}) \leq \mathsf{lag}(S^v_i, t + cH, \mathcal{S})$.

(b) $\mathsf{A}(S^v_i, t, t + cH, \mathcal{S}) \leq cHu^v_i$.

Proof. **(a)** By (4.14), we have

$$\begin{aligned}
\mathsf{lag}(S^v_i, t, \mathcal{S}) &= \sum_{j > ch^v} \mathsf{lag}(S^v_{i,j-ch^v}, t, \mathcal{S}) \\
&\leq \{\text{By Lemma 4.22}\} \\
&\quad \sum_{j > ch^v} \mathsf{lag}(S^v_{i,j}, t + cH, \mathcal{S}) \\
&\leq \{\text{By Lemma 4.24}\} \\
&\quad \sum_{1 \leq j \leq ch^v} \mathsf{lag}(S^v_{i,j}, t + cH, \mathcal{S}) + \sum_{j > ch^v} \mathsf{lag}(S^v_{i,j}, t + cH, \mathcal{S}) \\
&= \{\text{By (4.14)}\} \\
&\quad \mathsf{lag}(S^v_i, t + cH, \mathcal{S}).
\end{aligned}$$

(b) Assume that for S^v_i and time instant $t \geq \Phi^v$, $\mathsf{A}(S^v_i, t, t + cH, \mathcal{S}) > cHu^v_i$ holds. Then, by (4.11), we have $\mathsf{A}(S^v_i, t, t + cH, \mathcal{I}) - \mathsf{A}(S^v_i, t, t + cH, \mathcal{S}) < cHu^v_i - cHu^v_i = 0$. Thus, by (4.16), $\mathsf{lag}(S^v_i, t + cH, \mathcal{S}) < \mathsf{lag}(S^v_i, t, \mathcal{S})$, which contradicts Lemma 4.25(a). □

Finally, LAG-monotonicity is shown in the following lemma.

Lemma 4.26. *For any integer c and time instant $t \geq \Phi_{max}$,*

(a) $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) \leq \mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S})$,

(b) $\mathsf{A}(\Gamma_s, t, t + cH, \mathcal{S}) \leq cHU_{tot}$.

Proof. **(a)** By Lemma 4.25(a), we have $\sum_{v,i} \mathsf{lag}(S_i^v, t, \mathcal{S}) \leq \sum_{v,i} \mathsf{lag}(S_i^v, t + cH, \mathcal{S})$. Thus, by (4.15), the lemma holds.

(b) By Lemma 4.25(b), we have $\sum_{v,i} \mathsf{A}(S_i^v, t, t+cH, \mathcal{S}) \leq \sum_{v,i} cHu_i^v$. Since $\sum_{v,i} u_i^v = U_{tot}$, by (4.7), we have $\mathsf{A}(\Gamma_s, t, t + cH, \mathcal{S}) \leq cHU_{tot}$. □

We now address Step 2. We will show, in Lemma 4.33, that if $\mathsf{A}(\Gamma_s, t, t + H, \mathcal{S}) = HU_{tot}$ holds, then scheduling decisions in $\mathcal{S}$ are identical at times t and $t + H$. We begin by proving some lemmas (Lemmas 4.27–4.30) that establish server- and server-job-level properties at times t and $t + cH$, when Γ_s's LAG values are equal. The next lemma follows from (4.17) and (4.12).

Lemma 4.27. *For any positive integer c and time instant $t \geq \Phi_{max}$, $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S})$ holds if and only if $\mathsf{A}(\Gamma_s, t, t + cH, \mathcal{S}) = cHU_{tot}$.*

Proof. By (4.12), we have $\mathsf{A}(\Gamma_s, t, t + cH, \mathcal{I}) = cHU_{tot}$. By (4.17), we have $\mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t, \mathcal{S}) + \mathsf{A}(\Gamma_s, t, t + cH, \mathcal{I}) - \mathsf{A}(\Gamma_s, t, t + cH, \mathcal{S})$. Thus, if $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S})$ holds, then we have $\mathsf{A}(\Gamma_s, t, t+cH, \mathcal{S}) = \mathsf{A}(\Gamma_s, t, t+cH, \mathcal{I}) = cHU_{tot}$. Similarly, if $\mathsf{A}(\Gamma_s, t, t+cH, \mathcal{S}) = cHU_{tot} = \mathsf{A}(\Gamma_s, t, t + cH, \mathcal{I})$, then $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S})$. □

By Lemma 4.25(a), we have the following lemma. This lemma is similar to Lemma 3.18 shown in Chapter 3.

Lemma 4.28. *For any positive integer c and time instant $t \geq \Phi_{max}$, if $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S})$ holds, then for any S_i^v, $\mathsf{lag}(S_i^v, t, \mathcal{S}) = \mathsf{lag}(S_i^v, t + cH, \mathcal{S})$ holds.*

Proof. Assume there is a server S_i^v, integer $c > 0$, and time $t \geq \Phi_{max}$ such that $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t+ cH, \mathcal{S})$ and $\mathsf{lag}(S_i^v, t, \mathcal{S}) \neq \mathsf{lag}(S_i^v, t + cH, \mathcal{S})$. By Lemma 4.25(a), $\mathsf{lag}(S_i^v, t, \mathcal{S}) < \mathsf{lag}(S_i^v, t + cH, \mathcal{S})$. Therefore, by (4.15),

$$\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \sum_{w,k} \mathsf{lag}(S_k^w, t, \mathcal{S})$$

$$
\begin{aligned}
&= \mathsf{lag}(S_i^v, t, \mathcal{S}) + \sum_{w,k:(w,v)\neq(k,i)} \mathsf{lag}(S_k^w, t, \mathcal{S}) \\
&< \{\text{By Lemma 4.25(a) and since } \mathsf{lag}(S_i^v, t, \mathcal{S}) < \mathsf{lag}(S_i^v, t + cH, \mathcal{S})\} \\
&\quad \mathsf{lag}(S_i^v, t + cH, \mathcal{S}) + \sum_{w,k:(w,v)\neq(k,i)} \mathsf{lag}(S_k^w, t + cH, \mathcal{S}) \\
&= \{\text{By (4.15)}\} \\
&\quad \mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S}),
\end{aligned}
$$

a contradiction. □

Similarly, by Lemma 4.22, we have the following lemma.

Lemma 4.29. *For any positive integer c and time instant $t \geq \Phi_{max}$, if $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S})$ holds, then for any server job $S_{i,j}^v$, the following hold.*

(a) $\mathsf{lag}(S_{i,j}^v, t, \mathcal{S}) = \mathsf{lag}(S_{i,j+ch^v}^v, t + cH, \mathcal{S})$.

(b) $\mathsf{A}(S_{i,j}^v, 0, t, \mathcal{S}) = \mathsf{A}(S_{i,j+ch^v}^v, 0, t + cH, \mathcal{S})$.

Proof. **(a)** Assume that there is a server job $S_{i,j}^v$, positive integer c, and time instant $t \geq \Phi_{max}$ such that $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t + cH, \mathcal{S})$ and $\mathsf{lag}(S_{i,j}^v, t, \mathcal{S}) \neq \mathsf{lag}(S_{i,j+ch^v}^v, t + cH, \mathcal{S})$ hold. By Lemma 4.22, we have $\mathsf{lag}(S_{i,j}^v, t, \mathcal{S}) < \mathsf{lag}(S_{i,j+ch^v}^v, t + cH, \mathcal{S})$. By (4.14), we have

$$
\begin{aligned}
\mathsf{lag}(S_i^v, t, \mathcal{S}) &= \sum_{k\geq 1} \mathsf{lag}(S_{i,k}^v, t, \mathcal{S}) \\
&= \mathsf{lag}(S_{i,j}^v, t, \mathcal{S}) + \sum_{k\geq 1 \wedge k\neq j} \mathsf{lag}(S_{i,k}^v, t, \mathcal{S}) \\
&< \{\text{By Lemma 4.22 and since } \mathsf{lag}(S_{i,j}^v, t, \mathcal{S}) < \mathsf{lag}(S_{i,j+ch^v}^v, t + cH, \mathcal{S})\} \\
&\quad \mathsf{lag}(S_{i,j+ch^v}^v, t + cH, \mathcal{S}) + \sum_{k\geq 1 \wedge k\neq j} \mathsf{lag}(S_{i,k+ch^v}^v, t + cH, \mathcal{S}) \\
&= \sum_{k\geq 1} \mathsf{lag}(S_{i,k+ch^v}^v, t + cH, \mathcal{S}) \\
&= \sum_{k> ch^v} \mathsf{lag}(S_{i,k}^v, t + cH, \mathcal{S}) \\
&\leq \{\text{By Lemma 4.24}\}
\end{aligned}
$$

$$\sum_{1\le k\le ch^v} \mathsf{lag}(S^v_{i,k}, t+cH, \mathcal{S}) + \sum_{k>ch^v} \mathsf{lag}(S^v_{i,k}, t+cH, \mathcal{S})$$
$$= \sum_{k\ge 1} \mathsf{lag}(S^v_{i,k}, t+cH, \mathcal{S})$$
$$= \{\text{By (4.14)}\}$$
$$= \mathsf{lag}(S^v_i, t+cH, \mathcal{S}),$$

which contradicts Lemma 4.28.

(b) Assume there is a server job $S^v_{i,j}$, integer $c > 0$, and time $t \ge \Phi_{max}$ such that $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t+cH, \mathcal{S})$ and $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \neq \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S})$. By Lemma 4.20, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) > \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S})$. By (4.13),

$$\begin{aligned}\mathsf{lag}(S^v_{i,j}, t, \mathcal{S}) &= \mathsf{A}(S^v_{i,j}, 0, t, \mathcal{I}) - \mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \\ &< \{\text{By Lemma 4.21 and since } \mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) > \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S})\} \\ &\quad \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{I}) - \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S}) \\ &= \mathsf{lag}(S^v_{i,j+ch^v}, t+cH, \mathcal{S}),\end{aligned}$$

which contradicts Lemma 4.29(a). □

Lemma 4.30. *Assume that a server S^v_i, job index j, integer c, and time t exist such that $j + ch^v \ge 1$, $\min\{t, t+cH\} \ge \Phi^v$, and $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) = \mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S})$. Then, $f(S^v_{i,j}) \le t$ if and only if $f(S^v_{i,j+ch^v}) \le t + cH$.*

Proof. **Necessity.** Assume that $f(S^v_{i,j}) \le t$ holds. Then, we have $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) = C^v_i$. Therefore, $\mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S}) = C^v_i$ holds. Thus, $f(S^v_{i,j+ch^v}) \le t + cH$.

Sufficiency. Assume that $f(S^v_{i,j}) > t$ holds. Then, we have $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) < C^v_i$. Therefore, $\mathsf{A}(S^v_{i,j+ch^v}, 0, t+cH, \mathcal{S}) < C^v_i$ holds. Thus, $f(S^v_{i,j+ch^v}) > t + cH$. □

Similar to Lemma 4.2, we have the following lemma.

Lemma 4.31. *For any positive integers j and k such that $j \le k$ and time instant t, $\mathsf{A}(S^v_{i,j}, 0, t, \mathcal{S}) \ge \mathsf{A}(S^v_{i,k}, 0, t, \mathcal{S})$ holds.*

If the schedule over the interval $[t, t+H)$ in $\mathcal{S}$ repeats during $[t+H, t+2H)$, then for each server job $S^v_{i,j}$ that is ready (resp., not ready) at time t, $S^v_{i,j+h^v}$ must be ready (resp., not ready) at time $t+H$. The lemma below shows that this condition holds if Γ_s's LAG values at time t and $t+H$ are the same.

The following lemma shows that when LAG becomes equal at hyperperiod boundaries, scheduling decisions are the same at those time instants. Recall that, the same property for $P^v_i = 1$, was proved in Lemma 3.30 for sequential tasks. However, the proof for arbitrary parallelization levels is more involved, as we need to consider multiple server jobs per server.

Lemma 4.32. *If there is a time instant $t \geq \Phi_{max}$ such that $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t+H, \mathcal{S})$ holds, then $S^v_{i,j}$ is ready at time t if and only if $S^v_{i,j+h^v}$ is ready at time $t+H$.*

Proof. By Lemma 4.29, we have

$$\forall v, i, k : \mathsf{A}(S^v_{i,k}, 0, t, \mathcal{S}) = \mathsf{A}(S^v_{i,k+h^v}, 0, t+H, \mathcal{S}). \tag{4.25}$$

Sufficiency. Assume that $S^v_{i,j+h^v}$ is ready at time $t+H$, but $S^v_{i,j}$ is not ready at time t. Since $S^v_{i,j+h^v}$ is ready (hence, pending) at time $t+H$, by (4.25) and Lemma 4.16 (replacing j, t, and c with $j+h^v$, $t+H$, and -1, respectively), $S^v_{i,j}$ is pending at time t. Since $S^v_{i,j}$ is not ready at time t, by Definition 4.1, $j > P^v_i$ and $f(S^v_{i,j-P^v_i}) > t$ hold. By (4.25) and Lemma 4.30, we have $f(S^v_{i,j+h^v-P^v_i}) > t+H$. Thus, by Definition 4.1, $S^v_{i,j+h^v}$ is not ready at time $t+H$, a contradiction.

Necessity. Assume that $S^v_{i,j}$ is ready at time t, but $S^v_{i,j+h^v}$ is not ready at time $t+H$. Since $S^v_{i,j}$ is ready at time t, by (4.25) and Lemma 4.16, $S^v_{i,j+h^v}$ is pending at time t. Since $S^v_{i,j+h^v}$ is not ready at time $t+H$, by Definition 4.1, $j+h^v > P^v_i$ and $f(S^v_{i,j+h^v-P^v_i}) > t+H$ hold. We now consider two cases.

Case 1. $j > P^v_i$. By (4.25) and Lemma 4.30, $f(S^v_{i,j-P^v_i}) > t$. Thus, by Definition 4.1, $S^v_{i,j}$ is not ready at time t. Contradiction.

Case 2. $j \leq P^v_i$. In this case, $j+h^v-P^v_i \leq P^v_i+h^v-P^v_i = h^v$. Since $S^v_{i,j+h^v}$ is pending but not ready at time $t+H$, $\mathsf{A}(S^v_{i,j+h^v}, 0, t+H, \mathcal{S}) = 0$. By Lemma 4.31, for each $b \geq j$, $\mathsf{A}(S^v_{i,b+h^v}, 0, t+H, \mathcal{S}) = 0$ holds. Therefore, we have

$$\begin{aligned}\mathsf{A}(S^v_i, t, t+H, \mathcal{S}) &= \sum_{1 \leq b \leq j+h^v} \mathsf{A}(S^v_{i,b}, t, t+H, \mathcal{S}) \\ &= \{\text{By (4.9)}\}\end{aligned}$$

$$\sum_{1\le b\le j+h^v} (\mathsf{A}(S^v_{i,b}, 0, t+H, \mathcal{S}) - \mathsf{A}(S^v_{i,b}, 0, t, \mathcal{S})). \tag{4.26}$$

By (4.25), $\mathsf{A}(S^v_{i,b}, 0, t, \mathcal{S}) = \mathsf{A}(S^v_{i,b+h^v}, 0, t+H, \mathcal{S})$ for each $1 \le b \le j$. Thus, for each $1 \le b \le j$, applying $\mathsf{A}(S^v_{i,b+h^v}, 0, t+H, \mathcal{S}) - \mathsf{A}(S^v_{i,b}, 0, t, \mathcal{S}) = 0$ in (4.26),

$$\begin{aligned}
\mathsf{A}(S^v_i, t, t+H, \mathcal{S}) &= \sum_{1\le b\le h^v} \mathsf{A}(S^v_{i,b}, 0, t+H, \mathcal{S}) - \sum_{j+1\le b\le j+h^v} \mathsf{A}(S^v_{i,b}, 0, t, \mathcal{S}) \\
&\le \sum_{1\le b\le h^v} \mathsf{A}(S^v_{i,b}, 0, t+H, \mathcal{S}) \\
&= \{ \text{ Since } j + h^v - P^v_i \le h^v \} \\
&\quad \mathsf{A}(S^v_{i,j+h^v-P^v_i}, 0, t+H, \mathcal{S}) + \sum_{\substack{1\le b\le h^v \wedge \\ b\ne j+h^v-P^v_i}} \mathsf{A}(S^v_{i,b}, 0, t+H, \mathcal{S}) \\
&< \{\text{Since } \mathsf{A}(S^v_{i,b}, 0, t+H, \mathcal{S}) \le C^v_i \text{and } S^v_{i,j+h^v-P^v_i} \text{ is pending at time } t+H\} \\
&\quad C^v_i + (h^v - 1)C^v_i \\
&= \{\text{By Definition 4.3}\} \\
&\quad Hu^v_i.
\end{aligned} \tag{4.27}$$

By (4.16), (4.11), and (4.27), we have $\mathsf{lag}(S^v_i, t+H, \mathcal{S}) > \mathsf{lag}(S^v_i, t, \mathcal{S}) + Hu^v_i - Hu^v_i = \mathsf{lag}(S^v_i, t, \mathcal{S})$, which contradicts Lemma 4.28. □

We now complete Step 2 by giving the following lemma. Using Lemma 4.32, we show that if the time allocated to Γ_s over an interval $[t, t+H)$ equals HU_{tot}, then $S^v_{i,j}$ is scheduled at time t if and only if $S^v_{i,j+h^v}$ is scheduled at time $t+H$.

Lemma 4.33. *For any $t \ge \Phi_{max}$, if $\mathsf{A}(\Gamma_s, t, t+H, \mathcal{S}) = HU_{tot}$ holds, then the following hold.*

(a) For any $S^v_{i,j}$, $\mathsf{A}(S^v_{i,j}, t, t+1, \mathcal{S}) = \mathsf{A}(S^v_{i,j+h^v}, t+H, t+H+1, \mathcal{S})$.

(b) For any S^v_i, $\mathsf{A}(S^v_i, t, t+1, \mathcal{S}) = \mathsf{A}(S^v_i, t+H, t+H+1, \mathcal{S})$.

(c) $\mathsf{A}(\Gamma_s, t, t+1, \mathcal{S}) = \mathsf{A}(\Gamma_s, t+H, t+H+1, \mathcal{S})$.

Proof. **(a)** By Lemma 4.27, we have $\mathsf{LAG}(\Gamma_s, t, \mathcal{S}) = \mathsf{LAG}(\Gamma_s, t+H, \mathcal{S})$. Thus, by Lemma 4.32, $S^v_{i,j}$ is ready at time t if and only if $S^v_{i,j+h^v}$ is ready at time $t+H$. By Lemma 4.11, $y(S^v_{i,j+h^v}) = y(S^v_{i,j}) + H$

holds. Thus, for each pair of server jobs $S^v_{i,j}$ and $S^w_{k,\ell}$, we have $y(S^v_{i,j}) - y(S^w_{k,\ell}) = y(S^v_{i,j+h^v}) - y(S^w_{k,\ell+h^w})$. Since ties are broken consistently, $S^v_{i,j}$ has higher priority than $S^w_{k,\ell}$ if and only if $S^v_{i,j+h^v}$ has higher priority than $S^w_{k,\ell+h^w}$. Thus, $S^v_{i,j}$ is the p^{th} highest-priority ready job at time t if and only if $S^v_{i,j+h^v}$ is the p^{th} highest-priority ready job at time $t + H$. Therefore, $S^v_{i,j}$ is scheduled at time t if and only if $S^v_{i,j+h^v}$ is scheduled at time $t + H$. Thus, (a) holds.

(b) Follows from (a) and (4.6).

(c) Follows from (b) and (4.7). □

We also show the following lemma, which will be useful for Step 3.

Lemma 4.34. *For any positive integer c and time $t' \geq \Phi_{max}$, if $\mathsf{A}(\Gamma_s, t', t' + cH, \mathcal{S}) = cHU_{tot}$ holds, then, for each time instant $t \in [t', t' + (c-1)H]$, $\mathsf{A}(\Gamma_s, t, t + H, \mathcal{S}) = HU_{tot}$ holds.*

Proof. We first prove the following claim.

Claim 4.5. $\mathsf{A}(\Gamma_s, t', t' + H, \mathcal{S}) = HU_{tot}$.

Proof. For $c = 1$, the claim holds by the lemma assumptions, so assume $c \geq 2$. Assume for a contradiction that $\mathsf{A}(\Gamma_s, t', t' + H, \mathcal{S}) \neq HU_{tot}$. Then, by Lemma 4.26(b), we have $\mathsf{A}(\Gamma_s, t', t' + H, \mathcal{S}) < HU_{tot}$. Since $[t', t' + cH) = \bigcup_{i=0}^{c-1} [t' + iH, t' + (i+1)H)$, we have

$$
\begin{aligned}
&\mathsf{A}(\Gamma_s, t', t' + cH, \mathcal{S}) \\
&= \mathsf{A}(\Gamma_s, t', t' + H, \mathcal{S}) + \sum_{i=1}^{c-1} \mathsf{A}(\Gamma_s, t' + iH, t' + (i+1)H, \mathcal{S}) \\
&< \{\text{By Lemma 4.26(b) and since } \mathsf{A}(\Gamma_s, t', t' + H, \mathcal{S}) < HU_{tot}\} \\
&\quad HU_{tot} + (c-1)HU_{tot} \\
&= cHU_{tot},
\end{aligned}
$$

a contradiction. □

We now prove the lemma. Assume for a contradiction that time $t \in [t', t' + (c-1)H]$ exists such that $\mathsf{A}(\Gamma_s, t, t + H, \mathcal{S}) \neq HU_{tot}$. By Claim 4.5, $t > t'$. Thus, $\mathsf{A}(\Gamma_s, t - 1, t + H - 1, \mathcal{S}) = HU_{tot}$. Since $[t, t + H) = ([t - 1, t + H - 1) \bigcup [t + H - 1, t + H)) \setminus [t - 1, t)$, we have $\mathsf{A}(\Gamma_s, t, t + H, \mathcal{S}) = \mathsf{A}(\Gamma_s, t -$

$1, t + H - 1, \mathcal{S}) + \mathsf{A}(\Gamma_s, t + H - 1, t + H, \mathcal{S}) - \mathsf{A}(\Gamma_s, t - 1, t, \mathcal{S})$, which by Lemma 4.33(c) equals $\mathsf{A}(\Gamma_s, t - 1, t + H - 1, \mathcal{S}) = HU_{tot}$, a contradiction. □

4.4.3 Analysis of DAG Tasks

We now give an analysis of schedule $\mathcal{G}$ that completes Steps 3 and 4. We begin by showing, in Lemmas 4.35–4.48, that there are properties of lag and LAG in $\mathcal{G}$ that are analogous to the properties in $\mathcal{S}$. Intuitively, these properties hold as a job of Γ can execute only when its linked server job is scheduled.

Lemma 4.35. *If $\tau^v_{i,j}$ is linked to $S^v_{i,k}$, then for any time instant t, $\mathsf{A}(\tau^v_{i,j}, 0, t, \mathcal{G}) = \mathsf{A}(S^v_{i,k}, 0, t, \mathcal{S})$ holds.*

Proof. Follows from the budget Consumption Rule, Assumption 4.1, and Rule R3. □

By Lemmas 4.6, 4.31, and 4.35, we have the following lemma.

Lemma 4.36. *For any positive integers j and k such that $j < k$ and time instant t, $\mathsf{A}(\tau^v_{i,j}, 0, t, \mathcal{G}) \geq \mathsf{A}(\tau^v_{i,k}, 0, t, \mathcal{G})$ holds.*

Proof. Assume $\tau^v_{i,j}$ (resp., $\tau^v_{i,k}$) is linked to $S^v_{i,p}$ (resp., $S^v_{i,q}$). By Lemmas 4.6 and 4.31, $q > p$ and $\mathsf{A}(S^v_{i,p}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,q}, 0, t, \mathcal{S})$. By Lemma 4.35, $\mathsf{A}(\tau^v_{i,j}, 0, t, \mathcal{G}) \geq \mathsf{A}(\tau^v_{i,k}, 0, t, \mathcal{G})$. □

Lemma 4.37. *For any job $\tau^v_{i,j}$, positive integer c, and time instant $t \geq \Phi^v$, $\mathsf{A}(\tau^v_{i,j}, 0, t, \mathcal{G}) \geq \mathsf{A}(\tau^v_{i,j+ch^v}, 0, t + cH, \mathcal{G})$ holds.*

Proof. Assume $\tau^v_{i,j}$ (resp., $\tau^v_{i,j+ch^v}$) is linked to $S^v_{i,k}$ (resp., $S^v_{i,\ell}$). By Lemma 4.20, $\mathsf{A}(S^v_{i,k}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,k+ch^v}, 0, t + cH, \mathcal{S})$. By Lemma 4.6, $\ell \geq k + ch^v$. Thus, by Lemma 4.31, we have $\mathsf{A}(S^v_{i,k+ch^v}, 0, t + cH, \mathcal{S}) \geq \mathsf{A}(S^v_{i,\ell}, 0, t + cH, \mathcal{S})$. Hence, $\mathsf{A}(S^v_{i,k}, 0, t, \mathcal{S}) \geq \mathsf{A}(S^v_{i,\ell}, 0, t + cH, \mathcal{S})$ holds. By Lemma 4.35, $\mathsf{A}(\tau^v_{i,j}, 0, t, \mathcal{G}) \geq \mathsf{A}(\tau^v_{i,j+ch^v}, 0, t + cH, \mathcal{G})$. □

By Lemma 4.37, we can prove Lemmas 4.38–4.48 for Γ, which are analogous to Lemmas 4.21–4.29 and 4.34 for Γ_s. Readers may wish to skip the "identical" proofs.

Lemma 4.38. *For any job $\tau^v_{i,j}$, positive integer c, and time instant $t \geq \Phi^v$, $\mathsf{A}(\tau^v_{i,j}, 0, t, \mathcal{I}) = \mathsf{A}(\tau^v_{i,j+ch^v}, 0, t + cH, \mathcal{I})$ holds.*

Proof. Follows from (4.10) and a proof similar to Lemma 4.21. □

Lemma 4.39. *For any job $\tau_{i,j}^v$, positive integer c, and time instant $t \geq \Phi^v$, $\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) \leq \mathsf{lag}(\tau_{i,j+ch^v}^v, t + cH, \mathcal{G})$ holds.*

Proof. By (4.13), we have

$$\begin{aligned}
\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) &= \mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{I}) - \mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{G}) \\
&\leq \{\text{By Lemmas 4.37 and 4.38}\} \\
&\quad \mathsf{A}(\tau_{i,j+ch^v}^v, 0, t + cH, \mathcal{I}) - \\
&\qquad \mathsf{A}(\tau_{i,j+ch^v}^v, 0, t + cH, \mathcal{G}) \\
&= \{\text{By (4.13)}\} \\
&\quad \mathsf{lag}(\tau_{i,j+ch^v}^v, t + cH, \mathcal{G}).
\end{aligned}$$

Thus, the lemma holds. □

Lemma 4.40. *For any job $\tau_{i,j}^v$ and time instant $t \geq r(\tau_{1,j}^v) + T^v$, $\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) \geq 0$ holds.*

Proof. Since $\tau_{i,j}^v$ is first scheduled at time $r(\tau_{1,j}^v)$ in $\mathcal{I}$, we have

$$\begin{aligned}
\mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{I}) &= \mathsf{A}(\tau_{i,j}^v, r(\tau_{1,j}^v), t, \mathcal{I}) \\
&= \{\text{By (4.10)}\} \\
&\quad \min\{u_i^v(t - r(\tau_{1,j}^v)), C_i^v\} \\
&= \{\text{Since } t \geq r(\tau_{1,j}^v) + T^v \text{ and } u_i^v T^v = C_i^v\} \\
&\quad C_i^v.
\end{aligned}$$

Since $\tau_{i,j}^v$ does not execute more than C_i^v in $\mathcal{G}$, we have $\mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{G}) \leq C_i^v$. Thus, we have $\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) = \mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{I}) - \mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{G}) \geq C_i^v - C_i^v \geq 0$. □

Lemma 4.41. *For any job $\tau_{i,j}^v$ such that $1 \leq j \leq ch^v$ and any time instant $t \geq \Phi^v + cH$, $\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) \geq 0$ holds.*

Proof. Since G^v releases periodically, $r(\tau_{1,j}^v) + T^v = \Phi^v + (j-1)T^v + T^v = \Phi^v + jT^v$. Since $j \leq ch^v$, we have $r(\tau_{1,j}^v) + T^v = \Phi^v + jT^v \leq \Phi^v + ch^v T^v = \Phi^v + cH$. Thus, $t \geq r(\tau_{1,j}^v) + T^v$ holds. By Lemma 4.40, $\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) \geq 0$ holds. □

Lemma 4.42. *For any task τ_i^v, positive integer c, and time instant $t \geq \Phi^v$, $\mathsf{lag}(\tau_i^v, t, \mathcal{G}) \leq \mathsf{lag}(\tau_i^v, t + cH, \mathcal{G})$ holds.*

Proof. Since τ_i^v can only execute when server jobs of S_i^v are scheduled, by Lemma 4.25(b), we have

$$\mathsf{A}(\tau_i^v, t, t + cH, \mathcal{G}) \leq cHu_i^v. \tag{4.28}$$

By (4.16), we have

$$\begin{aligned}
\mathsf{lag}(\tau_i^v, t + cH, \mathcal{G}) &= \mathsf{lag}(\tau_i^v, t, \mathcal{G}) + \mathsf{A}(\tau_i^v, t, t + cH, \mathcal{I}) \\
&\quad - \mathsf{A}(\tau_i^v, t, t + cH, \mathcal{G}) \\
&\geq \{\text{By (4.11) and (4.28)}\} \\
&\quad \mathsf{lag}(\tau_i^v, t, \mathcal{G}) + cHu_i^v - cHu_i^v \\
&= \mathsf{lag}(\tau_i^v, t, \mathcal{G}).
\end{aligned}$$

□

Lemma 4.43. *For any positive integer c and time instant $t \geq \Phi_{max}$, the following hold.*

(a $\mathsf{LAG}(\Gamma, t, \mathcal{G}) \leq \mathsf{LAG}(\Gamma, t + cH, \mathcal{G})$.

(b) $\mathsf{A}(\Gamma, t, t + cH, \mathcal{G}) \leq cHU_{tot}$.

Proof. **(a)** By (4.15), we have

$$\begin{aligned}
\mathsf{LAG}(\Gamma, t, \mathcal{G}) &= \sum_{v,i} \mathsf{lag}(\tau_i^v, t, \mathcal{G}) \\
&\leq \{\text{By Lemma 4.42}\} \\
&\quad \sum_{v,i} \mathsf{lag}(\tau_i^v, t + cH, \mathcal{G}) \\
&= \{\text{By (4.15)}\} \\
&\quad \mathsf{LAG}(\Gamma, t + cH, \mathcal{G}).
\end{aligned}$$

(b) Assume that for a time instant $t \geq \Phi_{max}$, $\mathsf{A}(\Gamma, t, t + cH, \mathcal{G}) > cHU_{tot}$ holds. By (4.17) and (4.12), we have

$$\begin{aligned}\mathsf{LAG}(\Gamma, t + cH, \mathcal{G}) &= \mathsf{LAG}(\Gamma, t, \mathcal{G}) + \mathsf{A}(\Gamma, t, t + cH, \mathcal{I}) \\ &\quad - \mathsf{A}(\Gamma, t, t + cH, \mathcal{G}) \\ &< \mathsf{LAG}(\Gamma, t, \mathcal{G}) + cHU_{tot} - cHU_{tot} \\ &= \mathsf{LAG}(\Gamma, t, \mathcal{G}), \end{aligned} \tag{4.29}$$

which contradicts (a). □

Lemma 4.44. *For any positive integer c and time instant $t \geq \Phi_{max}$, if $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + cH, \mathcal{G})$ holds, then for any τ_i^v, $\mathsf{lag}(\tau_i^v, t, \mathcal{G}) = \mathsf{lag}(\tau_i^v, t + cH, \mathcal{G})$ holds.*

Proof. Assume there is a task τ_i^v, integer $c > 0$, and time $t \geq \Phi_{max}$ such that $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + cH, \mathcal{G})$ and $\mathsf{lag}(\tau_i^v, t, \mathcal{G}) \neq \mathsf{lag}(\tau_i^v, t + cH, \mathcal{G})$ hold. By Lemma 4.42, $\mathsf{lag}(\tau_i^v, t, \mathcal{G}) < \mathsf{lag}(\tau_i^v, t + cH, \mathcal{G})$. Thus, by (4.15), we have

$$\begin{aligned}&\mathsf{LAG}(\Gamma, t, \mathcal{G}) \\ &= \sum_{w,k} \mathsf{lag}(\tau_k^w, t, \mathcal{G}) \\ &= \mathsf{lag}(\tau_i^v, t, \mathcal{G}) + \sum_{w \neq v, k \neq i} \mathsf{lag}(\tau_k^w, t, \mathcal{G}) \\ &< \{\text{By Lemma 4.42 and } \mathsf{lag}(\tau_i^v, t, \mathcal{G}) < \mathsf{lag}(\tau_i^v, t + cH, \mathcal{G})\} \\ &\quad \mathsf{lag}(\tau_i^v, t + cH, \mathcal{G}) + \sum_{w \neq v, k \neq i} \mathsf{lag}(\tau_k^w, t + cH, \mathcal{G}) \\ &= \{\text{By (4.15)}\} \\ &\quad \mathsf{LAG}(\Gamma, t + cH, \mathcal{G}), \end{aligned}$$

a contradiction. □

Lemma 4.45. *For any positive integer c and time $t \geq \Phi_{max}$, if $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + cH, \mathcal{G})$, then for any job $\tau_{i,j}^v$, the following hold.*

(a) $\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) = \mathsf{lag}(\tau_{i,j+ch^v}^v, t + cH, \mathcal{G})$.

(b) $\mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{G}) = \mathsf{A}(\tau_{i,j+ch^v}^v, 0, t + cH, \mathcal{G})$.

Proof. **(a)** Assume a job $\tau_{i,j}^v$ exists such that $\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) \neq \mathsf{lag}(\tau_{i,j+ch^v}^v, t + cH, \mathcal{G})$. By Lemma 4.39, $\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) < \mathsf{lag}(\tau_{i,j+ch^v}^v, t + cH, \mathcal{G})$. By (4.14), we have

$$
\begin{aligned}
\mathsf{lag}(\tau_i^v, t, \mathcal{G}) &= \sum_{k \geq 1} \mathsf{lag}(\tau_{i,k}^v, t, \mathcal{G}) \\
&= \mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) + \sum_{k \geq 1 \wedge k \neq j} \mathsf{lag}(\tau_{i,k}^v, t, \mathcal{G}) \\
&< \{\text{By Lemma 4.39 and since } \mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) < \mathsf{lag}(\tau_{i,j+ch^v}^v, t + cH, \mathcal{G})\} \\
&\quad \mathsf{lag}(\tau_{i,j+ch^v}^v, t + cH, \mathcal{G}) + \sum_{k \geq 1 \wedge k \neq j} \mathsf{lag}(\tau_{i,k+ch^v}^v, t + cH, \mathcal{G}) \\
&= \sum_{k \geq 1} \mathsf{lag}(\tau_{i,k+ch^v}^v, t + cH, \mathcal{G}) \\
&= \sum_{k > ch^v} \mathsf{lag}(\tau_{i,k}^v, t + cH, \mathcal{G}) \\
&\leq \{\text{By Lemma 4.41}\} \\
&\quad \sum_{1 \leq k \leq ch^v} \mathsf{lag}(\tau_{i,k}^v, t + cH, \mathcal{G}) + \sum_{k > ch^v} \mathsf{lag}(\tau_{i,k}^v, t + cH, \mathcal{G}) \\
&= \sum_{k \geq 1} \mathsf{lag}(\tau_{i,k}^v, t + cH, \mathcal{G}) \\
&= \{\text{By (4.14)}\} \\
&\quad \mathsf{lag}(\tau_i^v, t + cH, \mathcal{G}),
\end{aligned}
$$

which contradicts Lemma 4.44.

(b) Assume for a contradiction that a job $\tau_{i,j}^v$ exists such that $\mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{G}) \neq \mathsf{A}(\tau_{i,j+ch^v}^v, 0, t + cH, \mathcal{G})$. By Lemma 4.37, $\mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{G}) > \mathsf{A}(\tau_{i,j+ch^v}^v, 0, t+cH, \mathcal{G})$. By (4.13), we have

$$
\begin{aligned}
\mathsf{lag}(\tau_{i,j}^v, t, \mathcal{G}) &= \mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{I}) - \mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{G}) \\
&< \{\text{By Lemma 4.38 and since } \mathsf{A}(\tau_{i,j}^v, 0, t, \mathcal{G}) > \mathsf{A}(\tau_{i,j+ch^v}^v, 0, t + cH, \mathcal{G})\} \\
&\quad \mathsf{A}(\tau_{i,j+ch^v}^v, 0, t + cH, \mathcal{I}) - \mathsf{A}(\tau_{i,j+ch^v}^v, 0, t + cH, \mathcal{G}) \\
&= \mathsf{lag}(\tau_{i,j+ch^v}^v, t + cH, \mathcal{G}),
\end{aligned}
$$

which contradicts (a). □

Lemma 4.46. *For any positive integer c and time instant $t \geq \Phi_{max}$, $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + cH, \mathcal{G})$ holds if and only if $\mathsf{A}(\Gamma, t, t + cH, \mathcal{G}) = cHU_{tot}$.*

Proof. By (4.12), $\mathsf{A}(\Gamma, t, t + cH, \mathcal{I}) = cHU_{tot}$. By (4.17), $\mathsf{LAG}(\Gamma, t + cH, \mathcal{G}) = \mathsf{LAG}(\Gamma, t, \mathcal{G}) + \mathsf{A}(\Gamma, t, t + cH, \mathcal{I}) - \mathsf{A}(\Gamma, t, t + cH, \mathcal{G})$. Thus, if $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + cH, \mathcal{G})$ holds, then we have $\mathsf{A}(\Gamma, t, t + cH, \mathcal{G}) = \mathsf{A}(\Gamma, t, t + cH, \mathcal{I}) = cHU_{tot}$. Similarly, if $\mathsf{A}(\Gamma, t, t + cH, \mathcal{G}) = cHU_{tot} = \mathsf{A}(\Gamma, t, t + cH, \mathcal{I})$ holds, then $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + cH, \mathcal{G})$ holds. □

Lemma 4.47. *For any positive integer c and time $t \geq \Phi_{max}$, if $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + cH, \mathcal{G})$, then the following hold.*

(a) $\mathsf{A}(\Gamma_s, t, t + cH, \mathcal{S}) = cHU_{tot}$.

(b) If a server job $S^v_{i,j}$ is scheduled at time $t' \in [t, t + cH)$, then a job is linked to it.

Proof. **(a)** By Lemma 4.46, we have $\mathsf{A}(\Gamma, t, t + cH, \mathcal{G}) = cHU_{tot}$. Since jobs in Γ only executes when server jobs in Γ_s is scheduled, we have $\mathsf{A}(\Gamma_s, t, t + cH, \mathcal{S}) \geq cHU_{tot}$. By Lemma 4.26(b), we have $\mathsf{A}(\Gamma_s, t, t + cH, \mathcal{S}) = cHU_{tot}$.

(b) Since, by (a), $\mathsf{A}(\Gamma, t, t + cH, \mathcal{G}) = \mathsf{A}(\Gamma_s, t, t + cH, \mathcal{S}) = cHU_{tot}$, the lemma holds. □

Lemma 4.48. *For any positive integer c and time $t' \geq \Phi_{max}$, if $\mathsf{LAG}(\Gamma, t', \mathcal{G}) = \mathsf{LAG}(\Gamma, t' + cH, \mathcal{G})$ holds, then for each $t \in [t', t' + (c - 1)H]$, $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + H, \mathcal{G})$.*

Proof. Assume otherwise. Let $t \geq t'$ be the first time instant such that $\mathsf{LAG}(\Gamma, t, \mathcal{G}) \neq \mathsf{LAG}(\Gamma, t+H, \mathcal{G})$ holds. By Lemma 4.46 and 4.43(b), we have $\mathsf{A}(\Gamma, t, t+H, \mathcal{G}) < HU_{tot}$. By Lemma 4.47(a), $\mathsf{A}(\Gamma_s, t', t' + cH, \mathcal{S}) = cHU_{tot}$. Thus, by Lemma 4.34, for each time instant $t \in [t', t' + (c - 1)H]$, we have $\mathsf{A}(\Gamma_s, t, t + H, \mathcal{S}) = HU_{tot}$. Thus, $\mathsf{A}(\Gamma, t, t + H, \mathcal{G}) < \mathsf{A}(\Gamma_s, t, t + H, \mathcal{S})$. Hence, time $t'' \in [t, t + H)$ exists such that a server job is scheduled at t'' but no job is linked to it, contradicting Lemma 4.47(b). □

Definition 4.5. Let $\Delta = \lceil \max_{v,i}\{R(S^v_i)\}/H \rceil H$, where $R(S^v_i)$ is defined in (4.2). Note that $\Delta \geq \max_{v,i}\{R(S^v_i)\}$ and $\Delta \geq H$ hold. ◀

We now address Step 3 by giving Lemmas 4.49–4.51. The repetition of the graph-level schedule $\mathcal{G}$ at time t' requires that the server-level schedule $\mathcal{S}$ repeats at time t' and each server job scheduled at or after t'

has a linked job. To ensure the latter, we need to consider a larger interval of length $(2H + \Delta)$ as shown in the following lemma.

Lemma 4.49. *If* $\mathsf{LAG}(\Gamma, t_s, \mathcal{G}) = \mathsf{LAG}(\Gamma, t_s + 2H + \Delta, \mathcal{G})$ *holds such that* $t_s \geq \Phi_{max}$*, then the following hold.*

(a) If a server job $S^v_{i,j}$ *is released during* $[t_s, t_s + 2H)$*, then a job is linked to it.*

(b) For each v *and* $j \leq h^v$*,* $f(\tau^v_{n^v,j}) \leq t_s + 2H + \Delta$ *holds.*

Proof. **(a)** Assume that there is a server job $S^v_{i,j}$ such that $t_s \leq r(S^v_{i,j}) < t_s + 2H$ holds, but no job is linked to it. Since $S^v_{i,j}$'s response time is at most $R(S^v_i)$, by Definition 4.5, we have $f(S^v_{i,j}) \leq r(S^v_{i,j}) + R(S^v_i) < t_s + 2H + \Delta$. Therefore, there is a time instant t_b such that $t_s \leq t_b < t_s + 2H + \Delta$ and $S^v_{i,j}$ is scheduled during $[t_b, t_b + 1)$. Since H divides Δ, by Lemma 4.47(b), a job is linked to $S^v_{i,j}$, a contradiction.

(b) We first prove the following claim.

Claim 4.6. $f(\tau^v_{n^v,1}) \leq t_s + H + \Delta$.

Proof. Let $S^v_{n^v,k}$ be the first server job of $S^v_{n^v}$ that is released at or after t_s. Since server jobs are released periodically and $t_s \geq \Phi_{max}$, $r(S^v_{n^v,k}) < t_s + H$ holds. Since $S^v_{n^v,k}$'s response time is at most $R(S^v_{n^v})$, by Definition 4.5, we have $f(S^v_{n^v,k}) < t_s + H + R(S^v_{n^v}) \leq t_s + H + \Delta$. By Lemma 4.49(a), $S^v_{n^v,k}$ is linked to a job $\tau^v_{n^v,j}$. By Lemma 4.8, $f(\tau^v_{n^v,j}) \leq f(S^v_{n^v,k})$. If $j = 1$, then the claim holds, so assume $j > 1$. Let $S^v_{n^v,\ell}$ be the server job to which $\tau^v_{n^v,1}$ is linked. By Lemmas 4.6 and 4.2, $\ell < k$ and $f(S^v_{n^v,\ell}) \leq f(S^v_{n^v,k})$ hold. Therefore, by Lemma 4.8, $f(\tau^v_{n^v,1}) \leq f(S^v_{n^v,\ell}) \leq f(S^v_{n^v,k}) < t_s + H + \Delta$ holds. □

We now prove the lemma. By Claim 4.6, we have $f(\tau^v_{n^v,1}) \leq t_s + H + \Delta$. Hence, since H divides Δ and $\mathsf{LAG}(\Gamma, t_s, \mathcal{G}) = \mathsf{LAG}(\Gamma, t_s + 2H + \Delta, \mathcal{G})$, by Lemma 4.48, we have $\mathsf{LAG}(\Gamma, t_s + H + \Delta, \mathcal{G}) = \mathsf{LAG}(\Gamma, t_s + 2H + \Delta, \mathcal{G})$. Thus, by Lemma 4.45(b), we have $\mathsf{A}(\tau^v_{n^v,h^v+1}, 0, t_s + 2H + \Delta, \mathcal{G}) = \mathsf{A}(\tau^v_{n^v,1}, 0, t_s + H + \Delta, \mathcal{G}) = C^v_{n^v}$. By Lemma 4.36, for each $j \leq h^v$, we have $\mathsf{A}(\tau^v_{n^v,j}, 0, t_s + 2H + \Delta, \mathcal{G}) = C^v_{n^v}$. Thus, for each $j \leq h^v$, $\tau^v_{n^v,j}$ completes at or before time $t_s + 2H + \Delta$. □

Using Lemmas 4.47–4.49, we now prove that if Γ's LAG values at time t and $t + 2H + \Delta$ are the same, then this value remains the same over any future H-sized interval.

Lemma 4.50. *If* $\mathsf{LAG}(\Gamma, t_s, \mathcal{G}) = \mathsf{LAG}(\Gamma, t_s + 2H + \Delta, \mathcal{G})$ *holds such that* $t_s \geq \Phi_{max}$*, then for each* $t \geq t_s$*,* $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + H, \mathcal{G})$ *holds.*

Proof. Let t be the first time instant at or after t_s such that $\mathsf{LAG}(\Gamma, t, \mathcal{G}) \neq \mathsf{LAG}(\Gamma, t + H, \mathcal{G})$ holds. Since H divides Δ, by Lemma 4.48 and Definition 4.5, we have the following.

$$t > t_s + H + \Delta \wedge t > t_s + 2H \tag{4.30}$$

We first prove the following claim.

Claim 4.7. *If a server job* $S^v_{i,j}$ *is released during* $[t_s + 2H, t)$*, then a job is linked to it.*

Proof. Figure 4.4 illustrates this proof. Assume otherwise. Let $S^v_{i,j}$ be the first job of S^v_i released during $[t_s + 2H, t)$ to which no job is linked. Let $t_r = r(S^v_{i,j})$. By Lemma 4.49(a) and the definition of t_r, we have:

Property 4.1. *Each server job of* S^v_i *released during* $[t_s, t_r)$ *has a job that is linked to it.*

Since H divides Δ and $t_r \in [t_s + 2H, t)$, by the definition of t, we have

$$\mathsf{LAG}(\Gamma, t_r - H, \mathcal{G}) = \mathsf{LAG}(\Gamma, t_r, \mathcal{G}). \tag{4.31}$$

Since server jobs are released periodically, $r(S^v_{i,j-1}) = t_r - T^v \geq t_r - H \geq t_s + 2H - H = t_s + H$. Thus, by Property 4.1, a job $\tau^v_{i,\ell}$ is linked to $S^v_{i,j-1}$. We now prove that $\tau^v_{i,\ell+1}$ is linked to $S^v_{i,j}$, thereby reaching a contradiction. By Rule R2, it suffices to prove that $r(\tau^v_{i,\ell+1}) \leq t_r$. We now prove the claim by considering two cases.

Case 1. $i = 1$. Thus, τ^v_i is the source node of G^v. Therefore, $r(\tau^v_{i,\ell+1}) = r(\tau^v_{i,\ell}) + T^v$. Since $\tau^v_{i,\ell}$ is linked to $S^v_{i,j-1}$, by Rule R2, we have $r(\tau^v_{i,\ell}) \leq r(S^v_{i,j-1}) = t_r - T^v$. Since τ^v_i releases job periodically, we have $r(\tau^v_{i,\ell+1}) \leq t_r$.

Case 2. $i > 1$. Thus, τ^v_i is a non-source node. Assume to the contrary that $r(\tau^v_{i,\ell+1}) > t_r$ (see Figure 4.4). Since a non-source node's $(\ell+1)^{st}$ job is released once each of its predecessors' $(\ell+1)^{st}$ job completes, there is a job $\tau^v_{k,\ell+1}$ such that $\tau^v_k \in pred(\tau^v_i)$ and $f(\tau^v_{k,\ell+1}) > t_r$. Therefore, we have

$$\mathsf{A}(\tau^v_{k,\ell+1}, 0, t_r, \mathcal{G}) < C^v_k. \tag{4.32}$$

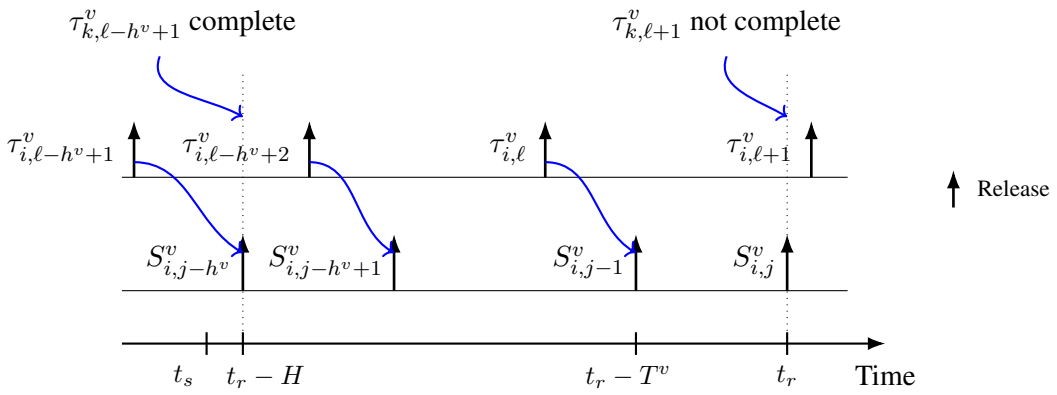


Figure 4.4: Illustration of Claim 4.7. Blue arrows from job releases to server job releases represent linking.

Since $t_r \in [t_s + 2H, t)$, we have $t_r - H \in [t_s, t)$. By Lemma 4.11, $S^v_{i,j-h^v}$ is released at time $t_r - H$ (thus, $j > h^v$). By Property 4.1, each server job of S^v_i released during $[t_r - H, t_r)$ has a job to which it is linked. Thus, since $S^v_{i,j-h^v}$ and $S^v_{i,j-1}$ are released at time $t_r - H$ and $t_r - T^v$, respectively, each server job $S^v_{i,j-b}$ such that $1 \le b \le h^v$ has a linked job. Since $\tau^v_{i,\ell}$ is linked to $S^v_{i,j-1}$, by Rule R2, for each $1 \le b \le h^v$, $\tau^v_{i,\ell-b+1}$ is linked to $S^v_{i,j-b}$. Thus, $\tau^v_{i,\ell-h^v+1}$ (hence, $\ell + 1 > h^v$) is linked to $S^v_{i,j-h^v}$. Since $\tau^v_{i,\ell-h^v+1}$ is linked to $S^v_{i,j-h^v}$ and $\tau^v_k \in pred(\tau^v_i)$, $f(\tau^v_{k,\ell-h^v+1}) \le r(S^v_{i,j-h^v}) = t_r - H$. Therefore, we have $\mathsf{A}(\tau^v_{k,\ell-h^v+1}, 0, t_r - H, \mathcal{G}) = C^v_k$. By (4.31), Lemma 4.45(b), we have $\mathsf{A}(\tau^v_{k,\ell+1}, 0, t_r, \mathcal{G}) = \mathsf{A}(\tau^v_{k,\ell-h^v+1}, 0, t_r - H, \mathcal{G}) = C^v_k$, which contradicts (4.32). □

Claim 4.8. *If a server job $S^v_{i,j}$ is scheduled during $[t-1, t)$, then a job is linked to it.*

Proof. By (4.30), $t > t_s + H + \Delta$ holds. By Definition 4.5, any server job released before t_s completes at or before time $t_s + \Delta < t$. Therefore, no server job released before t_s is pending at time $t - 1$. Thus, $S^v_{i,j}$ is released at or after time t_s. By Lemma 4.49(a) and Claim 4.7, a job is linked to $S^v_{i,j}$. □

Using the above claims, we now prove the lemma. By (4.30) and the definition of t, we have

$$\mathsf{LAG}(\Gamma, t - H - 1, \mathcal{G}) = \mathsf{LAG}(\Gamma, t - 1, \mathcal{G}). \tag{4.33}$$

By (4.33) and Lemma 4.47(a), we have $\mathsf{A}(\Gamma_s, t - H - 1, t - 1, \mathcal{S}) = HU_{tot}$. Therefore, by Lemma 4.33(c), we have

$$\mathsf{A}(\Gamma_s, t - H - 1, t - H, \mathcal{S}) = \mathsf{A}(\Gamma_s, t - 1, t, \mathcal{S}). \tag{4.34}$$

By (4.33) and Lemma 4.47(b), any server job scheduled during $[t-H-1, t-H)$ has a job linked to it. By Claim 4.8, any server job scheduled during $[t-1, t)$ has a job linked to it. Therefore, by (4.34), we have

$$\mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{G}) = \mathsf{A}(\Gamma, t-1, t, \mathcal{G}). \tag{4.35}$$

By (4.33), (4.12), and (4.35), we have $\mathsf{LAG}(\Gamma, t-1, \mathcal{G}) + \mathsf{A}(\Gamma, t-1, t, \mathcal{I}) - \mathsf{A}(\Gamma, t-1, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t-H-1, \mathcal{G}) + \mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{I}) - \mathsf{A}(\Gamma, t-H-1, t-H, \mathcal{G})$. Thus, by (4.17), $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t-H, \mathcal{G})$ holds, a contradiction. □

We now complete Step 3 by giving the following lemma.

Lemma 4.51. *If $\mathsf{LAG}(\Gamma, t_s, \mathcal{G}) = \mathsf{LAG}(\Gamma, t_s + 2H + \Delta, \mathcal{G})$ holds such that $t_s \geq \Phi_{max}$ and R^v is the maximum response time of DAG jobs of G^v that complete at or before time $t_s + 2H + \Delta$ in $\mathcal{G}$, then the response time of G^v is R^v in $\mathcal{G}$.*

Proof. Assume for a contradiction that G^v_j is the first DAG job of G^v with response time more than R^v. Assume that $\tau^v_{n^v,j}$ completes at time t, *i.e.,* $t = f(\tau^v_{n^v,j})$. Thus, we have

$$t - r(\tau^v_{1,j}) > R^v. \tag{4.36}$$

Since R^v is the maximum observed response time of G^v at or before $t_s + 2H + \Delta$, we have $t > t_s + 2H + \Delta$. Therefore, by Lemma 4.49(b), $j > h^v$ holds. At time $t-1$, $\tau^v_{n^v,j}$ is pending. Thus, $\mathsf{A}(\tau^v_{n^v,j}, 0, t-1, \mathcal{G}) < C^v_i$. Since $t - 1 \geq t_s + 2H + \Delta$, by Lemma 4.50, we have $\mathsf{LAG}(\Gamma, t-1, \mathcal{G}) = \mathsf{LAG}(\Gamma, t-H-1, \mathcal{G})$. Then, by Lemma 4.45(b), $\mathsf{A}(\tau^v_{n^v,j-h^v}, 0, t-H-1, \mathcal{G}) = \mathsf{A}(\tau^v_{n^v,j}, 0, t-1, \mathcal{G}) < C^v_i$. Thus, $\tau^v_{n^v,j-h^v}$ completes after time $t-H-1$, *i.e.,* $f(\tau^v_{n^v,j-h^v}) \geq t-H$. Thus, we have $f(\tau^v_{n^v,j-h^v}) - r(\tau^v_{1,j-h^v}) \geq t - H - r(\tau^v_{1,j-h^v})$. By Lemma 4.11, $t - H - r(\tau^v_{1,j-h^v}) = t - r(\tau^v_{1,j})$, which by (4.36), exceeds R^v. Therefore, we have $f(\tau^v_{n^v,j-h^v}) - r(\tau^v_{1,j-h^v}) > R^v$. Thus, $G^v_{j-h^v}$'s response time is more than R^v, a contradiction. □

We now complete Step 4. Our goal is to show that there exists a time instant t_s such that Γ's LAG values at time t_s and $t_s + 2H + \Delta$ are the same. This, by Lemma 4.51, implies that a DAG job with the maximum response time completes execution at or before time $t_s + 2H + \Delta$. We first give an upper bound and a lower bound of LAG of Γ at any time instant.

Definition 4.6. Let $E = \sum_{v=1}^{N} \sum_{i=1}^{n^v} R(\tau^v_i) u^v_i$, $F = \sum_{v=1}^{N} \sum_{i=1}^{n^v} C^v_i$, and $G = \lceil E + F + 1 \rceil$. ◀

Since τ_i^v's response time is at most $R(\tau_i^v)$ (by Theorem 4.1), we can show that τ_i^v's lag at any time is at most $R(\tau_i^v)u_i^v$.

Lemma 4.52. *For any τ_i^v and time instant t, $\mathsf{lag}(\tau_i^v, t, \mathcal{G}) \leq R(\tau_i^v)u_i^v$ holds.*

Proof. Since τ_i^v executes at rate u_i^v starting from time Φ^v in $\mathcal{I}$, for any time $t' \leq \Phi^v + R(\tau_i^v)$, $\mathsf{A}(\tau_i^v, 0, t', \mathcal{I}) \leq R(\tau_i^v)u_i^v$. Thus, by (4.15), for $t \leq \Phi^v + R(\tau_i^v)$, the claim holds. We thus assume $t > \Phi^v + R(\tau_i^v)$. Let G_j^v be the last DAG job of G^v released at or before time $t - R(\tau_i^v)$. Hence, by Theorem 4.1, any job $\tau_{i,k}^v$ such that $k \leq j$ completes by time $t - R(\tau_i^v) + R(\tau_i^v) = t$ in $\mathcal{G}$. Thus, we have

$$\begin{aligned}
\mathsf{A}(\tau_i^v, 0, t, \mathcal{G}) &\geq \sum_{k=1}^{j} C_i^v \\
&= \sum_{k=1}^{j} T^v u_i^v \\
&= \sum_{k=1}^{j} (r(\tau_{v,1})k + 1 - r(\tau_{v,1})k)u_i^v \\
&= (r(\tau_{v,1})j + 1 - r(\tau_{v,1})1)u_i^v \\
&= (r(\tau_{v,1})j + 1 - \Phi^v)u_i^v \\
&\geq (t - R(\tau_i^v) - \Phi^v)u_i^v. \qquad (4.37)
\end{aligned}$$

By (4.11), we have $\mathsf{A}(\tau_i^v, 0, t, \mathcal{I}) = (t - \Phi^v)u_i^v$. Thus, by (4.14) and (4.37), we have $\mathsf{lag}(\tau_i^v, t, \mathcal{G}) \leq R(\tau_i^v)u_i^v$. □

Since no job executes before its release, τ_i^v's lag at any time is at least $-C_i^v$. Using these, we have the following lemma.

Lemma 4.53. *For any time instant t, $-F \leq \mathsf{LAG}(\Gamma, t, \mathcal{G}) \leq E$.*

Proof. $\mathsf{LAG}(\Gamma, t, \mathcal{G}) \leq E$ holds by Definition 4.6 and (4.15), and since per-task lag is upper bounded by $R(\tau_i^v)u_i^v$. $\mathsf{LAG}(\Gamma, t, \mathcal{G}) \geq -F$ holds by Definition 4.6 and (4.15), and since per-task lag is lower bounded by C_i^v. □

Since all task parameters are integers, if for any positive c, $\mathsf{LAG}(\Gamma, t + cH, \mathcal{G}) > \mathsf{LAG}(\Gamma, t, \mathcal{G})$ holds, then Γ's LAG at time $t + cH$ is at least one unit larger than its LAG at time t. Therefore, since LAG either

increases or remains the same over any interval $[t, t + cH)$, it cannot increase over G consecutive intervals of size cH without violating the LAG upper bound. Thus, we have the following lemma.

Lemma 4.54. *There is a time instant* $t \in [\Phi_{max}, \Phi_{max} + G(2H + \Delta)]$ *such that* $\mathsf{LAG}(\Gamma, t, \mathcal{G}) = \mathsf{LAG}(\Gamma, t + 2H + \Delta, \mathcal{G})$ *holds.*

Proof. Assume otherwise. Then for each time instant $t \in [\Phi_{max}, \Phi_{max} + G(2H + \Delta)]$, $\mathsf{LAG}(\Gamma, t, \mathcal{G}) \neq \mathsf{LAG}(\Gamma, t + 2H + \Delta, \mathcal{G})$. Since Δ divides H, by Lemma 4.46, for each $t \in [\Phi_{max}, \Phi_{max} + G(2H + \Delta)]$, we have $\mathsf{A}(\Gamma, t, t+2H+\Delta, \mathcal{G}) \neq (2H+\Delta)U_{tot}$. Thus, by Lemma 4.43(b), for each $t \in [\Phi_{max}, \Phi_{max}+G(2H+\Delta)]$, we have $\mathsf{A}(\Gamma, t, t + 2H + \Delta, \mathcal{G}) < (2H + \Delta)U_{tot}$. We have $U_{tot} = \sum_{v=1}^{N} U^v = \sum_{v=1}^{N} \sum_{i=1}^{n^v} u_i^v = \sum_{v=1}^{N} \sum_{i=1}^{n^v} C_i^v/T^v = (\sum_{v=1}^{N} \sum_{i=1}^{n^v} h^v C_i^v)/H$. Since for each v and i, h^v and C_i^v are integers, HU_{tot} is also an integer. Since H divides Δ, $(2H + \Delta)U_{tot}$ is also an integer. Thus, $\mathsf{A}(\Gamma, t, t + 2H + \Delta, \mathcal{G}) \leq (2H + \Delta)U_{tot} - 1$.

Since $[\Phi_{max}, \Phi_{max} + G(2H + \Delta)] = \bigcup_{i=0}^{G-1} [\Phi_{max} + i(2H + \Delta), \Phi_{max} + (i + 1)(2H + \Delta))$, we have $\mathsf{A}(\Gamma, \Phi_{max}, \Phi_{max} + G(2H + \Delta), \mathcal{G}) = \sum_{i=0}^{G-1} \mathsf{A}(\Gamma, \Phi_{max} + i(2H + \Delta), \Phi_{max} + (i + 1)(2H + \Delta), \mathcal{G}) \leq \sum_{i=0}^{G-1} ((2H + \Delta)U_{tot} - 1) \leq G(2H + \Delta)U_{tot} - G$. Thus, by (4.17), we have

$$
\begin{aligned}
&\mathsf{LAG}(\Gamma, \Phi_{max} + G(2H + \Delta), \mathcal{G}) \\
&\quad = \mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{G}) + \mathsf{A}(\Gamma, \Phi_{max}, \Phi_{max} + G(2H + \Delta), \mathcal{I}) \\
&\qquad - \mathsf{A}(\Gamma, \Phi_{max}, \Phi_{max} + G(2H + \Delta), \mathcal{G}) \\
&\quad \geq \mathsf{LAG}(\Gamma, \Phi_{max}, \mathcal{G}) + G(2H + \Delta)U_{tot} - G(2H + \Delta)U_{tot} + G \\
&\quad \geq \{\text{By Lemma 4.53 and Definition 4.6}\} \\
&\qquad - F + E + F + 1 \\
&\quad \geq E + 1,
\end{aligned}
$$

which contradicts Lemma 4.53. □

Finally, we have the following theorem.

Theorem 4.2. *If* R^v *is the maximum response time of any DAG job of* G^v *completed at or before* $\Phi_{max} + (G + 1)(2H + \Delta)$*, then* G^v*'s response time is* R^v*, where* Δ *and* G *are defined by Definitions 4.5 and 4.6, respectively.*

Proof. Follows from Lemmas 4.54 and 4.51. □

Thus, simulating schedule $\mathcal{G}$ for at most $\Phi_{max} + (G+1)(2H+\Delta)$ time units is sufficient to determine the maximum response times of DAGs. However, the simulation can be terminated early by checking whether the condition given in Lemma 4.51 is met. For pseudo-harmonic task systems, by Definition 4.6 (resp., Definition 4.5 and (4.2)), G (resp., Δ) is polynomial with respect to the task and processor count and task parameters. Thus, for pseudo-harmonic task systems, simulating for $\Phi_{max} + (G+1)(2H+\Delta)$ time takes pseudo-polynomial time.

Note that Leung and Merrill gave a simulation length of $\Phi_{max} + 2H$ for deriving exact response-time bound of independent periodic tasks under uniprocessor EDF. The simulation length in Theorem 4.2 becomes equal to $\Phi_{max} + 2H$ when $G = 0$ and $\Delta = 0$ hold. However, for $M = 1$, $\Delta \geq 1$ holds by Definition 4.5 (as $R(S_i^v) > 0$ for all v and i). Moreover, by (4.2), $R(S_i^v)$ is at least $T^v + C_i^v$ even for $M = 1$ due to analytical pessimism present in SRT response-time analysis [Amert et al., 2019].

Removing Assumption 4.1. Let $\mathcal{G}'$ be a schedule of Γ when Assumption 4.1 does not hold. Theorem 4.3 below ensures that no job finishes later in $\mathcal{G}'$ than $\mathcal{G}$. Informally, no job is linked to a later server job in $\mathcal{G}'$ than in $\mathcal{G}$.

Theorem 4.3. *For each job $\tau_{i,j}^v$, if it completes at time t and t' in $\mathcal{G}$ and $\mathcal{G}'$, respectively, then $t' \leq t$ holds.*

Proof. We first note that the server schedule corresponding to both $\mathcal{G}$ and $\mathcal{G}'$ are the same. We now prove the following claims.

Claim 4.9. *If a job $\tau_{i,j}^v$ is linked to $S_{i,k}^v$ and $S_{i,\ell}^v$ in $\mathcal{G}$ and $\mathcal{G}'$, respectively, and $\ell \leq k$ holds, then $\tau_{i,j}^v$ finishes execution in $\mathcal{G}'$ at or before $\tau_{i,j}^v$ finishes execution in $\mathcal{G}$.*

Proof. Suppose $\tau_{i,j}^v$ finishes execution at time t and t' in $\mathcal{G}$ and $\mathcal{G}'$, respectively. Since $\ell \leq k$ holds, by Lemma 4.2, $f(S_{i,\ell}^v) \leq f(S_{i,k}^v)$. Since $\tau_{i,j}^v$ executes for C_i^v time units in $\mathcal{G}$, $t = f(S_{i,k}^v)$ holds. Since $\tau_{i,j}^v$ executes for at most C_i^v time units in $\mathcal{G}'$, $t' \leq f(S_{i,\ell}^v)$ holds. Thus, $t' \leq t$. □

Claim 4.10. *If a job $\tau_{i,j}^v$ is linked to $S_{i,k}^v$ and $S_{i,\ell}^v$ in $\mathcal{G}$ and $\mathcal{G}'$, respectively, then $\ell \leq k$ holds.*

Proof. Assume otherwise. Let t be the first time instant such that $t = r(S_{i,k}^v)$ holds and there is a job $\tau_{i,j}^v$ that is linked to $S_{i,k}^v$ in $\mathcal{G}$, but it is linked to $S_{i,\ell}^v$ in $\mathcal{G}'$ such that $\ell > k$ holds. Therefore, $r(S_{i,k}^v) < r(S_{i,\ell}^v)$. We consider two cases.

Case 1. $i = 1$. Since τ_1^v does not have any predecessor, $\tau_{1,j}^v$'s release time is the same in both $\mathcal{G}$ and $\mathcal{G}'$. Let t_r be the time instant when $\tau_{1,j}^v$ is released. By Rule R2, $t_r \leq r(S_{1,k}^v)$. Since $\tau_{1,j}^v$ is linked to $S_{1,k}^v$ in $\mathcal{G}$, by Rule R2, $\tau_{1,j-1}^v$ (if any) is already linked to a server job by time t in $\mathcal{G}$. By the definition of time t, $\tau_{1,j-1}^v$ (if any) is linked to a server job by time t in $\mathcal{G}'$. Thus, by Rule R2, $\tau_{1,j}^v$ is linked to $S_{1,k}^v$ in $\mathcal{G}'$, a contradiction.

Case 2. $i > 1$. Let τ_p^v be a task such that $\tau_p^v \in pred(\tau_i^v)$. Since $\tau_{i,j}^v$ is linked to $S_{i,k}^v$ in $\mathcal{G}$, by Rule R2, $\tau_{i,j}^v$ is released at or before time t in $\mathcal{G}$. Thus, $\tau_{p,j}^v$ is complete at time t in $\mathcal{G}$. Let $S_{p,x}^v$ be the server job to which $\tau_{p,j}^v$ is linked in $\mathcal{G}$. By the definition of time t, $\tau_{p,j}^v$ is linked to a server job $S_{p,y}^v$ in $\mathcal{G}'$ such that $y \leq x$ holds. By Claim 4.9, $\tau_{p,j}^v$ finishes execution in $\mathcal{G}'$ at or before it finishes execution in $\mathcal{G}$. Thus, $\tau_{i,j}^v$ is released in $\mathcal{G}'$ at or before its release time in $\mathcal{G}$. Let t_r and t_r' be the release times of $\tau_{i,j}^v$ in $\mathcal{G}$ and $\mathcal{G}'$, respectively. By Rule R2, $t_r \leq r(S_{i,k}^v) = t$ holds. Thus, we have $t_r' \leq t_r \leq t$. Since $\tau_{i,j}^v$ is linked to $S_{i,k}^v$ in $\mathcal{G}$, by Rule R2, $\tau_{i,j-1}^v$ (if any) is already linked to a server job at time t in $\mathcal{G}$. By the definition of time t, $\tau_{i,j-1}^v$ (if any) is linked to a server job at time t in $\mathcal{G}'$. Thus, by Rule R2, $\tau_{i,j}^v$ is linked to $S_{i,k}^v$ in $\mathcal{G}'$, a contradiction. □

The theorem follows from Claims 4.9 and 4.10. □

Slack reallocation. The response times of DAG tasks may potentially be improved by utilizing budgets of server jobs that have no linked job. Assuming $S_{i,j}^v$ is scheduled at time t, we propose the following slack reallocation policy.

Q1. If $S_{i,j}^v$ has no linked job or its linked job completes at or before time t, then the highest priority ready but unscheduled job of τ_i^v is scheduled on $S_{i,j}^v$ at time t.

When each P_i^v equals 1, the bounds in Theorem 4.2 are also exact with slack reallocation. This is because the allocation received by each server over any H-sized interval is at most HU_{tot}, as shown in Chapter 3, which translates to a similar task-level property.

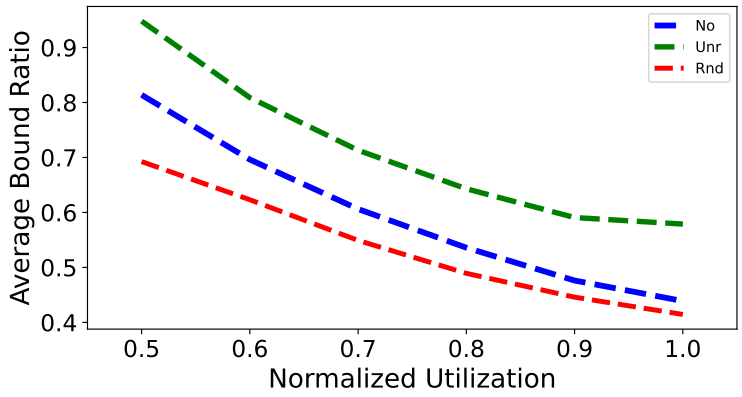
Asynchronous releases. Instead of synchronous server releases, asynchronous server releases are possible. We chose to limit attention to the former for simplicity (*e.g.,* asynchronous releases would necessitate different ideal schedules for tasks and servers).

4.5 Experimental Evaluation

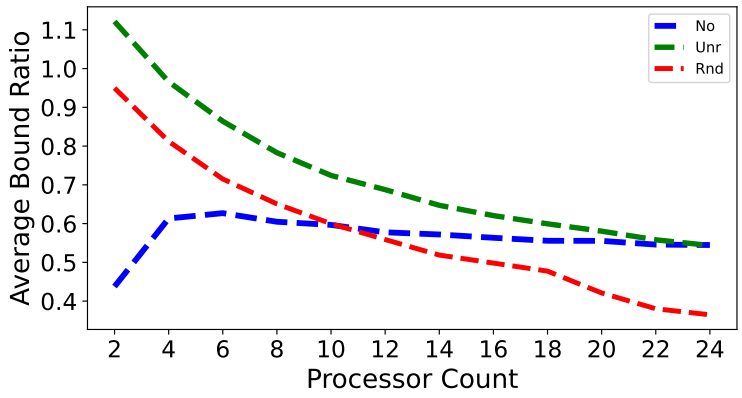
We now present the results of simulation experiments we conducted to evaluate the response-time bounds of our proposed scheduler. We compared our scheduler to other schedulers that provide bounded response times without utilization loss.

We generated task systems randomly for systems with 2 to 24 processors with a step size of 2.0. Such processor counts are common in real-world use cases [Akesson et al., 2022; Kato et al., 2018]. For each processor count, we generated task systems that have *normalized utilization*, *i.e.,* U_{tot}/M, from 0.5 to 1 with a step size of 0.1. Each task system consists of one or more DAGs. The number of DAGs was chosen uniformly from $[1, \lfloor U_{tot}/2 \rfloor]$. Motivated by automotive use cases, each DAG's period was uniformly selected from $\{1, 2, 5, 10, 20, 50, 100, 200\}$ms [Kramer et al., 2015]. The offset of each DAG was uniformly selected between 0 and its period. The number of nodes per DAG was chosen uniformly from $[10, 100)$. Each node's utilization was chosen uniformly following procedures from [Emberson et al., 2010]. The WCET of each node was rounded to the nearest microsecond. Edges were generated following the *Erdős-Rényi method* [Cordeiro et al., 2010], where an edge was added between two nodes if a uniformly generated number in $[0, 1]$ is at most a predefined *edge-generation probability*. We selected this probability value from $\{0.1, 0.3, 0.5, 0.7, 0.9\}$. As in [Saifullah et al., 2014], a minimum number of additional edges was added to make each DAG weakly connected. Each edge was directed from a lower-indexed task to a higher-indexed task. For each combination of processor count, normalized utilization, and edge-generation probabilities, we generated 1,000 random task systems.

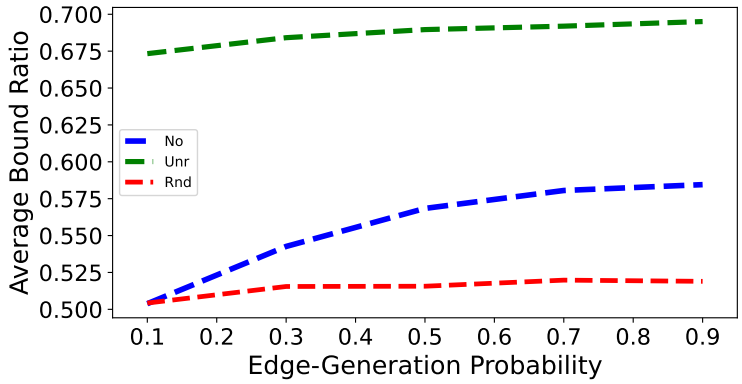
We considered three scenarios for each generated task system depending on task parallelism levels. In scenarios No and Unr, each task's parallelism level was set to 1 and M, respectively. In scenario Rnd, task parallelism levels were generated uniformly between 1 and M. For scenarios No, Unr, and Rnd, we compared our response-time bounds with those from [Liu and Anderson, 2010], [Yang et al., 2016], and [Amert et al., 2019], respectively. Recall from Chapter 2 that these works convert each DAG task into an "equivalent" independent sporadic task set and schedule the converted tasks by G-EDF. The response-time bounds from these prior works are non-exact and can be computed in polynomial time. For each scenario, we computed the average and maximum *bound ratio*, which is the ratio of the average and maximum response-time bound of our method to those of the corresponding prior method (so ratios below 1.0 show improvement by our method). These ratios are plotted in Figures 4.5 and 4.6.



(a) Average bound ratio vs. normalized utilization.



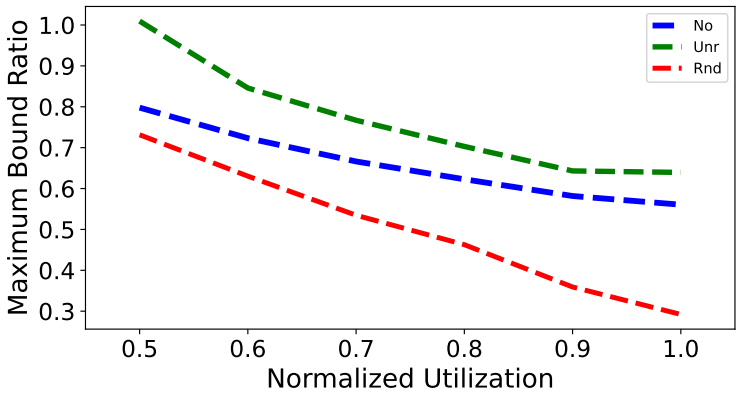
(b) Average bound ratio vs. processor count.



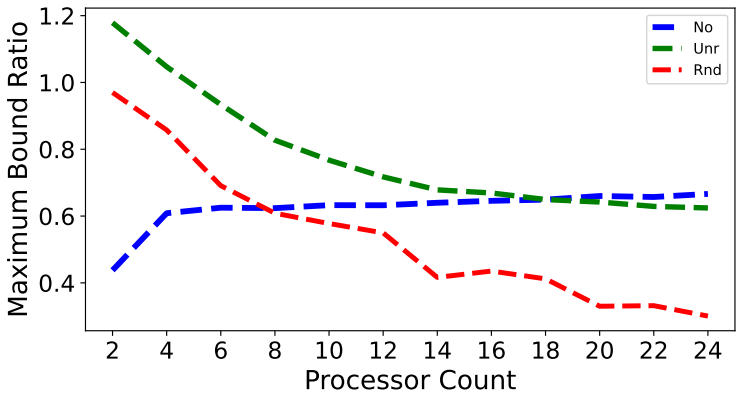
(c) Average bound ratio vs. edge-generation probability.

Figure 4.5: Averate bound ratios.

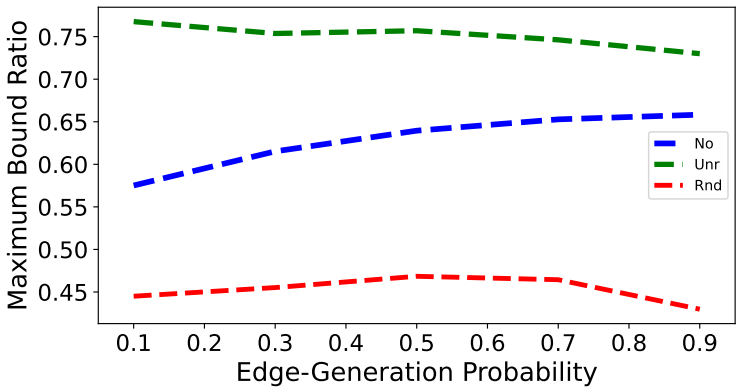
Observation 4.1. *For No, Rnd, and Unr, the average improvement of our bound over prior methods was around 43%, 48%, and 31%, respectively.*



(a) Maximum bound ratio vs. normalized utilization.



(b) Maximum bound ratio vs. processor count.



(c) Maximum bound ratio vs. edge-generation probability.

Figure 4.6: Maximum bound ratios.

The improvement is due to the pessimism inherent to prior bounds. Prior bounds that consider arbitrary parallelism levels suffer from pessimism present in the analysis of both no and unrestricted parallelism. This

yields a larger improvement for the Rnd scenario. The improvement is less for Unr as prior analysis with unrestricted parallelism is less pessimistic. Note that asynchronous server releases, as discussed earlier, may yield additional improvement.

Observation 4.2. *Our method provided a larger improvement with increasing (resp., decreasing) normalized utilization (resp., edge-generation probabilities). Except No, our method provided larger improvement as the processor count increases.*

This can be seen in Figure 4.5(a)–(c). The large improvement for higher normalized utilizations or processor counts is due to the increased pessimism in the corresponding prior analysis. For scenario No, the large improvement for small processor counts is due to the usage of slack reallocation. In contrast, for scenario Unr and small processor counts, the prior method gave smaller bounds. This happens because prior analysis is reasonably tight under the corresponding scheduling policy for small processor counts, while jobs may be delayed waiting for their linked server jobs in our scheduling strategy. With increasing edge-generation probabilities, DAGs become more *sequential*, which limits improvement under our method.

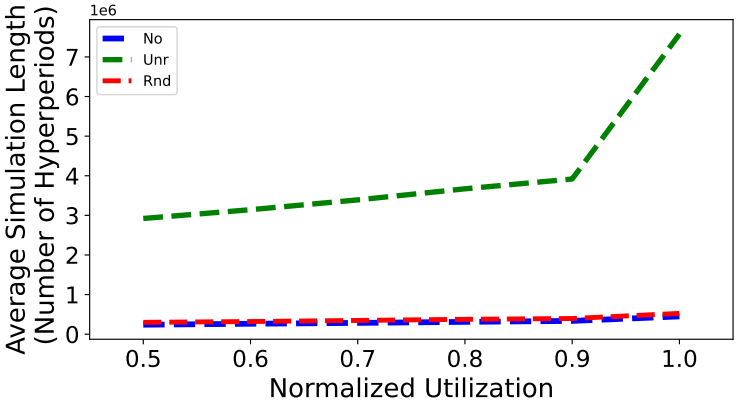
To determine the tightness of the simulation length, we computed the analytical simulation length from Theorem 4.2 and the actual simulation length by checking when the condition given in Lemma 4.51 is met for the first time. The observation below indicates that the analytical simulation length is pessimistic.

Observation 4.3. *The average analytical simulation length (from Theorem 4.2) is 3,564,060 times larger than the average actual simulation length.*

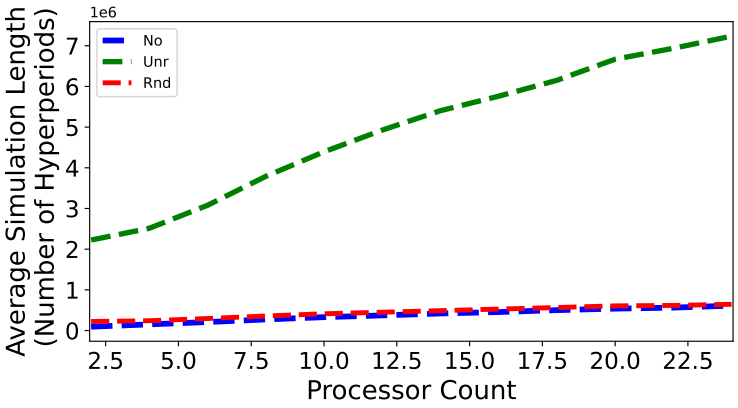
Figure 4.7 presents average simulation lengths in number of hyperperiods with respect to normalized utilizations, processor count, and edge-generation probabilities. Simulation lengths in Unr were around three times larger than No and Rnd. Unsurprisingly, simulation lengths were larger for large normalized utilizations, processor count, and edge-generation probabilities for all Unr, No, and Rnd.

Similar to simulation lengths, simulation time in Unr were larger than No and Rnd. However, across all generated task systems, our method found exact response times within reasonable time. Note that the execution time of our method depends on the hyperperiod and the granularity of time units.

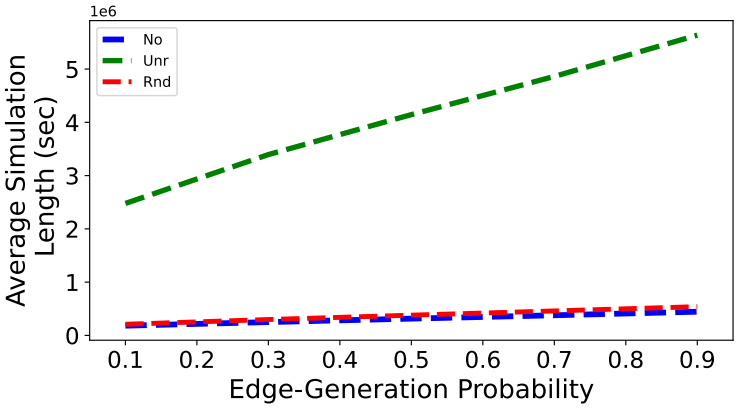
Observation 4.4. *The average (resp., maximum) simulation time (on a 24-core 2.50 GHz machine) was 6.83s (resp., 10872.81s). The average (resp., maximum) time to compute prior bounds was 0.10s (resp., 9.71s).*



(a) Simulation-interval length vs. normalized utilization.



(b) Simulation-interval length vs. processor count.

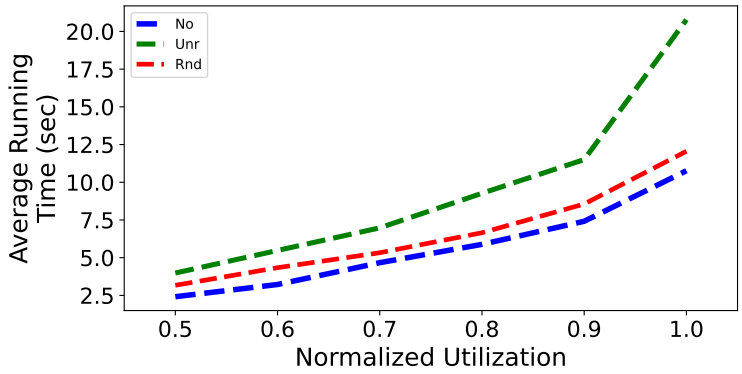


(c) Simulation-interval length vs. edge-generation probability.

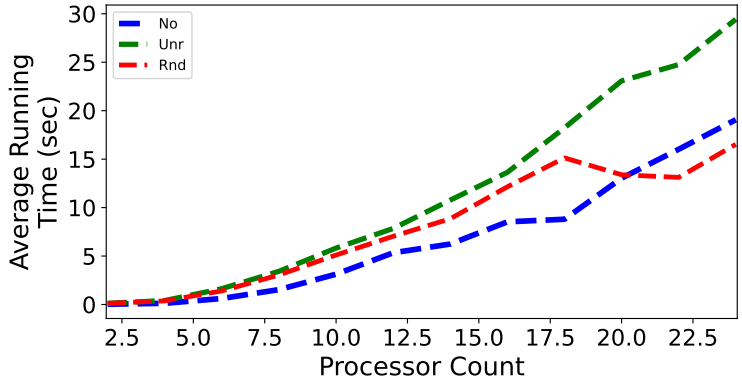
Figure 4.7: Simulation-interval length.

4.6 Chapter Summary

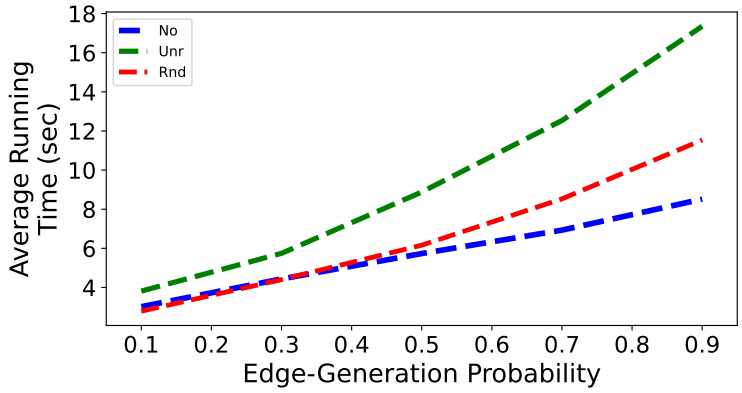
In this chapter, we presented a server-based scheduling policy for DAG tasks. We showed that server-based scheduling policy is SRT-optimal for scheduling DAG tasks under the rp model. We also gave a method



(a) Running time vs. normalized utilization.



(b) Running time vs. processor count.



(c) Running time vs. edge-generation probability.

Figure 4.8: Simulation execution times.

to compute exact response-time bounds under this policy. Moreover, our method takes pseudo-polynomial time for pseudo-harmonic DAG tasks.

CHAPTER 5: SUSPENSION-BASED MULTIPROCESSOR LOCKING PROTOCOLS[1]

In this chapter, we consider systems with non-CPU shared resources. Recall from Section 2.3 that asymptotically optimal suspension-based locking protocols for mutual exclusion (mutex) sharing have a factor of two in their pi-blocking bounds, while the best-known lower-bound result lacks this factor. In this chapter, we show that the factor of two is fundamental for a class of global JLFP schedulers that contains G-EDF and G-FP but not G-FIFO. For C-FIFO (thus, also G-FIFO and P-FIFO) scheduling, we show the opposite by devising a locking protocol that does not have the factor of two in its pi-blocking bound. We also give an optimal locking protocol for *k-exclusion* sharing and a nearly optimal one for *reader-writer* sharing under C-FIFO scheduling.

Organization. In the rest of the chapter, we first describe our system model (Section 5.1), then give lower-bound results for non-FIFO global JLFP schedulers (Section 5.2), present optimal locking protocols for FIFO schedulers (Section 5.3), present evaluation results (Section 5.4), and provide a summary (Section 5.5).

5.1 System Model

In this section, we describe the considered system model. Table 5.1 summarizes the notation given here.

Task model. We consider a system Γ of N sporadic *tasks* $\tau_1, \tau_2, \ldots, \tau_N$ to be scheduled on M identical processors. Each task τ_i releases a potentially infinite sequence of *jobs* $\tau_{i,1}, \tau_{i,2}, \ldots$. (We omit job indices if they are irrelevant.) Each task τ_i has a *period* T_i specifying the minimum spacing between consecutive job releases. The maximum and minimum periods among all tasks are denoted by T_{max} and T_{min}, respectively. Each task has a *relative deadline* D_i. The minimum relative deadline among all tasks is denoted by D_{min}. Each task τ_i has a WCET denoted by C_i. Each task τ_i has a parallelization level P_i that specifies the number

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

Ahmed, S. and Anderson, J. (2023a), Optimal Multiprocessor Locking Protocols under FIFO Scheduling, *Proceedings of the 35th Euromicro Conference on Real-Time Systems*, pages 16.1-16.21.

Ahmed, S. and Anderson, J. (2024), Open Problem Resolved: The 'Two' in Existing Multiprocessor PI-Blocking Bounds is Fundamental, *Proceedings of the 36th Euromicro Conference on Real-Time Systems*, pages 11.1-11.21.

Table 5.1: Notation summary for Chapter 5.

Symbol	Meaning
Γ	Task system
N	Number of tasks
M	Number of processors
τ_i	i^{th} task
T_i	Period of τ_i
C_i	WCET of τ_i
Y_i	RPP of τ_i
u_i	Utilization of τ_i
P_i	Parallelization level of τ_i
T_{min}	$\min_i\{T_i\}$
Φ_{max}	$\max_i\{\Phi_i\}$
D_{min}	$\min_i\{D_i\}$
$\tau_{i,j}$	j^{th} job of τ_i
$r(\tau_{i,j})$	Release time of $\tau_{i,j}$
$f(\tau_{i,j})$	Completion time of $\tau_{i,j}$
$y(\tau_{i,j})$	PP of $\tau_{i,j}$
n_r	Number of resources
ℓ_q	q^{th} resource
N_i^q	Maximum number of requests for ℓ_q by τ_i
L_i^q	Maximum request length for ℓ_q by τ_i
L_{max}^q	$\max_{1\leq i\leq n}\{L_i^q\}$
L_{max}	$\max_{1\leq q\leq n_r} L_{max}^q$
$\mathcal{R}$	A request
$L_{sum,h}^q$	Definition 5.1

of successive jobs of τ_i that can execute in parallel. Task τ_i's *utilization* is defined as $u_i = C_i/T_i$. The *release time* (resp., *finish time*) of a job $\tau_{i,j}$ is given by $r(\tau_{i,j})$ (resp., $f(\tau_{i,j})$).

We consider *clustered scheduling*[2] of jobs in Γ. Recall that clustered scheduling is a hybrid of partitioned and global scheduling, where all M processors are partitioned into $M/c \in \mathbb{N}$ clusters (for simplicity, we

[2]The results presented in Section 5.2 apply only to global scheduling.

assume that M is divisible by c) each containing c processors.[3] Each task is assigned to a cluster and can migrate only among the processors of that cluster. Thus, both partitioned and global scheduling are special cases ($c = 1$ and $c = M$, respectively).

Resource model. We consider a system that has a set $\{\ell_1, \ldots, \ell_{n_r}\}$ of shared resources. For now, we limit attention to mutex sharing, although other notions of sharing will be considered later. Under mutex sharing, a resource ℓ_q can be held by at most one job at any time. When a job $\tau_{i,j}$ requires a resource ℓ_q, it *issues* a *request* $\mathcal{R}$ for ℓ_q. $\mathcal{R}$ is *satisfied* as soon as $\tau_{i,j}$ *holds* ℓ_q, and *completes* when $\tau_{i,j}$ *releases* ℓ_q (see Figure 2.6). $\mathcal{R}$ is *active* from its issuance until its completion. Job $\tau_{i,j}$ is suspended until $\mathcal{R}$ can be satisfied. We assume that if a job $\tau_{i,j}$ holds a resource ℓ_q, then it must be scheduled to execute.[4] A resource access is called a *critical section* (CS).

We assume that each job can request or hold at most one resource at a time, *i.e.,* resource requests are non-nested. We let N_i^q denote the maximum number of times a job of task τ_i requests ℓ_q, and let L_i^q denote the maximum length of such a request. We define L_i^q as 0 if $N_i^q = 0$. Finally, we define $L_{max}^q = \max_{1 \leq i \leq n}\{L_i^q\}$ and $L_{max} = \max_{1 \leq q \leq n_r}\{L_{max}^q\}$.

Definition 5.1. We define $\mathcal{L}_i^q = [L_i^q, L_i^q, \ldots, L_i^q]$ to be a list that contains L_i^q exactly P_i times, and define $\mathcal{L}^q$ to contain all elements of $\mathcal{L}_1^q, \mathcal{L}_2^q, \ldots, \mathcal{L}_N^q$. Let $L_{sum,h}^q$ be the sum of the (up to) h largest elements of $\mathcal{L}^q$. ◀

Example 5.1. Assume that three tasks τ_1, τ_2, and τ_3 access a shared resource ℓ_q. Let $P_1 = 2, P_2 = 3$, and $P_3 = 1$. By Definition 5.1, $\mathcal{L}_1^q = [L_1^q, L_1^q]$, $\mathcal{L}_2^q = [L_2^q, L_2^q, L_2^q]$, and $\mathcal{L}_3^q = [L_3^q]$. Thus, $\mathcal{L}^q = [L_1^q, L_1^q, L_2^q, L_2^q, L_2^q, L_3^q]$. Assume that $L_1^q > L_2^q > L_3^q$. Then, $L_{sum,2}^q = 2L_1^q$ and $L_{sum,4}^q = 2L_1^q + 2L_2^q$. ◀

Eligible and ready jobs. We now provide a further refinement of jobs' readiness when they share resources and wait for an occupied resource by suspending.

Definition 5.2 (Eligible job)**.** A job $\tau_{i,j}$ is *eligible* at time t in a schedule $\mathcal{S}$ if and only if it is pending (see Definition 2.4) and $j \leq P_i$ or $f(\tau_{i,j-P_i}) \leq t$ hold. ◀

Definition 5.3 (Ready job)**.** A job $\tau_{i,j}$ is *ready* at time t in a schedule $\mathcal{S}$ if and only if it is eligible but not suspended. ◀

[3] Our results can be adapted for non-uniform cluster sizes at the expense of additional notation.

[4] As noted in Chapter 1, this is a common assumption in work on synchronization. It is needed for shared data, but may be pessimistic for other shared resources such as I/O devices.

Note that the definition of eligible jobs matches the definition of ready jobs in prior chapters where no suspension times were considered. We add the notion of eligibility to preserve the traditional semantics of ready jobs.

Pi-blocking. In this chapter, we consider accounting for pi-blocking that can arise due to mutex sharing under s-oblivious analysis. The formal definition of pi-blocking under s-oblivious analysis is given in Definition 2.2. This definition assumes that tasks have constrained deadlines and they are scheduled by a global scheduler. The refinement of the definition for clustered scheduling is trivial [Brandenburg and Anderson, 2011]. However, when tasks have soft real-time constraints or have deadlines larger than periods, further refinement is needed. This is to discount any self-dependency-related delays as s-oblivious pi-blocking time. Note that, under s-oblivious schedulability analysis (as in Chapters 3 and 4), self-dependency-related delays are explicitly handled but suspension times are not. We give the refined definition below.

Definition 5.4. Under s-oblivious schedulability analysis for clustered scheduling, a job $\tau_{i,j}$ incurs s-oblivious pi-blocking at time t if and only if $\tau_{i,j}$ is **eligible** but not scheduled and fewer than c higher-priority jobs are **eligible** in its cluster. ◀

Example 5.2. Figure 5.1 illustrates two consecutive jobs $\tau_{i,j}$ and $\tau_{i,j+1}$ of a task τ_i with $T_i = 7$, $D_i = 11$, and $P_i = 1$ (no parallel execution of two jobs of a task is allowed). Assume that task τ_i is scheduled on a cluster containing $c > 1$ processors. Job $\tau_{i,j+1}$ is released at time 7 and job $\tau_{i,j}$ finishes execution at time 10. Thus, job $\tau_{i,j+1}$ is pending but *not* eligible during the time interval $[7, 10)$. Assume that both $\tau_{i,j}$ and $\tau_{i,j+1}$ are among the c highest-priority pending jobs in their cluster during $[7, 10)$. However, since $P_i = 1$, $\tau_{i,j+1}$ is pending but not eligible during the interval $[7, 10)$. Thus, it is not s-oblivious pi-blocked during that interval according to Definition 5.4. In contrast, $\tau_{i,j+1}$ is eligible during $[12, 13)$. Assume that $\tau_{i,j+1}$ is among the c highest-priority eligible jobs during $[12, 13)$, but is suspended. Then, by Definition 5.4, $\tau_{i,j+1}$ is s-oblivious pi-blocked during $[12, 13)$. ◀

5.2 Lower-Bound Results for Non-FIFO Global JLFP Schedulers

In this section, we consider a class of global JLFP schedulers that contains G-EDF and G-FP schedulers but not G-FIFO schedulers. Under such a class of schedulers, we improve the existing lower bound of $M - 1$ request lengths on per-request pi-blocking time [Brandenburg and Anderson, 2010a]. For the purpose of

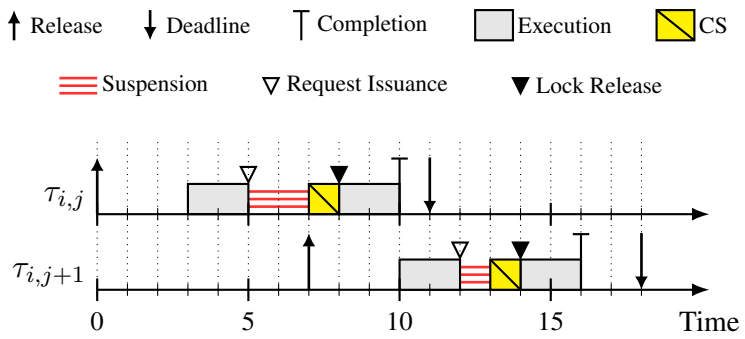


Figure 5.1: A schedule illustrating s-oblivious pi-blocking for arbitrary-deadline HRT tasks. These jobs are scheduled alongside jobs in other clusters that are not shown and cause lock-related suspensions.

deriving a lower bound, it is sufficient to assume that the system has only one mutex resource, *i.e.,* $n_r = 1$. Since there is one mutex resource, we do not use any resource index in the rest of this section, *e.g.,* the mutex resource is denoted by ℓ. In the rest of the section, we first give a general lower bound of $2M - 2$ request lengths on per-request pi-blocking (Section 5.2.1). Then, we consider a subclass of locking protocols for which the lower bound can be improved to a value that is one time unit smaller than $2M - 1$ request lengths (Section 5.2.2).

5.2.1 General Lower Bound on Pi-Blocking

In this section, we give a lower bound of $2M - 2$ request lengths on pi-blocking under a class of GEL schedulers that includes G-EDF and G-FP scheduling. Specifically, we demonstrate the existence of a task system and a corresponding release sequence such that a job incurs pi-blocking for at least $2M - 2$ request lengths. In the rest of the section, we first describe said task system (Section 5.2.1.1). Next, we show how a job in that system incurs pi-blocking for at least $2M - 2$ request lengths assuming a certain assignment of job priorities (Section 5.2.1.2). Finally, we show how that priority assignment can be realized under different schedulers (Section 5.2.1.3).

5.2.1.1 Task System

Let Γ be a set of N tasks that are globally scheduled on M processors. Γ consists of M disjoint groups of tasks. The first group of tasks consists of $2M - 2$ tasks $\{\tau_1^1, \tau_2^1, \ldots, \tau_{2M-2}^1\}$. Each of the remaining $M - 1$ groups of tasks consists of M tasks. We denote the set of tasks in the i^{th} group ($i > 1$) by $\{\tau_1^i, \tau_2^i, \ldots, \tau_M^i\}$. Thus, the total number of tasks is $N = (2M - 2) + (M - 1)M = M^2 + M - 2$.

Each job of each task issues a request for resource ℓ as soon as the job is released. The request length of each request is L. Each job completes as soon as its request for resource ℓ completes. Thus, $C_i = L$ holds. To establish our lower bound, we do not need any specific assignment of periods to the tasks in Γ, unless a period assignment is required to assign job priorities (as in G-EDF for implicit-deadline HRT tasks). We will show how periods can be assigned (if needed) in Section 5.2.1.3.

Feasibility of Γ. In the following lemma, we show that Γ can be feasibly scheduled under any JLFP scheduler when the minimum period and the minimum deadline of all tasks are large enough. Intuitively, all jobs can be scheduled sequentially yet meet deadlines.

Lemma 5.1. *If $T_{min} \geq NL$ and $D_{min} \geq NL$, then there exists a suspension-based locking protocol under which Γ is HRT-schedulable under any JLFP scheduler.*

Proof. We show that Γ is HRT-schedulable under any JLFP scheduler when lock requests are satisfied in FIFO order. We do so by showing that, in such a case, each job completes within NL time units after its release. Assume otherwise. Let τ be the job with the earliest release time, call it t_r, which does not complete execution within $t_r + NL$. Since no job released before t_r takes more than NL time units after its release to complete and $T_{min} \geq NL$, there is at most one pending job per task at time t_r. Thus, there are at most N pending jobs (including τ) at time t_r. Since requests are satisfied in FIFO order, τ's request is complete by the time these N requests are complete. Since JLFP scheduling is work-conserving, these N requests complete by time $t_r + NL$. Since each job finishes execution when its request completes, τ completes execution by time $t_r + NL$, reaching a contradiction. Since $D_{min} \geq NL$, each job meets its deadline. □

Release sequence. We now describe a release sequence (*i.e.,* instantiation) Γ_{seq} for tasks in Γ. Our lower-bound proof only requires one job of each task. For ease of notation, we omit the job index and use τ_i^j to denote both a task and its job. These jobs are released according to the following rules.

JR1. Jobs $\mathcal{J}^1 = \{\tau_1^1, \tau_2^1, \ldots, \tau_{2M-2}^1\}$ are released at time 0.

JR2. Let t_i be the time instant when the i^{th}-satisfied request is complete. We define $t_0 = 0$. At time t_{kM-1}, jobs $\mathcal{J}^{k+1} = \{\tau_1^{k+1}, \tau_2^{k+1}, \ldots, \tau_M^{k+1}\}$ are released.

JR3. No task releases a new job until all jobs in $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^M$ complete execution.

Note that Rules JR1–JR3 do not require an unsatisfied request to be satisfied immediately after a request completion. Thus, at time t_i, a locking protocol may insert delays before satisfying the next request. Since there is only one job per task (Rule JR3), no effect of parallelization levels of tasks can be observed for these jobs. Therefore, by Definitions 2.4 and 5.2, any pending job is also eligible. Thus, to determine whether a job is pi-blocked or not according to Definition 5.4, we will consider pending jobs.

Job priorities. We assume that job priorities satisfy the following rules. We will later illustrate how this priority ordering can be achieved under different schedulers in Section 5.2.1.3.

PR1. For any $v > w$, job τ_i^v has higher priority than job τ_j^w, *i.e.,* the jobs in $\mathcal{J}^v$ have higher priorities than the jobs in $\mathcal{J}^w$.

PR2. For any $i > j$, job τ_i^v has higher priority than job τ_j^v.

Example 5.3. Figure 5.2 depicts a release sequence according to Rules JR1–JR3 for $M = 4$. By Rule JR1, jobs τ_1^1–τ_6^1 are released at time 0. Since the $(M-1)^{st} = 3^{rd}$ satisfied request (τ_3^1's request) completes at time 3, by Rule JR2, jobs τ_1^2–τ_4^2 are released at time 3. Similarly, by Rule JR2, jobs τ_1^3–τ_4^3 are released at time 7, as the $(2M-1)^{st} = 7^{th}$ satisfied request (τ_3^2's request) completes then.

In Figure 5.2, the time intervals when a job experiences s-oblivious pi-blocking are marked red. For example, during $[0, 1)$, jobs τ_6^1, τ_4^1, and τ_3^1 suffer pi-blocking, as they are among the top $M = 4$ jobs by priority during this time interval (by Rule PR2). In contrast, a job does not experience pi-blocking during black-marked intervals. For example, jobs τ_1^1 and τ_2^1 are suspended but not pi-blocked during time interval $[0, 1)$. Note how their suspension time here is "negated" as pi-blocking time by the presence of $M = 4$ higher-priority jobs that are either executing or suspended. ◀

5.2.1.2 Lower-Bound Proof

In this section, we prove the following theorem.

Theorem 5.1. *There is a job τ_i^j in Γ_{seq} that incurs pi-blocking for at least $(2M-2)L$ time units when job priorities are determined by Rules PR1 and PR2.*

To prove Theorem 5.1, our goal is to show that there exists a time instant when $M-1$ jobs are pending with unsatisfied requests, and each such job has already incurred pi-blocking of at least ML time units. If no

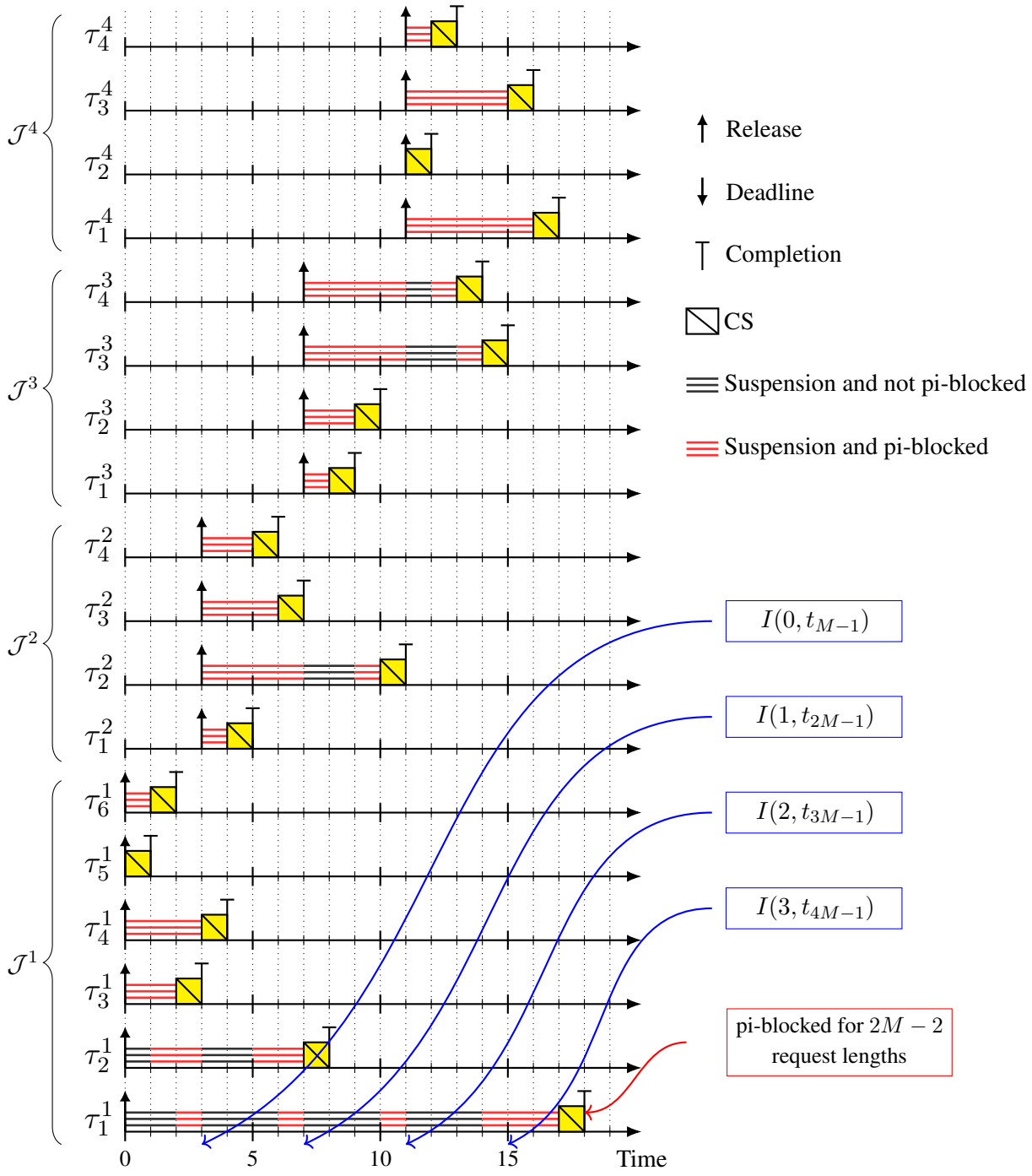


Figure 5.2: Release sequence by Rules JR1–JR3 for $M = 4$. Job priorities increase from bottom to top.

higher-priority jobs are released at or after such a time instant, then at least one job must incur pi-blocking for at least an additional $(M-2)L$ time units (as any locking protocol must impart some ordering of these requests). To prove this, we show that an invariant holds at times $t_{M-1}, t_{2M-1}, \ldots, t_{M^2-1}$. We first define some notation.

Definition 5.5. Let $B(x,t)$ denote the number of pending jobs at time t that have incurred pi-blocking of at least xL time units by that time. ◀

Using $B(x,t)$, we define the following predicate.

$$I(p,t) \equiv \bigwedge_{x=M,v=p}^{x=1,v=M+p-1} B(x,t) \geq \min\{v, M-1\} \tag{5.1}$$

Thus, $I(p,t)$ is true if and only if $B(M,t) \geq \min\{p, M-1\} \wedge B(M-1,t) \geq \min\{p+1, M-1\} \wedge \cdots \wedge B(1,t) \geq \min\{M+p-1, M-1\}$. This means that, at time t, there exist $\min\{p, M-1\}$ pending jobs that have incurred at least ML time units of pi-blocking, $\min\{p+1, M-1\}$ pending jobs that have incurred at least $(M-1)L$ time units of pi-blocking, and so on.

Example 5.3 (Continued)**.** Consider the schedule in Figure 5.2. At time 3, the only pending jobs that were released previously (and thus could have been pi-blocked) are τ_4^1, τ_2^1, and τ_1^1. These jobs have incurred pi-blocking for $3L, 2L$, and L time units, respectively. Thus, there is (are) one (resp., two, three) job(s) that has (have) incurred pi-blocking for $3L$ (resp., $2L$, L) time units by time 3 (and none that have experienced pi-blocking for $4L$ time units). Thus, $B(4,3) = 0, B(3,3) = 1, B(2,3) = 2$, and $B(1,3) = 3$ hold. Hence, $I(0,3)$ holds. Similarly, by time 7, τ_2^2, τ_2^1, and τ_1^1, which were all released previously and are still pending, have incurred pi-blocking for $4L, 4L$, and $2L$ time units, respectively. Thus, $B(4,7) = 2$, as there are two pending jobs at time 7 that have incurred pi-blocking for at least $4L$ time units by time 7. Similarly, $B(3,7) = 2$, $B(2,7) = 3$, and $B(1,7) = 3$ hold. Thus, we have $B(4,7) \geq \min\{1,3\} \wedge B(3,7) \geq \min\{2,3\} \wedge B(2,7) \geq \min\{3,3\} \wedge B(1,7) \geq \min\{4,3\}$. Since $M = 4$, by (5.1), $I(1,7)$ holds. ◀

To prove Theorem 5.1, our goal is to show that $I(M-1,t)$ holds at some time instant t in Γ_{seq} under any suspension-based locking protocol. By (5.1), this would imply that $B(M,t) \geq (M-1)$. Thus, there exists a time instant when $M-1$ pending jobs have already incurred ML time units of pi-blocking. In the following lemma, we first show that there exists a time instant t when $I(0,t)$ holds.

Lemma 5.2. *$I(0, t_{M-1})$ holds.*

Proof. Time instant $t_{M-1} = t_3$ in Figure 5.2 illustrates this lemma. By (5.1), we need to prove that $B(M, t_{M-1}) \geq 0 \wedge B(M-1, t_{M-1}) \geq 1 \wedge \cdots \wedge B(1, t_{M-1}) \geq M-1$ holds. Since no jobs in $\mathcal{J}^2 \cup$

$\mathcal{J}^3 \cup \cdots \cup \mathcal{J}^M$ are released before time t_{M-1}, only jobs in $\mathcal{J}^1$ can incur pi-blocking before time t_{M-1}. By Rule JR2, the first i satisfied requests and their corresponding jobs complete by time t_i. Thus, at time t_{M-1}, there are $2M - 2 - (M-1) = M - 1$ pending jobs of $\mathcal{J}^1$. Let $\tau_{hp(i)}$ be the i^{th} highest-priority pending job among the jobs of $\mathcal{J}^1$ at time t_{M-1}. We prove the lemma by first establishing the following claim.

Claim 5.1. *Job $\tau_{hp(i)}$ incurs pi-blocking for at least $(M-i)L$ time units by time t_{M-1}.*

Proof. We first show that job $\tau_{hp(i)}$ is one of the M highest-priority pending jobs during $[t_{i-1}, t_{M-1})$. Note that $i \leq M-1$ (hence, $i-1 < M-1$) holds, as there are $M-1$ pending jobs (including $\tau_{hp(i)}$) of $\mathcal{J}^1$ at time t_{M-1}. By the definition of $\tau_{hp(i)}$, among the $M-1$ pending jobs of $\mathcal{J}^1$ at time t_{M-1}, exactly $M-1-i$ jobs have lower priorities than $\tau_{hp(i)}$.

By Rules JR1 and JR2, no job is released during $(0, t_{M-1})$. Thus, by the definition of time instant t_{i-1}, there are $2M - 2 - (i-1) = 2M - i - 1$ pending jobs at time t_{i-1}. Since $\tau_{hp(i)}$ has higher priority than $M-i-1$ jobs of $\mathcal{J}^1$ at time t_{M-1}, $\tau_{hp(i)}$ is among the $2M - i - 1 - (M-i-1) = M$ highest-priority pending jobs at time t_{i-1}. $\tau_{hp(i)}$ remains one of the M highest-priority pending jobs during $[t_{i-1}, t_{M-1})$, as no jobs are released during $[t_{i-1}, t_{M-1})$.

During the time interval $[t_{i-1}, t_{M-1})$, the $i^{th}, (i+1)^{st}, \ldots, (M-1)^{st}$ satisfied requests are satisfied and complete. Thus, $t_{M-1} - t_{i-1} \geq (M - 1 - i + 1)L = (M-i)L$. Note that $t_M - t_{i-1}$ can be greater than $(M-i)L$ if a locking protocol inserts any delay between two consecutive satisfied requests. Since $\tau_{hp(i)}$ is pending at time t_{M-1} and it is among the top-M jobs by priority during $[t_{i-1}, t_{M-1})$, it is continuously pi-blocked during $[t_{i-1}, t_{M-1})$. Thus, $\tau_{hp(i)}$ incurs pi-blocking for at least $(M-i)L$ time units by time t_{M-1}. □

Continuing the proof of the lemma, we now show that, for any $i < M$, $B(M-i, t_{M-1}) \geq i$. Consider the set of jobs $\{\tau_{hp(1)}, \tau_{hp(2)}, \ldots, \tau_{hp(i)}\}$. By Claim 5.1, each job in $\{\tau_{hp(1)}, \tau_{hp(2)}, \ldots, \tau_{hp(i)}\}$ incurs at least $(M-i)L$ time units of pi-blocking by time t_{M-1}. Thus, $B(M-i, t_{M-1}) \geq i$ holds for each $i < M$. Moreover, $B(M, t_M) \geq 0$ holds trivially. Thus, $I(0, t_M) = \bigwedge_{x=M, v=0}^{x=1, v=M-1} B(x, t_{M-1}) \geq v$ holds. □

In the following three lemmas, we show that each job that is pending but unscheduled during time intervals $[t_{kM-2}, t_{kM-1})$ (for any $2 \leq k \leq M$) incurs pi-blocking during this time interval. This allows the system to steadily reach a time instant t when $I(M-1, t)$ holds. In Figure 5.2, the time intervals

$[t_{kM-2}, t_{kM-1})$ refer to time intervals $[6, 7)$, $[10, 11)$, and $[14, 15)$. Note that, during each of these intervals, exactly $M = 4$ jobs are pending.

Lemma 5.3. *For any integer $2 \leq k \leq M$, there are M pending jobs during time interval $[t_{kM-2}, t_{kM-1})$.*

Proof. By Rule JR1, $2M - 2$ jobs are released at time 0. By Rule JR2, for each time instant t_{iM-1} with $1 \leq i \leq M - 1$, M jobs are released. By Rules JR1 and JR2, at or before time t_{kM-2}, jobs are released only at time instants $0, t_{M-1}, t_{2M-1}, \ldots, t_{(k-1)M-1}$. Thus, the number of jobs that are released at or before time t_{kM-2} is $2M - 2 + (k - 1)M = (k + 1)M - 2$. By the definition of time instant t_{kM-2}, the first $kM - 2$ satisfied requests are complete and these request-issuing jobs finish execution by time t_{kM-2}. Thus, the number of pending jobs at time t_{kM-2} is $(k + 1)M - 2 - kM + 2 = M$. Since no job is released during $[t_{kM-2}, t_{kM-1})$, the lemma follows. □

Lemma 5.4. *For any integer $2 \leq k \leq M - 1$, there are $M - 1$ pending jobs of $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^k$ at time t_{kM-1}.*

Proof. By Rules JR1 and JR2, only jobs in $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^k$ are released at or before time t_{kM-2}. By Lemma 5.3, there are M pending jobs during time interval $[t_{kM-2}, t_{kM-1})$. By the definition of time instants t_{kM-2} and t_{kM-1}, a request completes and the request-issuing job finishes execution at time t_{kM-1}. Thus, there are $M - 1$ pending jobs of $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^k$ at time t_{kM-1}. □

Lemma 5.5. *For any integer $2 \leq k \leq M$, all but one job that is pending at time t_{kM-2} incurs pi-blocking throughout the time interval $[t_{kM-2}, t_{kM-1})$.*

Proof. By Rule JR2, no jobs in $\mathcal{J}^i$ with $i > k$ are released at or before time t_{kM-2}. By Lemma 5.3, there are M pending jobs during the time interval $[t_{kM-2}, t_{kM-1})$. Among these M jobs, the job whose request completes at time t_{kM-1} must be scheduled during $[t_{kM-2}, t_{kM-1})$. Thus, each of the remaining $M - 1$ pending jobs incurs pi-blocking throughout $[t_{kM-2}, t_{kM-1})$. □

Using Lemma 5.5, we now prove the following lemma. Informally, since $M - 1$ jobs incur pi-blocking during time interval $[t_{kM-2}, t_{kM-1})$, the number of pending jobs at time t_{kM-1} that have already incurred at least L units of pi-blocking is at least $M - 1$.

Lemma 5.6. *For any integer $2 \leq k \leq M$, $B(1, t_{kM-1}) \geq M - 1$.*

Proof. By Lemma 5.3, there are M pending jobs during $[t_{kM-2}, t_{kM-1})$. By Lemma 5.5, $M-1$ jobs among these M pending jobs incur pi-blocking throughout $[t_{kM-2}, t_{kM-1})$. By the definition of time instants t_i, $t_{kM-1} - t_{kM-2} \geq L$. Thus, there are at least $M-1$ pending jobs at time t_{kM-1} that have incurred pi-blocking for the duration of at least L time units, and the lemma follows. □

We now show that there exist time instants t such that $I(k,t)$ holds for each $k \leq M-1$. In Figure 5.2, these time instants are times $3, 7, 11$, and 15 when $I(0,3), I(1,7), I(2,11)$, and $I(3,15)$ hold, respectively. Specifically, we show that if $I(k-2, t_{(k-1)M-1})$ holds, then $I(k-1, t_{kM-1})$ will also hold.

Lemma 5.7. *For any integer* $1 \leq k \leq M$, $I(k-1, t_{kM-1})$ *holds.*

Proof. We use Figure 5.3 to illustrate the proof. By Lemma 5.2, $I(0, t_{M-1})$ holds. We thus prove the lemma for $k \geq 2$. Assume for a contradiction that $p \geq 2$ is the smallest integer for which $I(p-1, t_{pM-1})$ does not hold. Thus, by the definition of p, we have

$$I(p-2, t_{(p-1)M-1}) \wedge \neg I(p-1, t_{pM-1}). \tag{5.2}$$

Since $\neg I(p-1, t_{pM-1})$ holds, by (5.1), we have

$$\bigvee_{x=M, v=p-1}^{x=1, v=M+p-2} B(x, t_{pM-1}) < \min\{v, M-1\}. \tag{5.3}$$

Note that the index x decreases from M to 1 in (5.3). Assume that $M-q$ (with $0 \leq q \leq M-1$) is the largest index x in (5.3) such that $B(x, t_{pM-1}) < \min\{v, M-1\}$ holds. In (5.3), the index v equals $p-1$ when x equals M, and v increases as x decreases. Thus, when $x = M-q$, we have $v = p-1+q$. Therefore,

$$B(M-q, t_{pM-1}) < \min\{p-1+q, M-1\}. \tag{5.4}$$

To reach a contradiction, we will show that (5.4) cannot hold by considering the requests that are satisfied during $[t_{(p-1)M-1}, t_{pM-1})$. By Rule JR2, M jobs of $\mathcal{J}^p$ are released at time $t_{(p-1)M-1}$ (see Figure 5.3). By Lemma 5.3, there are M pending jobs of $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^{p-1}$ during $[t_{(p-1)M-2}, t_{(p-1)M-1})$. By the definition of time instant $t_{(p-1)M-1}$, one of these M pending jobs during $[t_{(p-1)M-2}, t_{(p-1)M-1})$ completes at time $t_{(p-1)M-1}$. Thus, at time $t_{(p-1)M-1}$, there are $M-1$ pending jobs of $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^{p-1}$. Let $\mathcal{J}_{old}$

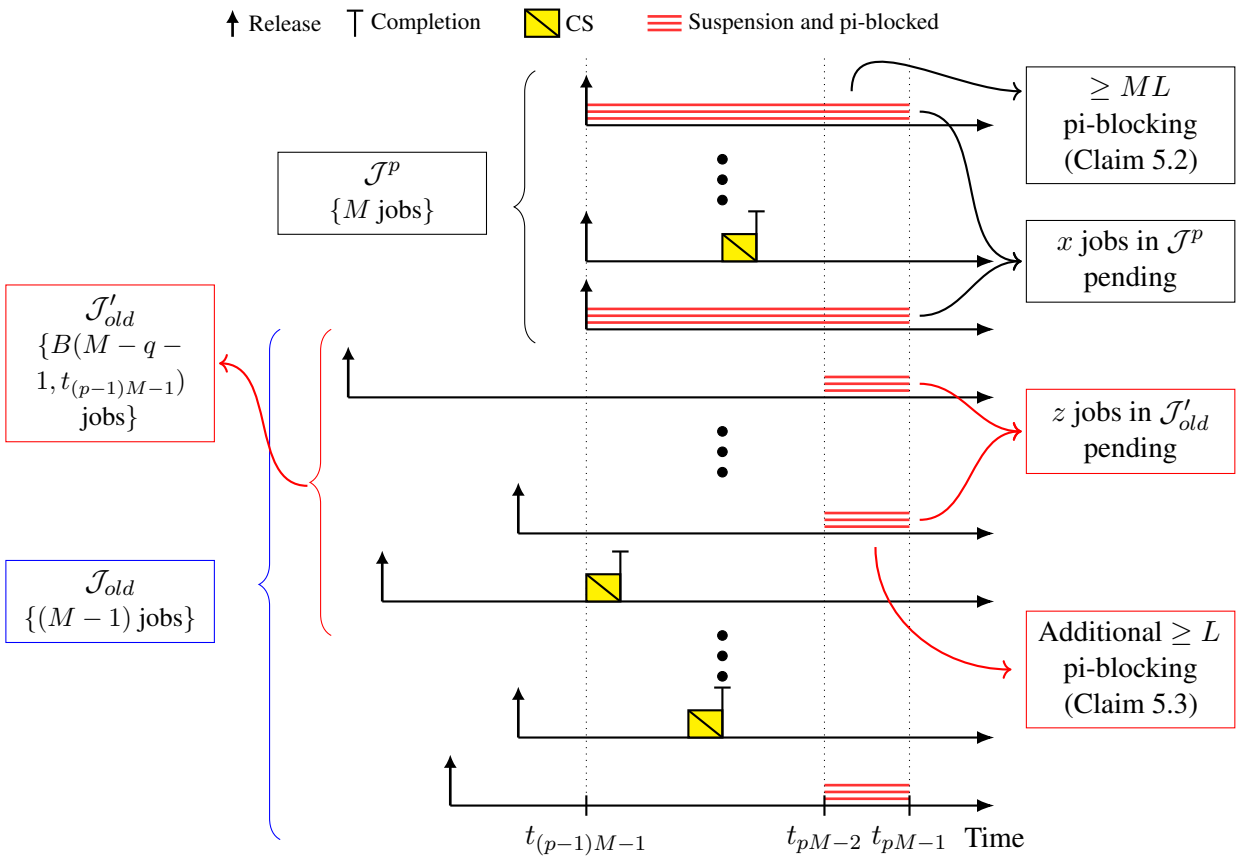


Figure 5.3: Illustration of the proof of Lemma 5.7.

be the set of these $M-1$ jobs of $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^{p-1}$ that are pending at time $t_{(p-1)M-1}$ (see Figure 5.3). We now consider two cases depending on the value of $M-q$.

Case 1. $M-q=1$. Replacing $q=M-1$ in (5.4), we have $B(1, t_{pM-1}) < \min\{p-1+M-1, M-1\}$. Since $p \geq 2$, we have $p-1+M-1 \geq 2-1+M-1 = M$. Thus, $B(1, t_{pM-1}) < \min\{M, M-1\} = M-1$, contradicting Lemma 5.6.

Case 2. $M-q>1$. Thus, $M-q-1 \geq 1$ holds. By Definition 5.5, there are $B(M-q-1, t_{(p-1)M-1})$ pending jobs at time $t_{(p-1)M-1}$ that have incurred pi-blocking for the duration of $(M-q-1)L \geq L$ time units. Thus, each of these $B(M-q-1, t_{(p-1)M-1})$ jobs is from $\mathcal{J}_{old}$, as they must be released before time $t_{(p-1)M-1}$. Let $\mathcal{J}'_{old} \subseteq \mathcal{J}_{old}$ be these $B(M-q-1, t_{(p-1)M-1})$ jobs (see Figure 5.3).

We now lower bound the number of jobs in $\mathcal{J}^p \cup \mathcal{J}'_{old}$. Since $\mathcal{J}^p$ and $\mathcal{J}'_{old}$ are disjoint, we have $|\mathcal{J}^p| + |\mathcal{J}'_{old}| = M + B(M-q-1, t_{(p-1)M-1})$. Thus, to lower bound $|\mathcal{J}^p| + |\mathcal{J}'_{old}|$, we derive a lower bound on $B(M-q-1, t_{(p-1)M-1})$ using $I(p-2, t_{(p-1)M-1})$ (which holds by (5.2)). Since $M-q-1 \geq 1$ and $I(p-2, t_{(p-1)M-1})$ hold, by (5.1), we have $B(M-q-1, t_{(p-1)M-1}) \geq \min\{p-2+q+1, M-1\}$.

Using this lower bound on $B(M-q-1, t_{(p-1)M-1})$, we get

$$|\mathcal{J}^p| + |\mathcal{J}'_{old}| = M + B(M-q-1, t_{(p-1)M-1}) \geq M + \min\{p+q-1, M-1\}. \tag{5.5}$$

By Lemma 5.4, there are $M-1$ pending jobs of $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^p$ at time t_{pM-1}. Thus, $M-1$ jobs in $\mathcal{J}_{old} \cup \mathcal{J}^p$ are pending at time t_{pM-1}. Assume that, at time t_{pM-1}, among these $M-1$ pending jobs of $\mathcal{J}_{old} \cup \mathcal{J}^p$, x jobs are from $\mathcal{J}^p$, z jobs are from $\mathcal{J}'_{old}$, and the remaining $M-1-x-z$ jobs are from $\mathcal{J}_{old} \setminus \mathcal{J}'_{old}$ (see Figure 5.3). Thus, $|\mathcal{J}^p| - x$ jobs from $\mathcal{J}^p$, and $|\mathcal{J}'_{old}| - z$ jobs from $\mathcal{J}'_{old}$ complete execution by time t_{pM-1}. Since (by Rule JR2) a total of M jobs complete execution during $[t_{(p-1)M-1}, t_{pM-1})$, we have

$$|\mathcal{J}^p| - x + |\mathcal{J}'_{old}| - z \leq M,$$

which implies

$$\begin{aligned} x + z &\geq |\mathcal{J}^p| + |\mathcal{J}'_{old}| - M \\ &\geq \{\text{By (5.5)}\} \\ &\quad M + \min\{p+q-1, M-1\} - M \\ &= \min\{p+q-1, M-1\}. \end{aligned} \tag{5.6}$$

We now show that these $x+z$ pending jobs of $\mathcal{J}^p \cup \mathcal{J}'_{old}$ incur at least $(M-q)L$ time units of pi-blocking by time t_{pM-1}. By Definition 5.5 and (5.6), this implies that $B(M-q, t_{pM-1}) \geq x+z \geq \min\{p+q-1, M-1\}$, contradicting (5.4). The following two claims help to establish this.

Claim 5.2. *Each of the x jobs of $\mathcal{J}^p$ that are pending at time t_{pM-1} incurs pi-blocking for at least ML time units by time t_{pM-1}.*

Proof. Let τ be one of the x jobs of $\mathcal{J}^p$ that are pending at time t_{pM-1}. Since $J \in \mathcal{J}^p$ and $\mathcal{J}_{old} \subseteq (\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^{(p-1)})$, by Rule PR1, τ has higher priority than each job in $\mathcal{J}_{old}$. Since only jobs in $\mathcal{J}^p \cup \mathcal{J}_{old}$ are pending during $[t_{(p-1)M-1}, t_{pM-1})$ and $|\mathcal{J}^p| = M$ (by Rule JR2), τ is among the top-M pending jobs by priority throughout $[t_{(p-1)M-1}, t_{pM-1})$. Thus, τ incurs pi-blocking throughout $[t_{(p-1)M-1}, t_{pM-1})$. During $[t_{(p-1)M-1}, t_{pM-1})$, M requests complete execution. Thus, τ incurs pi-blocking for at least ML time units. □

Claim 5.3. *Each of the z jobs of $\mathcal{J}'_{old}$ that are pending at time t_{pM-1} incurs pi-blocking for at least L time units during time interval $[t_{(p-1)M-1}, t_{pM-1})$.*

Proof. By Lemma 5.5, each of the z jobs of $\mathcal{J}'_{old}$ that are pending at time t_{pM-1} incurs pi-blocking throughout the time interval $[t_{pM-2}, t_{pM-1})$. By the definition of t_i, $t_{pM-1} - t_{pM-2} \geq L$. Thus, the claim holds. □

By the definition of $\mathcal{J}'_{old}$, each of the z pending jobs of $\mathcal{J}'_{old}$ has incurred at least $(M-q-1)L$ time units of pi-blocking by time $t_{(p-1)M-1}$. By Claim 5.3, each such job incurs pi-blocking for at least L time units during $[t_{(p-1)M-1}, t_{pM-1})$. Thus, these z jobs incur pi-blocking for at least $(M-q)L$ time units by time t_{pM-1}. By Claim 5.2, each of the x jobs of $\mathcal{J}^p$ that are pending at time t_{pM-1} incurs at least $ML \geq (M-q)L$ time units of pi-blocking by time t_{pM-1}. Thus, at time t_{pM-1}, there are $x+z$ pending jobs that have incurred pi-blocking for at least $(M-q)L$ time units. Therefore, by (5.6), $B(M-q, t_{pM-1}) \geq x+z \geq \min\{p+q-1, M-1\}$, contradicting (5.4).

Thus, in both cases, the lemma holds. □

We now prove Theorem 5.1.

Proof of Theorem 5.1. By Lemma 5.7, $I(M-1, t_{M^2-1})$ holds. Thus, by (5.1), at time t_{M^2-1}, there are $M-1$ pending jobs that have incurred pi-blocking for at least ML time units. By Lemma 5.4, there are $M-1$ pending jobs of $\mathcal{J}^1 \cup \mathcal{J}^2 \cup \cdots \cup \mathcal{J}^M$ at time t_{M^2-1}. By Rule JR3, no new job is released before these $M-1$ jobs are complete. Thus, each of these pending jobs incurs pi-blocking until its request is satisfied. Since any locking protocol must satisfy these $M-1$ pending jobs' requests in some order, the job whose request is satisfied last incurs at least an additional $(M-2)L$ time units of pi-blocking. Therefore, there exists a job that incurs pi-blocking for at least $ML + (M-2)L = (2M-2)L$ time units. □

5.2.1.3 Job Priority Assignment

In this section, we show how the lower-bound proof in Section 5.2.1.2 applies under different schedulers. We do so by showing how jobs can be assigned priorities under these schedulers so that Rules PR1 and PR2 hold.

G-FP schedulers. The following theorem shows that the lower-bound proof in Section 5.2.1.2 applies to any G-FP scheduler.

Theorem 5.2. *A job in Γ_{seq} incurs pi-blocking for at least $(2M-2)$ request lengths under any G-FP scheduler.*

Proof. Consider a G-FP scheduler $\mathcal{F}$. We re-index the tasks in Γ based on the task priority assignment under $\mathcal{F}$. For each $v > w$, τ_i^v has higher priority than τ_j^w. Also, for each $i > j$, τ_i^v has higher priority than τ_j^v. Thus, job priorities under $\mathcal{F}$ satisfy Rules PR1 and PR2. The theorem follows from Theorem 5.1. □

GEL schedulers. We now show that the lower-bound proof in Section 5.2.1.2 also applies under a class of GEL schedulers. Recall from previous chapters that, under GEL scheduling, each task τ_i^v has a relative priority point (RPP) Y_i^v. A job τ_i^v's *priority point* (PP) is defined as

$$y(\tau_i^v) = r(\tau_i^v) + Y_i^v. \tag{5.7}$$

We show that the lower-bound proof in Section 5.2.1.2 applies under GEL schedulers that assign RPPs to tasks in Γ satisfying the following constraints.

$$\forall \tau_M^v : 2 \leq v \leq M-1 :: Y_M^v > Y_1^{v+1} + (2M-2)L \tag{5.8}$$

$$\forall \tau_i^v : 2 \leq v \leq M-1 \wedge 1 \leq i \leq M-1 :: Y_i^v > Y_{i+1}^v \tag{5.9}$$

$$Y_{2M-2}^1 > Y_1^2 + (2M-2)L \tag{5.10}$$

$$\forall \tau_i^1 : 1 \leq i \leq 2M-3 :: Y_i^1 > Y_{i+1}^1 \tag{5.11}$$

Theorem 5.3. *A job in Γ_{seq} incurs pi-blocking for at least $(2M-2)$ request lengths under any GEL scheduler that satisfies* (5.8)–(5.11).

Proof. Assume that each job in Γ_{seq} incurs pi-blocking for less than $(2M-2)L$ time units under a GEL scheduler satisfying (5.8)–(5.11). We first prove the following claim.

Claim 5.4. *For $i \geq 0$, $t_{i+1} - t_i < (2M-2)L$.*

Proof. By the definitions of t_{i+1} and t_i, a request is satisfied and complete during time interval $[t_i, t_{i+1})$. If $t_{i+1} - t_i \geq (2M-2)L$ holds, then a pending job during time interval $[t_i, t_{i+1})$ incurs pi-blocking for at least $(2M-2)L$ time units, a contradiction. □

Using Claim 5.4, we show that the jobs in Γ_{seq} satisfy Rules PR1 and PR2. We first show that PR2 holds. We consider two jobs τ_i^v and τ_j^w, and show that they satisfy Rules PR1 and PR2.

Case 1. *Both τ_i^v and τ_j^w are from the same group, i.e., $v = w$.* Without loss of generality, assume that $i > j$. By (5.7) and Rules JR1 and JR2 (both jobs are released concurrently), we have $y(\tau_i^v) = r(\tau_i^v) + Y_i^v = r(\tau_j^v) + Y_i^v$. Since $i > j$ holds, by (5.9) and (5.11), we have $Y_i^v < Y_j^v$. Thus, $y(\tau_i^v) < r(\tau_j^v) + Y_j^v = y_j^v$. Thus, τ_i^v has an earlier PP (hence, higher priority) than τ_j^v, satisfying PR2.

Case 2: *Both τ_i^v and τ_j^w are from different group, i.e., $v \neq w$.* Without loss of generality, assume that $w > v$. We first consider the sub-case $2 \leq v \leq M-1$. By (5.7), we have

$$
\begin{aligned}
y(\tau_j^w) &= r(\tau_j^w) + Y_j^w \\
&\leq \{\text{By Rule JR2 and } w > v\text{, jobs in } \mathcal{J}^w \text{ are released no earlier than } \mathcal{J}^{v+1}\} \\
&\quad r(\tau_j^{v+1}) + Y_j^w \\
&= \{\text{By Rule JR2}\} \\
&\quad t_{vM-1} + Y_j^w \\
&= t_{vM-1} - t_{(v-1)M-1} + t_{(v-1)M-1} + Y_j^w \\
&< \{\text{By Claim 5.4}\} \\
&\quad (2M-2)L + t_{(v-1)M-1} + Y_j^w \\
&< \{\text{By Rule JR2 and (5.9)}\} \\
&\quad (2M-2)L + r(\tau_i^v) + Y_1^w \\
&< \{\text{By (5.8)}\} \\
&\quad (2M-2)L + r(\tau_i^v) + Y_M^v - (2M-2)L \\
&< \{\text{By (5.9)}\} \\
&\quad r(\tau_i^v) + Y_i^v \\
&= \{\text{By (5.7)}\} \\
&\quad y(\tau_i^v).
\end{aligned}
$$

Thus, job τ_j^w has an earlier PP (hence, higher priority) than job τ_i^v where $2 \leq v \leq M-1$. Similarly, using (5.10) and (5.11) in the above calculation, we can show that τ_j^w with $w > 1$ has higher priority than τ_i^1. Thus, both Rules PR1 and PR2 are satisfied. Therefore, By Theorem 5.1, there is a job that is pi-blocked for at least $(2M-2)$ request lengths, a contradiction. □

For G-EDF scheduling of arbitrary-deadline tasks, the lower-bound result holds by assigning task deadlines following constraints in (5.8)–(5.11) and ensuring that $D_{min} \geq NL$ (to satisfy Lemma 5.1). The lower-bound result holds even for constrained-deadline systems by considering systems with sufficiently large periods and deadlines satisfying (5.8)–(5.11). Finally, the following corollary shows that the bound holds for implicit-deadline systems too.

Corollary 5.1. *A job in Γ_{seq} incurs pi-blocking for at least $(2M-2)$ request lengths under G-EDF scheduling when tasks have implicit deadlines.*

Proof. We let $T_M^M \geq (M^2+M-2)L = NL$. We assign periods to each task so that the constraints (5.8)–(5.11) are satisfied. Note that $T_M^M = T_{min}$ holds. Thus, by Lemma 5.1, Γ is feasible. Since $Y_i^u = T_i^u$ holds under G-EDF, by Theorem 5.3, the corollary holds. □

The case of G-FIFO scheduling. Our lower-bound proof heavily relies on being able to release M jobs that have higher priorities than any earlier-released jobs (Rule JR2 and PR1). The proof does not apply to any scheduler that prevents higher-priority job releases. G-FIFO, which is a GEL scheduler, is one such example as it prioritizes jobs by their release times (a future job cannot have an earlier release time).

Request vs. release blocking. Recall from Section 2.3 that a job may experience both release blocking and request blocking under a locking protocol. Also, recall from Table 2.4 that the C-OMLP achieves a per-job pi-blocking bound of $(2M-1)L$ time units through a combination of a request-blocking bound of $(M-1)L$ time units and a release-blocking bound of ML time units. By the construction of Γ and Γ_{seq}, it may appear that the lower bound of $(2M-2)L$ time units of pi-blocking applies only for request blocking (as each job issues a request as soon as it is released), contradicting the C-OMLP's request-blocking bound. However, our bound actually applies for the total per-job pi-blocking (the sum of request and release blocking). A locking protocol like the C-OMLP may choose to decompose the worst-case per-job pi-blocking of $(2M-2)$ request lengths into separate release blocking and request blocking terms.

5.2.2 Improved Lower Bound Under An Additional Assumption

In this section, we improve the lower bound established in Section 5.2.1 for a class of locking protocols that satisfy a certain property. We begin by introducing some terms that we use to define this class of locking protocols.

Definition 5.6. Consider a job τ that issues a request $\mathcal{R}$ at time $t_a(\mathcal{R})$ that is satisfied at time $t_s(\mathcal{R})$. Define $t_h(\mathcal{R})$ as follows: if τ ever becomes one of the M highest-priority pending jobs in $[t_a(\mathcal{R}), t_s(\mathcal{R}))$, then let $t_h(\mathcal{R})$ denote the first such time; otherwise, let $t_h(\mathcal{R}) = \infty$. ◀

Using the above definition, we define *reorder-bounded* locking protocols.

Definition 5.7. Let τ_i and τ_j be two jobs that issue request $\mathcal{R}_i$ and $\mathcal{R}_j$, respectively. A *reorder-bounded* locking protocol decides the order in which $\mathcal{R}_i$ and $\mathcal{R}_j$ are satisfied (relative to each other) no later than time $\max\{t_h(\mathcal{R}_i), t_h(\mathcal{R}_j)\}$. ◀

In the rest of this section, we limit attention to locking protocols that are reorder-bounded. We now show that there exists a task system and a corresponding release sequence such that the pi-blocking incurred by a job can be one unit smaller than $(2M - 1)$ request lengths. We organize the proof in a similar structure as in Section 5.2.1.

5.2.2.1 Task System

Let $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$ be a set of N tasks that are scheduled on M processors. Each job of each task issues a request of length L for a resource ℓ as soon as it is released. We assume $L \geq 2$. Each job completes as soon as its request for resource ℓ completes. Thus, $C_i = L$. We assume that $N = (2M-1) + (2M-1)L$. Below, we show that Γ is HRT-schedulable under any JLFP scheduler when the minimum period T_{min} and relative deadlines are large enough. Intuitively, for sufficiently large task periods and deadlines, all jobs can execute sequentially yet meet deadlines.

Lemma 5.8. *If $T_{min} \geq NL$ and $D_{min} \geq NL$, then there exists a suspension-based locking protocol under which Γ is HRT-schedulable under any JLFP scheduler.*

Proof. Similar to Lemma 5.1. □

Release sequence for Γ. We now give a release sequence Γ_{seq} for Γ. We only give the release time for one job of each task τ_i. For notational convenience, we also denote the job by τ_i. We denote job τ_i's request by $\mathcal{R}_i$. These jobs are released according to the following rules. (An example release sequence is given in Figure 5.4, which we cover in detail below.)

SR1. Jobs $\{\tau_1, \tau_2, \cdots, \tau_M\}$ are released at time 0. Without loss of generality, assume that, $\mathcal{R}_1$ is the last satisfied request among $\{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_M\}$.

SR2. Let t_k be the time instant when the k^{th}-satisfied request is satisfied. Job τ_{M+k} is released at time $t_k + 1$ if *one* of the following conditions is met.

SR2.1. $k = 1$ holds.

SR2.2. $1 < k \leq N - M$ holds and $\mathcal{R}_{M+k-1}$ is ordered (hence, satisfied) before $\mathcal{R}_1$.

SR3. Let $h = \min\{N - M - k, M - 1\}$. Jobs $\{\tau_{M+k+1}, \tau_{M+k+2}, \cdots, \tau_{M+k+h}\}$ are released at time $t_k + 1$ if $\mathcal{R}_{M+k}$ is ordered (hence, satisfied) after $\mathcal{R}_1$.

SR4. If some jobs are released by Rule SR3, then no task releases a new job until all jobs released by Rule SR3 complete execution.

Note that Rules SR2 and SR3 require that, when a job τ_i with $i > M$ is released, the order in which the prior job τ_{i-1}'s request will be satisfied with respect to $\mathcal{R}_1$ is known. In Lemma 5.10, we will show that this ordering has already been finalized when τ_{i-1} is released under any reorder-bounded locking protocol. Finally, when a job τ_{M+k} is released whose request is ordered after $\mathcal{R}_1$, h new jobs are instantaneously released at the same time by Rule SR3. For simplicity, we assume such instantaneous releases are possible. This assumption can be removed by delaying the release of the h new jobs by a time unit.

Job priorities. We assume job priorities satisfy the following rule. In Section 5.2.2.3, we will describe how such priorities can be assigned under different schedulers.

QR. For any $i > j$, job τ_i has higher priority than job τ_j.

Example 5.4. Figure 5.4 depicts a release sequence according to Rules SR1–SR4 for $M = 4$ and $L = 3$. By Rule SR1, jobs τ_1–τ_4 are released at time 0. At time 0, τ_2's request is satisfied. Thus, $t_1 = 0$ holds. At

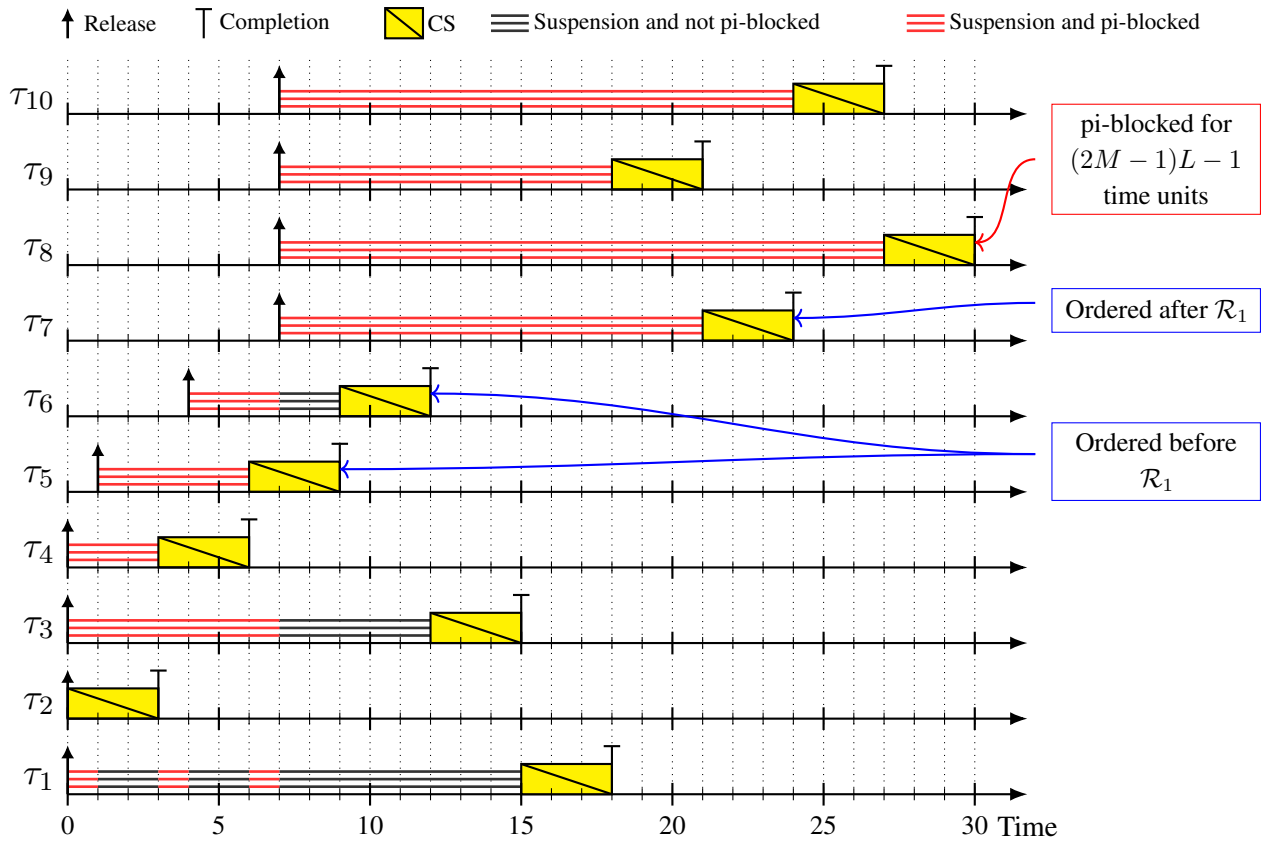


Figure 5.4: Release sequence by Rules SR1–SR4 for $M = 4$ and $L = 3$.

time $t_1 + 1 = 1$, by Rule SR2, job $\tau_{M+1} = \tau_5$ is released. $t_2 = 3$ holds because the second request to be satisfied (τ_4's request) is satisfied at time 3. Assuming τ_5's request is ordered before $\mathcal{R}_1$, by Rule SR2, τ_6 is released at time $t_2 + 1 = 4$. At time 7, τ_7 is released. Job τ_7's request is ordered after $\mathcal{R}_1$. Thus, by Rule SR3, τ_8–τ_{10} are released at time 7.

In Figure 5.4, the time intervals when a job experiences s-oblivious pi-blocking are marked red. During time interval $[0, 1)$, τ_1 is one of the top $M = 4$ jobs by priority. Thus, it experiences pi-blocking during this interval. However, due to the release of τ_5 and Rule QR, τ_1 is not one of the top $M = 4$ jobs by priority during time interval $[2, 3)$. Thus, it does not experience pi-blocking during this interval. ◀

5.2.2.2 Lower-Bound Proof

In this section, we prove the following theorem.

Theorem 5.4. *Under a reorder-bounded locking protocol, there exists a job in Γ_{seq} that incurs pi-blocking for at least $(2M-1)L-1$ time units when job priorities satisfy Rule QR.*

We prove Theorem 5.4 using the following two lemmas.

Lemma 5.9. *If no job is released by Rule SR3 at or before time t_i where $1 \leq i \leq N-M+1$, then there are M pending jobs at time t_i.*

Proof. We first determine the number of jobs released at or before time t_i. By Rule SR1, M jobs are released at time 0. By Rule SR2, only job τ_{M+k-1} is released during $[t_{k-1}, t_k)$ for all $2 \leq k \leq i$. Note that τ_{M+k-1} is valid because $M+k-1 \leq M+i-1 \leq M+N-M+1-1 = N$. Thus, the number of jobs released by time t_i is $M+i-1$. By the definition of t_i (Rule SR2), the i^{th}-satisfied request is satisfied but not complete at time t_i. Thus, exactly $i-1$ jobs complete execution by time t_i. Therefore, the number of pending jobs at time t_i is $M+i-1-(i-1) = M$. □

Lemma 5.10. *The relative order in which each request $\mathcal{R}_i$ (with $i > 1$) is satisfied with respect to $\mathcal{R}_1$ is determined when $\mathcal{R}_i$ is issued.*

Proof. By Rule SR1, each request $\mathcal{R}_i$ with $1 \leq i \leq M$ is issued at time 0 when τ_i is one of the top-M jobs by priority. Thus, $t_h(\mathcal{R}_i) = t_a(\mathcal{R}_i) = 0 \geq t_a(\mathcal{R}_1)$ for each $1 \leq i \leq M$. By Rule QR, job τ_i has higher priority than any job τ_j with $j < i$. By Rules SR2 and SR3, at most M jobs are released at any time. Thus, for any job τ_i with $i > M$, $t_h(\mathcal{R}_i) = t_a(\mathcal{R}_i) > 0 = t_a(\mathcal{R}_1) = t_h(\mathcal{R}_1)$ holds. Therefore, for any i, $t_a(\mathcal{R}_i) = \max\{t_h(\mathcal{R}_i), t_h(\mathcal{R}_1)\}$. By Definition 5.7, the relative order in which $\mathcal{R}_i$ is satisfied with respect to $\mathcal{R}_1$ is determined at time $t_a(\mathcal{R}_i)$. □

We now prove Theorem 5.4.

Proof of Theorem 5.4. For a contradiction, we assume the following.

Assumption 5.1. Each job in Γ_{seq} incurs pi-blocking for less than $(2M-1)L-1$ time units.

In the following claim, we show that a job must exist whose request is satisfied later than $\mathcal{R}_1$ to satisfy Assumption 5.1. Consequently, when such a job is released, h new jobs are also released by Rule SR3.

Claim 5.5. *There exists a request $\mathcal{R}_{M+q}$ with $1 \leq q \leq N-2M+1$ that is satisfied after $\mathcal{R}_1$.*

Proof. Assume that each request $\mathcal{R}_{M+q}$ with $1 \leq q \leq N - 2M + 1$ is satisfied before $\mathcal{R}_1$. By Rule SR1, $\mathcal{R}_1$ is the last satisfied request among $\{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_M\}$. Thus, $\mathcal{R}_1$ is the last satisfied request among all requests in $\{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_{M+N-2M+1}\}$. Hence, by the definition of time instant t_i (Rule SR2), $\mathcal{R}_1$ is not satisfied before time $t_{M+N-2M+1} = t_{N-M+1}$.

Since each request $\mathcal{R}_{M+q}$ with $1 \leq q \leq N - 2M + 1$ is satisfied before $\mathcal{R}_1$, no job τ_{M+q} with $1 \leq q \leq N - 2M + 1$ is released by Rule SR3. Consider any time instant t_q with $1 \leq q \leq N - 2M + 1$. By Lemma 5.9, there are M pending jobs at time t_q. By Rule SR2, no job is released during $(t_q, t_q + 1)$. Thus, the number of pending jobs throughout $[t_q, t_q + 1)$ is M. Since $\mathcal{R}_1$ is not satisfied before time t_{N-M+1}, τ_1 is pending and pi-blocked during all intervals $[t_q, t_q + 1)$ with $1 \leq q \leq N - 2M + 1$. Thus, τ_1 incurs pi-blocking for at least $(N - 2M + 1) \cdot 1 = (2M - 1) + (2M - 1)L - 2M + 1 = (2M - 1)L$ time units, contradicting Assumption 5.1. □

Let $\mathcal{R}_{M+q}$ be the request with smallest $(M+q)$ value that is ordered after $\mathcal{R}_1$ (τ_7's request in Figure 5.4). Claim 5.5 guarantees the existence of such request. Therefore, the following holds.

Property 5.1. *Each request $\mathcal{R}_i$ with $i < M + q$ and $i \neq 1$ is satisfied before $\mathcal{R}_1$. (Requests of jobs τ_2–τ_6 in Figure 5.4.)*

By Rule SR3, in addition to τ_{M+q}, $\min\{N - M - q, M - 1\}$ new jobs are released at time $t_q + 1$. By Claim 5.5, $q \leq N - 2M + 1$ holds. Thus, $N - M - q \geq N - M - (N - 2M + 1) = M - 1$ and $\min\{N - M - q, M - 1\} = M - 1$. Therefore, including τ_{M+q}, a total of $M - 1 + 1 = M$ new jobs are released at time t_{q+1}. Thus, the following holds.

Property 5.2. *There are M active requests $\mathcal{R}_i$ at time $t_q + 1$ with $i \geq M + q$. (Requests of jobs τ_7–τ_{10} in Figure 5.4 at time 7.)*

Since $t_q + 1$ is the first time instant when jobs are released by Rule SR3, by Lemma 5.9, there are M pending jobs τ_i with $i < M + q$ at time t_q. By the definition of t_q, a request from one of these M jobs is satisfied at time t_q. Since $1 < L$, the satisfied request is not complete by time $t_q + 1$. Thus, these M pending jobs with $i < M + q$ are also pending at time $t_q + 1$. Therefore, we have the following.

Property 5.3. *There are M active requests $\mathcal{R}_i$ at time $t_q + 1$ with $i < M + q$. (Requests of jobs τ_1, τ_3, τ_5, and τ_6 in Figure 5.4 at time 7.)*

By Properties 5.2 and 5.3, there are total $2M$ pending requests at time $t_q + 1$. Let $\mathcal{R}_i$ be the last satisfied request among these $2M$ requests. By the definition of $\mathcal{R}_{M+q}$, $i \neq 1$ holds, as $\mathcal{R}_{M+q}$ is ordered after $\mathcal{R}_1$. Therefore, by Property 5.1, $i \geq M + q$ holds. By Rule QR, τ_i has higher priority than each of the M jobs τ_j with $j < M + q$. By Rule SR4, no new jobs will be released until $\mathcal{R}_i$ is complete. Thus, τ_i experiences pi-blocking until it is satisfied, which occurs after the completion of the other $2M - 1$ requests. Since one of these $2M - 1$ requests is satisfied at time t_q and τ_i is released at time $t_q + 1$, τ_i incurs pi-blocking for at least $(2M - 1)L - 1$ time units, contradicting Assumption 5.1. □

5.2.2.3 Job Priority Assignment

In this section, we show how the lower-bound proof in Section 5.2.2.2 applies under different schedulers by demonstrating how jobs can be assigned priorities under these schedulers so that Rule QR holds.

G-FP schedulers. The following theorem shows that the lower-bound proof in Section 5.2.2.2 is valid under any G-FP scheduler.

Theorem 5.5. *Under any reorder-bounded locking protocol, a job in Γ_{seq} incurs pi-blocking for at least $(2M - 1)L - 1$ time units under any G-FP scheduler.*

Proof. Consider a G-FP scheduler $\mathcal{F}$. We re-index the tasks in Γ based on the task priority assignment under $\mathcal{F}$. For each $i > j$, τ_i has higher priority than τ_j. Thus, job priorities under $\mathcal{F}$ satisfies Rule QR. The theorem follows from Theorem 5.4. □

GEL schedulers. We now show that the lower-bound proof in Section 5.2.2.2 also applies under a class of GEL schedulers. The GEL schedulers in this class assign RPPs to tasks in Γ satisfying the following constraints.

$$\forall\, i : 1 \leq i \leq N - 1 :: Y_i > Y_{i+1} + (2M - 1)L \tag{5.12}$$

Theorem 5.6. *Under any reorder-bounded locking protocol, a job in Γ_{seq} incurs pi-blocking for at least $(2M - 1)L - 1$ time units under any GEL scheduler that satisfies* (5.12).

Proof. Assume that each job in Γ_{seq} incurs pi-blocking for less than $(2M - 1)L - 1$ time units under a GEL scheduler that assigns task RPPs according to (5.12). We first claim that the following hold.

$$\forall\, i \geq 1 : t_{i+1} - t_i < (2M - 1)L \tag{5.13}$$

$$\forall\, i \geq 1 : t_1 < (2M-1)L \tag{5.14}$$

If either (5.13) or (5.14) does not hold, then a job must incur at least $(2M-1)L$ time units of pi-blocking during $[t_i, t_{i+1})$, a contradiction.

We now show that (5.12) satisfies Rule QR. We show this by considering two consecutive jobs τ_i and τ_{i+1}. We first show that $y(\tau_{i+1}) - y(\tau_i) < 0$ by considering the following three cases.

Case 1. $i = M$. In this case, by Rules SR1 and SR2, τ_i and τ_{i+1} are released at times 0 and t_1, respectively. By (5.7), we have $y(\tau_{i+1}) - y(\tau_i) = r(\tau_{i+1}) + Y_{i+1} - r(\tau_i) - Y_i = t_1 + 1 + Y_{i+1} - Y_i$. Thus, by (5.12) and (5.14), we have $y(\tau_{i+1}) - y(\tau_i) < (2M-1)L - (2M-1)L = 0$.

Case 2. $i < M$ or τ_{i+1} is released by Rule SR3. In this case, by Rules SR1 and SR3, both jobs τ_i and τ_{i+1} are released at the same time, *i.e.,* $r(\tau_{i+1}) = r(\tau_i)$ holds. By (5.7) and (5.12), we have $y(\tau_{i+1}) - y(\tau_i) = r(\tau_{i+1}) + Y_{i+1} - r(\tau_i) - Y_i = Y_{i+1} - Y_i < -(2M-1)L < 0$.

Case 3. $i > M$ and τ_{i+1} is not released by Rule SR3. In this case, both jobs τ_i and τ_{i+1} are released by Rule SR2. By Rule SR2, τ_{M+k} is released at time $t_k + 1$. Thus, $\tau_i = \tau_{M+(i-M)}$ is released at time $t_{i-M} + 1$. Similarly, $\tau_i = \tau_{M+(i+1-M)}$ is released at time $t_{i+1-M} + 1$. Thus, $r(\tau_{i+1}) = t_{i+1-M} + 1$, and $r(\tau_i) = t_{i-M} + 1$. Hence, by (5.13), we have $r(\tau_{i+1}) - r(\tau_i) = t_{i-M+1} - t_{i-M} < (2M-1)L$. Now, by (5.7) and (5.12), we have $y(\tau_{i+1}) - y(\tau_i) = r(\tau_{i+1}) + Y_{i+1} - r(\tau_i) - Y_i = r(\tau_{i+1}) - r(\tau_i) + Y_{i+1} - Y_i < (2M-1)L - (2M-1)L = 0$.

In all three cases, $y(\tau_{i+1}) < y(\tau_i)$ holds. Therefore, τ_{i+1} has an earlier PP (hence, higher priority) than τ_i. Hence, Rule QR is satisfied. Thus, by Theorem 5.4, there is a job that incurs pi-blocking for at least $(2M-1)L - 1$ time units, a contradiction. □

Corollary 5.2. *Under any reorder-bounded locking protocol, a job in Γ_{seq} incurs pi-blocking for at least $(2M-1)L - 1$ time units G-EDF scheduling.*

Proof. Let $T_N \geq (2M-1)L + (2M-1)L^2/1 = NL$, and for each $i < N$, let $T_i = T_{i+1} + 2ML$. Thus, (5.12) is satisfied, as $Y_i = T_i$ holds under G-EDF. Note that $T_N = T_{min}$ holds. Thus, by Lemma 5.8, Γ is feasible. By Theorem 5.6, the corollary holds. □

5.3 Optimality Results Under FIFO Scheduling

In the previous section, we gave a lower bound of $2M - 2$ request lengths on per-request pi-blocking for mutex sharing under a class of non-FIFO global JLFP schedulers. However, since this lower bound does not apply to FIFO scheduling, the best-known lower bound on per-request pi-blocking under FIFO is $M - 1$ request lengths [Brandenburg and Anderson, 2010a]. In this section, we give a suspension-based locking protocol that achieves a pi-blocking upper bound of $M - 1$ request lengths. Thus, our locking protocol is optimal and the known lower bound of $M - 1$ request lengths is tight under FIFO. In addition to mutex locks, we also give multiprocessor locking protocols for *k-exclusion* sharing and *reader-writer* sharing under FIFO scheduling. Our locking protocols are designed for C-FIFO scheduling, so they apply to both G-FIFO and P-FIFO scheduling.

5.3.1 Resource-Holder's Progress Under FIFO Scheduling

Recall from Section 2.3 that any real-time locking protocol needs to ensure a resource-holding job's progress whenever a job waiting for the same resource is pi-blocked, for otherwise, the maximum per-job pi-blocking can be very large or even unbounded. This is done by employing *progress mechanisms* that may temporarily raise a job's *effective priority*. To date, many progress mechanisms have been devised to design multiprocessor locking protocols that are asymptotically optimal under any JLFP scheduling policy [Brandenburg and Anderson, 2010a, 2014]. Interestingly, for C-FIFO scheduling, no such progress mechanism is required to design optimal locking protocols. In fact, the C-FIFO scheduling policy itself has properties that ensure the progress of a resource-holding job. The key property that enables such progress is given in the following lemma.

Lemma 5.11. *Under C-FIFO scheduling, if a job $\tau_{i,j}$ becomes one of the c highest-priority eligible jobs in its cluster at time t_h, then it remains so during $[t_h, f(\tau_{i,j}))$.*

Proof. Assume for a contradiction that t is the first time instant in $[t_h, f(\tau_{i,j}))$ such that $\tau_{i,j}$ is not one of the c highest-priority eligible jobs in its cluster. Then, $t > t_h$ holds. By the definition of time t, there are at most $c - 1$ (resp., at least c) eligible jobs with higher priority than $\tau_{i,j}$ at time $t - 1 \geq t_h$ (resp., t) in $\tau_{i,j}$'s cluster. Thus, there is a task τ_p that has an eligible job $\tau_{p,v}$ with higher priority than $\tau_{i,j}$ at time t, but it has no such job at time $t - 1$.

Since $\tau_{p,v}$'s priority exceeds $\tau_{i,j}$'s, $r(\tau_{p,v}) \leq r(\tau_{i,j})$ holds. Since $\tau_{i,j}$ is eligible at time t_h, $r(\tau_{i,j}) \leq t_h$ holds. Thus, $r(\tau_{p,v}) \leq t_h$ and $\tau_{p,v}$ is pending at time $t-1$. We now consider two cases.

Case 1. $v = 1$. In this case, $\tau_{p,v}$ is also eligible at time t_h. Thus, τ_p has an eligible job with higher priority than $\tau_{i,j}$ at time $t-1$, a contradiction.

Case 2. $v > 1$. Since $\tau_{p,v}$ is not eligible at time $t-1$, job $\tau_{p,v-1}$ is eligible at time $t-1$. We have $r(\tau_{p,v-1}) < r(\tau_{p,v}) \leq r(\tau_{i,j})$. Thus, τ_p has an eligible job with higher priority than $\tau_{i,j}$ at time $t-1$, a contradiction.

Therefore, we reach a contradiction in both cases. □

Utilizing Lemma 5.11, we have the following lemma.

Lemma 5.12. *If a job $\tau_{i,j}$ issues a request $\mathcal{R}$ when it is one of the c highest-priority jobs in its cluster, then $\tau_{i,j}$ is always scheduled from $\mathcal{R}$'s satisfaction to completion.*

Proof. Let t_r, t_s, and t_c be the time instants when $\mathcal{R}$ is issued, satisfied, and complete, respectively. Thus, $t_r \leq t_s \leq t_c$ holds. Since $\tau_{i,j}$ is one of the c highest-priority eligible jobs in its cluster at time t_r, by Lemma 5.11, $\tau_{i,j}$ remains one of the c highest-priority eligible jobs in its cluster throughout $[t_r, t_c)$. Since $\mathcal{R}$ is satisfied at time $t_s \geq t_r$, $\tau_{i,j}$ is ready throughout $[t_s, t_c)$. Thus, $\tau_{i,j}$ is scheduled during $[t_s, t_c)$. □

Thus, by requiring a request to be issued only when the request-issuing job is one of the top-c-priority jobs in its cluster, we can ensure a resource-holder's progress under C-FIFO scheduling. We exploit this property in designing our protocols. Note that the C-OMLP ensures this property by employing priority donation as its progress mechanism at the expense of additional release blocking that may be incurred by a job even if it does not require any resources [Brandenburg and Anderson, 2011]. Due to this, our protocols have features in common with the C-OMLP.

5.3.2 Mutex Locks

In this section, we introduce the *optimal locking protocol for mutual exclusion sharing under C-FIFO scheduling* (OLP-F), which achieves the optimal pi-blocking bound under C-FIFO scheduling. To match the lower bound on pi-blocking, the OLP-F ensures that each job suffers pi-blocking for the duration of at most $M-1$ request lengths and incurs no release blocking.

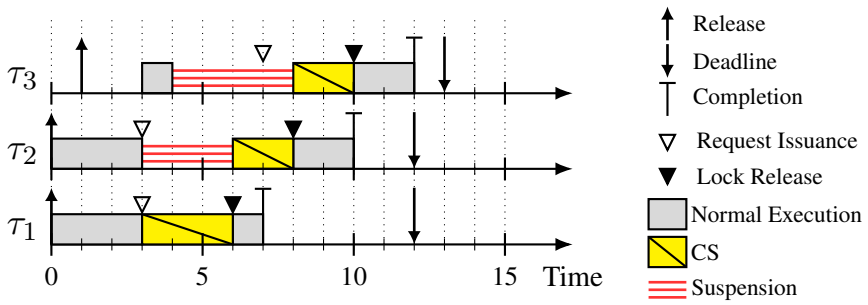


Figure 5.5: A schedule illustrating the OLP-F.

Structures. For each resource ℓ_q, we have a FIFO queue FQ_q that contains requests for ℓ_q. A request $\mathcal{R}$ is satisfied if and only if $\mathcal{R}$ is the head of the FQ_q.

Rules. When a job $\tau_{i,j}$ attempts to issue a request $\mathcal{R}$ for a resource ℓ_q, it proceeds according to the following rules.

MR1. $\tau_{i,j}$ is allowed to issue $\mathcal{R}$ only if it is one of the c highest-priority eligible jobs in its cluster. $\tau_{i,j}$ is suspended if necessary to ensure this condition.

MR2. When $\tau_{i,j}$ issues $\mathcal{R}$, $\mathcal{R}$ is enqueued in FQ_q. If $\tau_{i,j}$ becomes the head of FQ_q, then it is immediately satisfied. Otherwise, it is suspended.

MR3. $\mathcal{R}$ is satisfied when it is the head of FQ_q. $\mathcal{R}$ is removed from the FQ_q when it is complete.

Example 5.5. Figure 5.5 illustrates a C-FIFO schedule of three jobs on a two-processor cluster. The jobs of τ_1 and τ_2 are released earlier (hence, have higher priorities) than the job of τ_3. Both jobs of τ_1 and τ_2 issue requests for resource ℓ_q at time 3 and τ_1's request is enqueued first. Assuming no job in a different cluster holds ℓ_q, τ_1's job acquires ℓ_q at time 3 by Rule MR2. At time 3, since τ_2's job is suspended, the job of τ_3 starts to execute. At time 4, the job of τ_3 attempts to issue a request for ℓ_q, but it is suspended due to Rule MR1 as it is not one of the top-2-priority jobs at that time. At time 6, τ_1's job releases ℓ_q and τ_2's request is satisfied according to Rule MR3. Since the job of τ_3 becomes one of the top-2-priority jobs when τ_1's job completes, it issues a request for ℓ_q at time 7. ◀

Analysis. To derive an upper bound on the pi-blocking suffered by a job, we first show that FQ_q contains no more than M requests at any time.

Lemma 5.13. *Under the OLP-F, at any time, FQ_q contains at most M requests.*

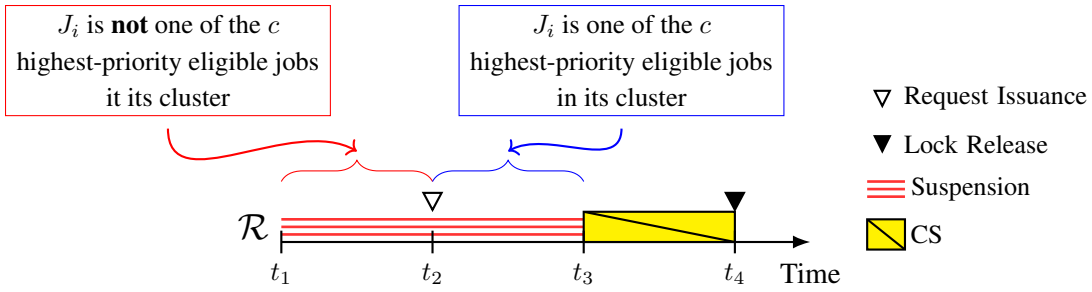


Figure 5.6: Timeline of a request under the OLP-F.

Proof. Assume that t is the first time instant when FQ_q contains more than M requests. Each job has at most one active request at any time. Thus, at time t, FQ_q must contain a request $\mathcal{R}$ issued by a job $\tau_{i,j}$ that is not one of the c highest-priority eligible jobs in its cluster. Let $t' \leq t$ be the time instant when $\tau_{i,j}$ issues $\mathcal{R}$. By Rule MR1, $\tau_{i,j}$ is one of the c highest-priority eligible jobs in its cluster at time t'. Since $\tau_{i,j}$ is not complete at time t, by Lemma 5.11, it is one of the c highest-priority eligible jobs in its cluster at time t, a contradiction. □

We now determine an upper bound on the request blocking suffered by job $\tau_{i,j}$ when it issues a request $\mathcal{R}$ for resource ℓ_q. Figure 5.6 depicts the timeline of $\mathcal{R}$ from when $\tau_{i,j}$ attempts to issue $\mathcal{R}$ to when $\mathcal{R}$ completes. Let t_1 be the time instant when job $\tau_{i,j}$ attempts to issue request $\mathcal{R}$. Let t_2 be the first time instant at or after time t_1 when $\tau_{i,j}$ becomes one of the top-c-priority eligible jobs in its cluster. Therefore, by Rule MR1, $\mathcal{R}$ is issued at time t_2. Let t_3 and t_4 be the time instants when $\mathcal{R}$ is satisfied and completed, respectively.

Lemma 5.14. *During $[t_1, t_3]$, $\tau_{i,j}$ incurs pi-blocking for at most $L^q_{sum,M-1}$ time units.*

Proof. By the definition of t_2, $\tau_{i,j}$ is not one of the top-c-priority eligible jobs in its cluster during $[t_1, t_2)$. Hence, $\tau_{i,j}$ is not pi-blocked during that time. By Lemma 5.11, $\tau_{i,j}$ is pi-blocked throughout $[t_2, t_3)$. By Lemma 5.12, $\tau_{i,j}$ is continuously scheduled during $[t_3, t_4)$. Thus, from t_1 to t_4, $\tau_{i,j}$ is only pi-blocked during $[t_2, t_3)$.

By Lemma 5.13, at most $M - 1$ other requests precede $\mathcal{R}$ in FQ_q at time t_2. By Rule MR3 and Lemma 5.12, each job at the head of FQ_q is continuously scheduled until its request is complete. Since each task τ_k has at most P_k eligible jobs and each job has at most one request at any time, $t_3 - t_2$ is not more than $L^q_{sum,M-1}$ (by Definition 5.1) time units and the lemma follows. □

We now show that the OLP-F does not cause any release blocking under C-FIFO scheduling.

Table 5.2: Asymptotically optimal locking protocols for k-exclusion locks under s-oblivious analysis.

Scheduling	**Protocol**	**Release blocking**	**Request blocking**
Clustered JLFP	C-K-OMLP [Brandenburg and Anderson, 2010a]	$\max_q\{\lceil M/k_q \rceil L^q_{max}\}$	$(\lceil M/k_q \rceil - 1)L^q_{max}$
Global JLFP	OKGLP [Elliott and Anderson, 2013]	0	$(2\lceil M/k_q \rceil + 4)L^q_{max}$
Global JLFP	R^{2}DGLP [Ward et al., 2012]	0	$(2\lceil M/k_q \rceil - 2)L^q_{max}$
C-FIFO	k-OLP-F (This dissertation)	0	$(\lceil M/k_q \rceil - 1)L^q_{max}$

Lemma 5.15. *Under the OLP-F, no job incurs release blocking.*

Proof. Since a resource-holding job is scheduled only when its priority is among the top c in its cluster, a resource request $\mathcal{R}$ does not cause pi-blocking to any job (within and across cluster boundaries) that does not issue a request during the time $\mathcal{R}$ is satisfied. □

Theorem 5.7. *Under the OLP-F, $\tau_{i,j}$ is pi-blocked for at most $b_i = \sum_{q=1}^{n_r} N_i^q \cdot L^q_{sum,M-1}$ time units.*

Proof. Follows from Lemmas 5.14 and 5.15. □

Thus, the OLP-F is an optimal locking protocol under C-FIFO scheduling.

5.3.3 *k*-Exclusion Locks

k-exclusion generalizes mutual exclusion by allowing up to k simultaneous lock holders; thus, mutual exclusion is equivalent to 1-exclusion. In this section, we give an optimal k-exclusion locking protocol under C-FIFO scheduling. We assume that a resource ℓ_q can be concurrently held by up to $k_q \leq M$ jobs. We begin by reviewing known lower-bound results for k-exclusion.

Lower bound on pi-blocking. For k-exclusion, Elliot *et al.* showed that a task system and a release sequence for it exist such that a job requesting a resource ℓ_q incurs s-oblivious pi-blocking for the duration of $\lceil \frac{M-k_q}{k_q} \rceil$ request lengths under any JLFP scheduler [Elliott and Anderson, 2013].

Asymptotically optimal locking protocols. Under s-oblivious analysis, the C-K-OMLP [Brandenburg and Anderson, 2010a], the OKGLP [Elliott and Anderson, 2013], and the R^{2}DGLP [Ward et al., 2012] ensure asymptotically optimal pi-blocking for k-exclusion. Table 5.2 summarizes these protocols.

The k-OLP-F. We now introduce the *optimal locking protocol for k-exclusion under C-FIFO scheduling* (k-OLP-F), which achieves optimal pi-blocking for k-exclusion under C-FIFO scheduling. The k-OLP-F ensures that a job suffers pi-blocking for the duration of no more than $\lceil \frac{M-k_q}{k_q} \rceil$ request lengths for each request for ℓ_q and incurs no release blocking.

Structures. For each resource ℓ_q, we have a FIFO queue FQ_q that contains *waiting* requests for ℓ_q. We also have a queue SQ_q of length at most k_q that contains the *satisfied* requests for ℓ_q. Initially, both queues are empty. A request $\mathcal{R}$ is satisfied if and only if $\mathcal{R}$ is in SQ_q.

Rules. When a job $\tau_{i,j}$ attempts to issue a request $\mathcal{R}$ for a resource ℓ_q, it proceeds according to the following rules.

KR1. $\tau_{i,j}$ is allowed to issue $\mathcal{R}$ only if $\tau_{i,j}$ is one of the c highest-priority eligible jobs in its cluster. $\tau_{i,j}$ suspends if necessary to ensure this condition.

KR2. If the length of SQ_q is less than k_q when $\tau_{i,j}$ issues $\mathcal{R}$, then $\mathcal{R}$ is enqueued in SQ_q and is immediately satisfied. Otherwise, $\mathcal{R}$ is enqueued in FQ_q and $\tau_{i,j}$ suspends.

KR3. When $\mathcal{R}$ completes, it is removed from SQ_q. If FQ_q is non-empty at that time, then the head of FQ_q is dequeued, enqueued in SQ_q, and satisfied.

Example 5.6. Figure 5.7 shows a schedule of five jobs that share a resource ℓ_q with $k_q = 2$. The jobs of τ_1, τ_2, and τ_3 (resp., τ_4 and τ_5) are C-FIFO scheduled on a two-processor cluster G_1 (resp., G_2). Since SQ_q is initially empty, by Rule KR2, the jobs of τ_4 and τ_1 acquire ℓ_q at times 2 and 3, respectively. Since the jobs of both τ_2 and τ_5 are one of the top-2-priority eligible jobs in their clusters, by Rule KR1, they issue requests for ℓ_q at times 4 and 5, respectively. At time 5, the job of τ_3 attempts to issue a request for ℓ_q, but is suspended by Rule KR1. At time 5, the job of τ_4 releases ℓ_q and is removed from SQ_q by Rule KR3. τ_2's request is at the head of FQ_q at time 5, so by Rule KR3, it is removed from FQ_q, enqueued in SQ_q, and satisfied. At time 7, the job of τ_1 completes and the job of τ_3 becomes one of the top-2-priority jobs in G_1 and issues its request, by Rule KR1. ◀

Analysis. We now derive an upper bound on the pi-blocking suffered by a job under the k-OLP-F. We first derive an upper bound on the number of waiting requests in FQ_q.

Lemma 5.16. *Under the k-OLP-F, FQ_q contains at most $M - k_q$ requests.*

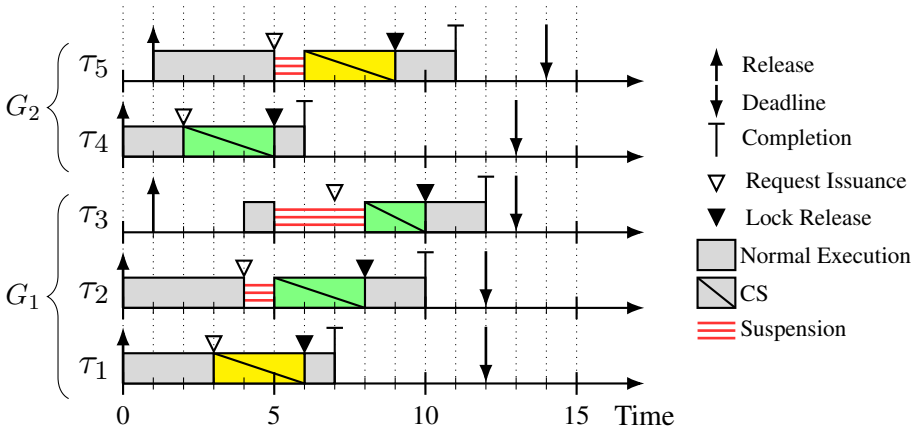


Figure 5.7: A schedule illustrating the k-OLP-F. Concurrent resource accesses are shaded differently.

Proof. Assume otherwise. Let t be the first time instant such that FQ_q contains more than $M - k_q$ requests. Thus, a new request $\mathcal{R}'$ is enqueued in FQ_q at time t. By Rule KR2, SQ_q contains k_q requests at time t. Thus, the number of active requests (either satisfied or waiting) is more than $k_q + M - k_q = M$ at time t. Since each job has at most one active request at any time, there is an active request $\mathcal{R}$ issued by a job $\tau_{i,j}$ that is not one of the c highest-priority jobs in its cluster. By Rule KR1, $\tau_{i,j}$ is one of the c highest-priority jobs in its cluster when it issues $\mathcal{R}$ at time $t' \leq t$. By Lemma 5.11, $\tau_{i,j}$ remains one of the c highest-priority jobs in its cluster at time t, a contradiction. □

We now determine an upper bound on request blocking. We consider a job $\tau_{i,j}$ that issues a request $\mathcal{R}$ for resource ℓ_q. As in Figure 5.6, let t_1, t_2, t_3, and t_4 be the time instants corresponding to when $\tau_{i,j}$ attempts to issue $\mathcal{R}$, and when $\mathcal{R}$ is issued, satisfied, and complete, respectively.

Lemma 5.17. *For request $\mathcal{R}$, $\tau_{i,j}$ suffers request blocking for at most $L^q_{sum,\lceil \frac{M-k_q}{k_q} \rceil}$ time units.*

Proof. By Definition 5.4, $\tau_{i,j}$ does not suffer any pi-blocking during $[t_1, t_2)$ and $[t_3, t_4)$. By Lemma 5.11 and the definition of t_2, $\tau_{i,j}$ suffers pi-blocking during the entire duration of $[t_2, t_3)$, so it suffices to upper bound $(t_3 - t_2)$. If SQ_q contains fewer than k_q requests at time t_2, then $t_3 - t_2 = 0$ holds by Rule KR2, so assume otherwise. At time t_2, no two requests in SQ_q and FQ_q are from the same task. By Rule KR3, $\mathcal{R}$ is satisfied when it is dequeued from FQ_q. Thus, by Lemma 5.16, at most $M - k_q$ requests are required to be dequeued to satisfy $\mathcal{R}$. By Rule KR2, k_q jobs hold ℓ_q throughout $[t_2, t_3)$. By Rule KR1 and Lemma 5.12, each resource-holding job is always scheduled. Thus, per $L^q_{sum,h}$ time units during $[t_2, t_3)$ at least $h \cdot k_q$

requests complete—and hence, by Rule KR3, at least $h \cdot k_q$ requests are dequeued from FQ_q. Dequeuing $M - k_q$ requests from FQ_q thus requires at most $L^q_{sum,\lceil \frac{M-k_q}{k_q} \rceil}$ time units, so $t_3 - t_2 \leq L^q_{sum,\lceil \frac{M-k_q}{k_q} \rceil}$. □

Similar to the OLP-F, no release blocking occurs under the k-OLP-F. Therefore, by Lemma 5.17, we have the following theorem.

Theorem 5.8. *Under the k-OLP-F, $\tau_{i,j}$ suffers pi-blocking for at most $b_i = \sum_{q=1}^{n_r} N_i^q \cdot L^q_{sum,\lceil \frac{M-k_q}{k_q} \rceil}$ time units.*

Thus, the k-OLP-F is optimal for k-exclusion locking under C-FIFO scheduling.

5.3.4 Reader-Writer Locks

Some resources can be read without alteration. For such resources, it may be desirable to support *reader-writer* (RW) sharing. Here, *writers* have mutually exclusive access to the resource, but multiple *readers* can access the resource simultaneously.

Under RW sharing, it is often desirable to ensure fast read access. However, enabling fast read access may cause write requests to starve. This can happen under a *read-preference* RW lock that never satisfies a write request if a read request is active. More generally, these observations give rise to an important question: what is the minimum request blocking a read request can incur without causing a write request to starve?

Lower bound on read request blocking. As we show next, ensuring a read request delay of $2L^q_{max} - 2$ time units can in fact cause writer starvation.

Theorem 5.9. *For $M \geq 8$, a task system and a release sequence for it exist such that any locking protocol that ensures request blocking of at most $2L^q_{max} - 2$ time units for read requests causes unbounded request blocking for write requests under any work-conserving scheduler.*

Proof. We give an example task system Γ and a release sequence for it supporting the claim. Let $\tau_1, \tau_2, \ldots, \tau_M$ be M sporadic tasks scheduled on M processors. All tasks have WCETs of $L + 1$ time units with $2 \leq L \leq (M-2)/3$. Figure 5.8 illustrates this for $M = 8$ and $L = 2$. Each job's execution consists of 1.0 time unit of non-CS execution followed by L time units of CS execution. Tasks $\tau_1, \tau_2, \ldots, \tau_{M-1}$ issue read requests for resource ℓ_q, while τ_M issues a write request for ℓ_q. The periods of all tasks are $M - 1$. Each task has an implicit deadline.

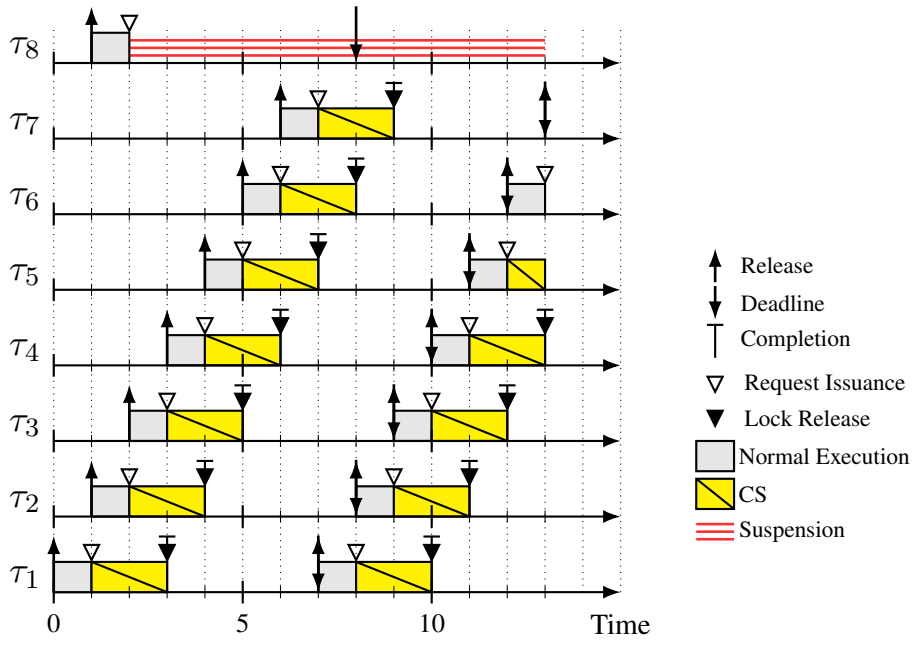


Figure 5.8: A schedule illustrating Theorem 5.9.

Feasibility of Γ. We show that Γ is HRT-schedulable under a *write-preference* RW lock and any work-conserving scheduler. A write-preference RW lock does not satisfy any read request if a write request is waiting. Since τ_M is the only writer task, under a write-preference RW lock, τ_M's jobs acquire ℓ_q immediately (if no reader jobs hold ℓ_q) or immediately after the currently satisfied read requests complete (otherwise). Thus, the following property is satisfied.

Property 5.4. *Each of τ_M's jobs acquires ℓ_q within L time units of its request issuance.*

Since there are M tasks, a processor is always available for τ_M. Thus, with a WCET of $L+1$ and resource acquisition time of at most L, each job of τ_M completes within $L + 1 + L = 2L + 1 \leq 2(M - 2)/3 + 1 < M - 2 + 1 = M - 1 = T_M$ time units after its release.

For reader tasks $\tau_1, \tau_2, \ldots, \tau_{M-1}$, a read request $\mathcal{R}$ issued at time t is satisfied immediately if there is no waiting write request. Otherwise, by Property 5.4, the pending write request by τ_M's job is satisfied by time $t + L$ and complete by time $t + L + L = t + 2L$ (as a processor is available). Since τ_M is the only writer task, after completion of the write request, there is no pending write request. Thus, $\mathcal{R}$ is satisfied by time $t + 2L$. With a WCET of $L + 1$, the job issuing $\mathcal{R}$ completes within $L + 1 + 2L = 3L + 1 \leq 3(M - 2)/3 + 1 = M - 2 + 1 = M - 1 = T_i$ time units after its release. Therefore, Γ is HRT-feasible.

Release sequence for Γ. τ_M releases its jobs periodically from time 1. τ_1 releases its first job at time 0 and its subsequent jobs' release times are defined as $r(\tau_{1,j+1}) = f(\tau_{M-1,j}) - L$. The release times of τ_i's jobs with $2 \leq i < M$ are $r(\tau_{i,j}) = f(\tau_{i-1,j}) - L$. Thus, for $2 \leq i < M$, we have

$$\begin{aligned} r(\tau_{i,j}) &= f(\tau_{i-1,j}) - L \\ &\geq \{\text{Since } \tau_{i-1,j} \text{ executes for } L+1 \text{ time units}\} \\ &\quad r(\tau_{i-1,j}) + L + 1 - L \\ &= r(\tau_{i-1,j}) + 1. \end{aligned} \tag{5.15}$$

Similarly, for τ_1, it can be shown that

$$r(\tau_{1,j+1}) \geq r(\tau_{M-1,j}) + 1. \tag{5.16}$$

We now show that consecutive jobs of τ_i with $i < M$ are released at least T_i time units apart. For $2 \leq i < M$, by (5.15), we have

$$\begin{aligned} r(\tau_{i,j+1}) &\geq r(\tau_{i-1,j+1}) + 1 \\ &\geq \{\text{Applying (5.15) repeatedly for } i-2 \text{ times}\} \\ &\quad r(\tau_{1,j+1}) + 1 + (i-2) \\ &\geq \{\text{By (5.16)}\} \\ &\quad r(\tau_{M-1,j}) + 1 + (i-1) \\ &\geq \{\text{Applying (5.15) repeatedly for } M-1-i \text{ times}\} \\ &\quad r(\tau_{i,j}) + (M-1-i) + i \\ &= r(\tau_{i,j}) + M - 1 \\ &= r(\tau_{i,j}) + T_i. \end{aligned} \tag{5.17}$$

Similarly, we can show that consecutive jobs of τ_1 are released at least T_1 time units apart.

We now show that each job of τ_i with $i < M$ is eligible when it is released by showing that $\tau_{i,j}$ completes before $\tau_{i,j+1}$'s release. For $2 \leq i < M-1$, in the third step of the derivation of (5.17), applying

(5.15) repeatedly for $M-2-i$ times instead of $M-1-i$ times, we have $r(\tau_{i,j+1}) \geq r(\tau_{i+1,j}) + (M-2-i) + i = r(\tau_{i+1,j}) + M - 2$. Since $L \leq (M-2)/3 < M-2$ and $r(\tau_{i+1,j}) = f(\tau_{i,j}) - L$, we get $r(\tau_{i,j+1}) > r(\tau_{i+1,j}) + L = f(\tau_{i,j})$. For $i = M-1$, the first step in the derivation of (5.17) yields $r(\tau_{M-1,j+1}) \geq r(\tau_{1,j+1}) + 1 + (M-1-2) = r(\tau_{1,j+1}) + M - 2 > r(\tau_{1,j+1}) + L$. Since $r(\tau_{1,j+1}) = f(\tau_{M-1,j}) - L$, we get $r(\tau_{M-1,j+1}) > f(\tau_{M-1,j})$. For $i = 1$, applying (5.15) in (5.16) repeatedly for $M-3$ times, we have $r(\tau_{1,j+1}) \geq r(\tau_{2,j}) + M - 2 > r(\tau_{2,j}) + L = f(\tau_{1,j})$. Thus, $r(\tau_{i,j+1}) > f(\tau_{i,j})$ for $i < M$.

Finishing up. We now prove the theorem by showing that $\tau_{M,1}$'s write request is never satisfied if the request blocking for read requests is at most $2L-2$. Assume that $\tau_{M,1}$'s request is satisfied at time t. We have $t > 2$, as $\tau_{M,1}$ issues its request at time 2 and $\tau_{1,1}$ holds ℓ_q then (under a work-conserving scheduling policy, $\tau_{1,1}$ acquires ℓ_q at time 1). Since the scheduling policy is work-conserving, a job $\tau_{i,j}$ must release ℓ_q at time t. Thus, $f(\tau_{i,j}) = t$.

By the job release pattern of $\tau_1, \tau_2, \ldots, \tau_{M-1}$, there exists a job $\tau_{w,v}$ such that $r(\tau_{w,v}) = f(\tau_{i,j}) - L = t - L$. Since each job is eligible when it is released and there are M tasks, $\tau_{w,v}$ issues a read request $\mathcal{R}$ at time $r(\tau_{w,v}) + 1 = t - L + 1 < t$ (as $L \geq 2$). Since $\tau_{M,1}$'s write request is satisfied at time t, $\mathcal{R}$ cannot be satisfied before time $t + L$. Since the task count is M, $\tau_{w,v}$ is pi-blocked for a duration of at least $t + L - (t - L + 1) = 2L - 1$ time units. Thus, request blocking for read requests exceeds $2L-2$ time units, reaching a contradiction. □

Thus, read request blocking of at least $2L^q_{max} - 1$ time units is fundamental to avoid writer starvation. We now establish a lower bound on write request blocking when read requests suffer request blocking for at most $2L^q_{max} - 1$ time units.[5]

Theorem 5.10. *For $M \geq 4$, there exists a task system and a release sequence for it such that any locking protocol that ensures at most $2L^q_{max} - 1$ read request blocking causes write request blocking of $(2M - 5)L^q_{max} - 1$ time units under any work-conserving scheduler.*

Proof. Let $\tau_1, \tau_2, \ldots, \tau_N$ be N tasks scheduled on $M \geq 4$ processors, where $N = 2M - 4$. Each task has a WCET of $L + 1$ time units with $L \geq 1$. Figure 5.9 illustrates this for $M = 5$ and $L = 3$. Each job's

[5]Assuming higher read request blocking would yield a smaller lower bound on write request blocking. Note that deriving tight lower bounds for RW locks is much more complicated than for the other locks considered in this dissertation because much leeway exists regarding the interplay between readers and writers.

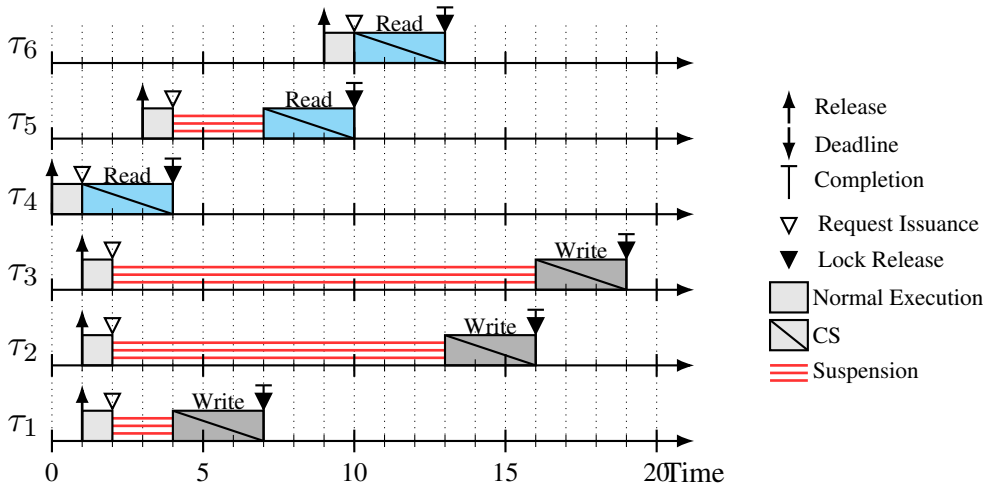


Figure 5.9: A schedule illustrating Theorem 5.10. Read and write CSs are shaded differently.

execution consists of 1.0 time unit of non-CS execution followed by L time units of CS execution. Tasks $\tau_1, \tau_2, \ldots, \tau_{M-2}$ issue write requests for resource ℓ_q, while $\tau_{M-1}, \tau_M, \ldots, \tau_{2M-4}$ issue read requests for ℓ_q. Each task's period is $T \geq (2M-4) \cdot (L+1)$ and relative deadline is $T \geq (2M-4) \cdot (L+1)$. The task WCETs sum to $(2M-4) \cdot (L+1)$, so the task system can be scheduled by sequentially executing the jobs on a single processor (*i.e.,* it is feasible).

Tasks $\tau_1, \tau_2, \ldots, \tau_{M-2}$ release their first jobs at time 1. Task τ_{M-1} releases its first job at time 0. For $i > M-1$, the release time of $\tau_{i,1}$ is determined as $r(\tau_{i,1}) = f(\tau_{i-1,1}) - 1$. Hence, from time 0, there is always an eligible first job of a task until all first jobs are complete. Since all WCETs sum to $(2M-4) \cdot (L+1)$, under a work-conserving scheduler, the first job of each task completes by time $(2M-4) \cdot (L+1) \leq T$. Subsequent job release times can be easily defined so that each task's consecutive job releases are at least T time units apart.

We now prove that each first job $\tau_{i,1}$ always incurs pi-blocking when it is waiting for ℓ_q. For any job $\tau_{i,1}$ with $i > M$, we have $r(\tau_{i,1}) = f(\tau_{i-1,1}) - 1 \geq r(\tau_{i-1,1}) + L + 1 - 1 = f(\tau_{i-2,1}) - 1 + L$. Since $L \geq 1$, we have $r(\tau_{i,1}) \geq f(\tau_{i-2,1})$. Thus, at most two first jobs of the last $M-2$ tasks are pending at the same time. Therefore, at most $M - 2 + 2 = M$ first jobs are pending at any time, which implies that a job $\tau_{i,1}$ incurs pi-blocking if it is waiting.

Finally, we prove the claim of the theorem by showing that there is a writer job that incurs pi-blocking for the duration of $(2M-5)L - 1$ time units. Job $\tau_{M-1,1}$ issues a read request at time 1 and acquires ℓ_q (as

Table 5.3: Asymptotically optimal locking protocols for RW locks under s-oblivious analysis.

Scheduling	**Protocol**	**Release blocking**	**Read request blocking**	**Write request blocking**
Clustered JLFP	CRW-OMLP [Brandenburg and Anderson, 2010a]	$2mL_{max}$	$2L^q_{max}$	$(2m-1)L^q_{max}$
C-FIFO	RW-OLP-F (This dissertation)	0	$2L^q_{max}-1$	$(2m-3)L^q_{max}$

the scheduling policy is work-conserving). Figure 5.9 illustrates this. Each job $\tau_{i,1}$ with $i < M-1$ issues a write request at time 2.

Each job $\tau_{i,1}$ with $i > M-1$ (*e.g.,* the jobs of τ_5 and τ_6 in Figure 5.9) is released 1.0 time unit before $\tau_{i-1,1}$ completes and issues a read request when $\tau_{i-1,1}$ completes. Thus, $\tau_{i,1}$'s read request cannot be delayed to satisfy two or more pending write requests without incurring read request blocking of at least $2L$ time units. As a result, at most one write request can be satisfied between two consecutive read requests. Thus, there is a write request from a job $\tau_{w,1}$ with $i < M-1$ (*e.g.,* τ_3's job in Figure 5.9) that must be satisfied after all read and write requests of each job $\tau_{i,1}$ with $i \neq u$ complete.

Since $\tau_{w,1}$ issues its request at time 2 and $\tau_{M-1,1}$ (*e.g.,* τ_4's job in Figure 5.9) acquires ℓ_q at time 1, $\tau_{M-1,1}$ pi-blocks $\tau_{u,1}$ for $L-1$ time units. The stated job release pattern ensures that no two of the remaining $M-3$ read requests (*e.g.,* those by τ_5 and τ_6 in Figure 5.9) overlap, so they pi-block $\tau_{w,1}$ for $(M-3)L$ time units. Finally, $\tau_{w,1}$ is pi-blocked by each of the other $M-3$ write requests (*e.g.,* those by τ_1 and τ_2 in Figure 5.9) for $(M-3)L$ time units. Thus, $\tau_{w,1}$ incurs pi-blocking for $L-1+(M-3)L+(M-3)L = (2M-5)L-1$ time units. □

For simplicity, Theorems 5.8 and 5.10 are stated for work-conserving scheduling. However, both theorems are also true under a wider class of schedulers and locking protocols that are *top-c-work-conserving*. On a c-processor cluster, a top-c-work-conserving scheduling ensures that any top-c-highest priority ready job immediately acquires a shared resource (including processor) if such a resource is idle. Note that a work-conserving scheduler and locking protocol combination is also top-c-work-conserving.

Asymptotically optimal RW locking protocols. For RW locks, the CRW-OMLP is an asymptotically optimal locking protocol under clustered JLFP scheduling [Brandenburg and Anderson, 2010a]. The CRW-OMLP is a *phase-fair* RW locking protocol. *Phase-fair* RW locks satisfy read and write requests in alternating phases [Brandenburg and Anderson, 2010b]. At the beginning of a *reader phase*, all waiting read

requests are satisfied simultaneously, while at the beginning of a *writer phase*, a single waiting write request is satisfied. Table 5.3 summarizes the CRW-OMLP.

The RW-OLP-F. We now introduce the *read-optimal RW locking protocol under C-FIFO scheduling (RW-OLP-F)*, which achieves optimal pi-blocking for read requests under C-FIFO scheduling. The RW-OLP-F is a phase-fair RW locking protocol that achieves $2L^q_{max} - 1$ (resp., $(2M-3)L^q_{max}$) request blocking for read (resp., write) requests. Unlike the CRW-OMLP, the RW-OLP-F has no release blocking under C-FIFO scheduling.

Structures. For each resource ℓ_q, we have two queues RQ^1_q and RQ^2_q that contain read requests for ℓ_q, and a FIFO queue WQ_q that contains write requests for ℓ_q. One of the read queues acts as a *collecting* queue and the other acts as a *draining* queue. The roles of RQ^1_q and RQ^2_q alternate, *i.e.,* each switches over time between being the collecting queue and being the draining queue. Initially, RQ^1_q is the collecting queue and RQ^2_q is the draining queue.

Reader rules. Assume that a job $\tau_{i,j}$ attempts to issue a read request $\mathcal{R}$ for resource ℓ_q. Let RQ^c_q and RQ^d_q be the collecting and draining queues, respectively, when $\tau_{i,j}$ issues $\mathcal{R}$.

RR1. $\tau_{i,j}$ is allowed to issue $\mathcal{R}$ only if it is one of the c highest-priority eligible jobs in its cluster. $\tau_{i,j}$ suspends if necessary to ensure this condition.

RR2. If WQ_q is empty when $\tau_{i,j}$ issues $\mathcal{R}$, then $\mathcal{R}$ is immediately satisfied and enqueued in RQ^d_q. Otherwise, $\tau_{i,j}$ suspends and $\mathcal{R}$ is enqueued in RQ^c_q.

RR3. If $\mathcal{R}$ is in RQ^c_q, then it is satisfied (along with all other requests in RQ^c_q) when RQ^c_q becomes the draining queue (see Rule WR3). If RQ^c_q becomes the draining queue at time t and a read request is issued at time t, then that request is enqueued in RQ^c_q before making it the draining queue. $\mathcal{R}$ is removed from RQ^c_q when it is complete. If RQ^c_q becomes empty because of $\mathcal{R}$'s removal, then the head of WQ_q (if any) is satisfied.

Writer rules. When a job $\tau_{w,j}$ attempts to issue a write request $\mathcal{R}$ for a resource ℓ_q, it proceeds according to the following rules.

WR1. $\tau_{w,j}$ is allowed to issue $\mathcal{R}$ only if it is one of the c highest-priority eligible jobs in its cluster. $\tau_{w,j}$ suspends if necessary to ensure this condition.

WR2. If RQ_q^1, RQ_q^2, and WQ_q are empty when $\mathcal{R}$ is issued, then $\mathcal{R}$ is immediately satisfied and enqueued in WQ_q. Otherwise, $\mathcal{R}$ is enqueued in WQ_q and $\tau_{w,j}$ suspends.

WR3. Let RQ_q^d and RQ_q^c be the draining and collecting queues, respectively, when $\mathcal{R}$ is the head of WQ_q. $\mathcal{R}$ is satisfied when $\mathcal{R}$ is the head of WQ_q and RQ_q^d is empty. When $\mathcal{R}$ is complete, $\mathcal{R}$ is dequeued from WQ_q and if RQ_q^c is non-empty, then RQ_q^c (resp., RQ_q^d) becomes the draining (resp., collecting) queue. Otherwise (RQ_q^c is empty), the new head of WQ_q (if any) is satisfied.

Analysis. We now determine an upper bound on request blocking. For $M \leq 2$, by Lemma 5.11 and Rules RR1 and WR1, there are at most two active requests and at most one waiting request at any time, so request blocking is at most L_{max}^q time units for both reads and writes. Henceforth, we assume $M \geq 3$. The following lemma follows from Lemma 5.11 and Rules RR1 and WR1; we omit its proof as it is similar to Lemma 5.13.

Lemma 5.18. *The total number of requests in RQ_q^1, RQ_q^2, and WQ_q is at most M.*

We now give two helper lemmas.

Lemma 5.19. *If a write request $\mathcal{R}$ is the head of WQ_q at time t, then it is satisfied by time $t + L_{max}^q$.*

Proof. Let RQ_q^c and RQ_q^d be the collecting and draining queue, respectively, at time t. If $\mathcal{R}$ is not satisfied at time t, then by Rule WR3, RQ_q^d is non-empty at time t. By Rule RR3, jobs with requests in RQ_q^d hold ℓ_q at time t. Let t' be the time instant when all such requests are complete. By Lemma 5.12 and Rule RR1, $t' \leq t + L_{max}^q$. By Rule RR2, no read requests are enqueued in RQ_q^d during $[t, t')$. Thus, RQ_q^d becomes empty at time t'. By Rule WR3, $\mathcal{R}$ is satisfied at time t'. Thus, the lemma holds. □

Lemma 5.20. *If a write request $\mathcal{R}$ is the head of WQ_q at time t, then it is complete by time $t + 2L_{max}^q$.*

Proof. By Lemma 5.19, $\mathcal{R}$ is satisfied by time $t + L_{max}^q$. By Lemma 5.12 and Rule WR1, $\mathcal{R}$ completes within L_{max}^q time units after being satisfied. Thus, the lemma holds. □

We now determine an upper bound on the request blocking suffered by a job when it issues a read request. We consider a job $\tau_{i,j}$ that issues a read request $\mathcal{R}$ for resource ℓ_q. As depicted in Figure 5.6, let t_1, t_2, t_3, and t_4 be the time instants corresponding to when $\tau_{i,j}$ attempts to issue $\mathcal{R}$, and when $\mathcal{R}$ is issued, satisfied, and complete, respectively. In the lemma below, we show that request blocking for read requests is at most $2L_{max}^q$.

Lemma 5.21. *For a read request $\mathcal{R}$, $\tau_{i,j}$ suffers request blocking for at most $2L^q_{max} - 1$ time units.*

Proof. $\tau_{i,j}$ suffers pi-blocking for the duration of $[t_2, t_3)$. Let RQ^c_q and RQ^d_q be the collecting and draining queue, respectively, at time t_2. If WQ_q is empty at time t_2, then $t_2 = t_3$ holds according to Rule RR2, so assume otherwise. By Rule RR2, $\mathcal{R}$ is enqueued in RQ^c_q. Let $\mathcal{R}'$ be the request at the head of WQ_q at time t_2. Assume that $\mathcal{R}'$ completes at time t'_2. By Rules WR3 and RR3, RQ^c_q becomes the draining queue and $\mathcal{R}$ is satisfied at time t'_2. Therefore, $t'_2 = t_3$ holds. We now consider two cases.

Case 1. *$\mathcal{R}'$ is satisfied at or before t_2.* Then, the completion time of $\mathcal{R}'$ is $t'_2 \leq t_2 + L^q_{max}$. Since $\mathcal{R}$ is enqueued in RQ^c_q at time t_2 and $\mathcal{R}'$ is satisfied throughout $[t_2, t'_2)$, by Rule WR3, RQ^c_q becomes the draining queue at time t'_2. Thus, by Rule RR3, all requests in RQ^c_q, including $\mathcal{R}$, are satisfied at time t'_2. Therefore, for $\mathcal{R}$, $\tau_{i,j}$ incurs pi-blocking for at most L^q_{max} time units.

Case 2. *$\mathcal{R}'$ is satisfied after t_2.* Since $\mathcal{R}'$ is the head of WQ_q at time t_2, read requests of the draining queue are satisfied at time t_2. By Rule RR3, the queue containing the satisfied read requests becomes the draining queue at or before time $t_2 - 1$, as otherwise $\mathcal{R}$ would have also been enqueued in that queue and immediately satisfied. All these read requests complete by time $t_2 - 1 + L^q_{max}$. Thus, the draining queue becomes empty (by Rule RR3) and $\mathcal{R}'$ is satisfied (by Rule WR3) by time $t_2 - 1 + L^q_{max}$. Therefore, $\mathcal{R}'$ completes and RQ^c_q becomes the draining queue (by Rule WR3) by time $t_2 - 1 + 2L^q_{max}$. Hence, $\mathcal{R}$ is satisfied by time $t_2 - 1 + 2L^q_{max}$, which implies that $\tau_{i,j}$ is pi-blocked for at most $2L^q_{max} - 1$ time units for request $\mathcal{R}$.

In both cases, $\tau_{i,j}$ suffers pi-blocking for at most $2L^q_{max} - 1$ time units for read request $\mathcal{R}$. □

Finally, we give an upper bound on the request blocking incurred by a job when issuing a write request. Let $\tau_{w,j}$ be a job that issues a write request $\mathcal{R}$ at time t.

Lemma 5.22. *For a write request $\mathcal{R}$, $\tau_{w,j}$ incurs request blocking for at most $(2M - 3)L^q_{max}$ time units.*

Proof. If no request holds ℓ_q at time t, then by Rule WR2, $\mathcal{R}$ is immediately satisfied. This leaves two cases.

Case 1. *A job with a read request holds ℓ_q at time t.* By Lemma 5.18, RQ^1_q, RQ^2_q, and WQ_q hold at most M requests at time t. Since there is an active read request, at most $M - 2$ write requests precede $\mathcal{R}$ in WQ_q. By Rule WR3, each of those write requests becomes the head of WQ_q when its preceding write request completes. By Lemma 5.20, a write request at the head of WQ_q completes within $2L^q_{max}$ time units from when it becomes the head. Thus, all $M - 2$ write requests that precede $\mathcal{R}$ in WQ_q are complete

by time $t + 2(M-2)L^q_{max}$. By Lemma 5.19, after becoming the head of WQ_q, $\mathcal{R}$ is satisfied within an additional L^q_{max} time units. Thus, $\mathcal{R}$ is satisfied by time $t + (2M-3)L^q_{max}$.

Case 2. *A job with a write request $\mathcal{R}'$ holds ℓ_q at time t.* We consider two subcases.

Case 2a. *WQ_q contains M requests at time t.* Thus, $M-1$ requests precede $\mathcal{R}$ in WQ_q. By Lemma 5.12 and Rule WR1, $\mathcal{R}'$ completes within L^q_{max} time units from t. By Lemma 5.11 and Rules RR1 and WR1, no requests are issued before $\mathcal{R}'$ completes. Thus, by Rule WR3, the write request $\mathcal{R}''$ following $\mathcal{R}'$ is satisfied when $\mathcal{R}'$ is complete. By Lemma 5.12 and Rule WR1, $\mathcal{R}''$ completes within L^q_{max} time from when it is satisfied. Thus, the top two requests in WQ_q are complete by time $t + 2L^q_{max}$. By Lemma 5.20, each of the remaining $M-3$ write requests preceding $\mathcal{R}$ is complete within $2L^q_{max}$ time units after becoming the head of WQ_q. Thus, $\mathcal{R}$ becomes the head of WQ_q by time $t + 2L^q_{max} + 2(M-3)L^q_{max} = t + 2(M-2)L^q_{max}$. By Lemma 5.19, $\mathcal{R}$ is satisfied within L^q_{max} time units after becoming WQ_q's head. Thus, $\mathcal{R}$ is satisfied by time $t + (2M-3)L^q_{max}$.

Case 2b. *WQ_q contains at most $M-1$ requests at time t.* Thus, at most $M-2$ requests precede $\mathcal{R}$ in WQ_q. By Lemma 5.12, $\mathcal{R}'$ completes within L^q_{max} time units from t. By Lemma 5.20, each of the remaining $M-3$ write requests preceding $\mathcal{R}'$ completes within $2L^q_{max}$ time units from when it becomes the head of WQ_q. Thus, $\mathcal{R}$ becomes the head of WQ_q within $L^q_{max} + 2(M-3)L^q_{max} = (2M-5)L^q_{max}$ time units from t. By Lemma 5.19, $\mathcal{R}$ is satisfied within L^q_{max} time units after becoming WQ_q's head. Thus, $\mathcal{R}$ is satisfied by time $(2M-4)L^q_{max}$. □

Similar to the OLP-F, no job suffers release blocking due to a resource-holding job under the RW-OLP-F. By Lemma 5.21 and 5.22 and letting $N^{q,r}_i$ and $N^{q,w}_i$ denote the maximum number of read and write requests for ℓ_q by τ_i, we have the following.

Theorem 5.11. *Under the RW-OLP-F, $\tau_{i,j}$ is pi-blocked for at most*

$$b_i = \sum_{q=1}^{n_r} \left(N^{q,r}_i \cdot (2L^q_{max} - 1) + N^{q,w}_i \cdot (2M-3)L^q_{max}\right).$$

By Rules RR1, RR2, WR1, and WR2, G-FIFO scheduling and RW-OLP-F ensures top-c-work-conserving property. Thus, by Theorems 5.9 and 5.10, the RW-OLP-F ensures that request blocking for read requests is optimal, while that for write requests is just within two request lengths of optimal. Note

that a more nuanced expression of the bound in Theorem 5.10 is possible by replacing $2L^q_{max}$ in by the sum of the maximum read request length and the maximum write request length.

5.4 Experimental Evaluation

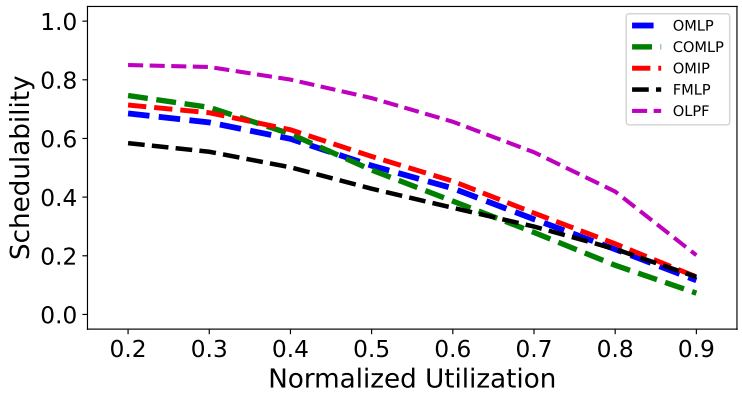
In this section, we present the results of experiments we have conducted to evaluate our proposed locking protocols for FIFO schedulers.

Task system generation. Our task-system generation method is similar to that used in prior locking-related schedulability studies [Brandenburg, 2011, 2014; Yang et al., 2015]. We generated task systems randomly for systems with $\{4, 8, 16\}$ processors. For each processor count, we generated task systems that have a *normalized utilization*, *i.e.,* $\sum_{i=1}^{N} u_i/M$, from 0.2 to 0.9 with a step size of 0.1. We chose the number of tasks uniformly from $[2M, 150]$. We generated each task's utilization uniformly following procedures from [Emberson et al., 2010]. We chose each task's period randomly from $[3, 33]$ms (*short*), $[10, 100]$ms (*moderate*), or $[50, 500]$ms (*long*). We set each task's WCET C_i to $T_i \cdot u_i$ rounded to the next microsecond.

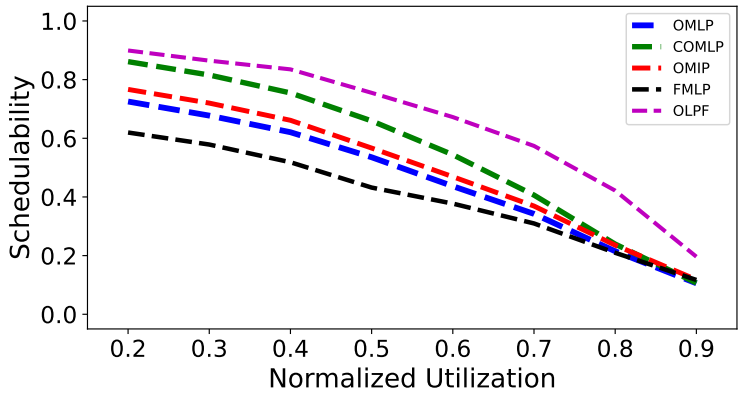
We considered $\{M/4, M/2, M, 2M\}$ number of shared resources. For each τ_i and resource ℓ_q, we selected τ_i to access resource ℓ_q with probability $p^{acc} \in \{0.1, 0.25, 0.5\}$. If so selected, τ_i was defined to access ℓ_q via $N^q_i \in \{1, 2, \ldots, 5\}$ requests. For each $N^q_i > 0$, we chose the maximum request length L^q_i randomly from three uniform distributions ranging over $[1, 15]\mu$s (*short*), $[1, 100]\mu$s (*medium*), or $[5, 1280]\mu$s (*long*). A chosen L^q_i value was decreased accordingly if it caused the sum of all request length of τ_i to exceed C_i. For each combination of M, normalized utilization, T_i, L^q_i, p^{acc}, and n_r, we generated 1,000 task systems. We call each combination of these parameters a *scenario*.

Experiment 1. In our first experiment, we considered mutex sharing. Each task had a *soft* timing constraint, meaning that it was deemed schedulable if its response time was bounded. We considered resource synchronization under the OLP-F, the OMLP [Brandenburg and Anderson, 2010a], the C-OMLP [Brandenburg and Anderson, 2011], the OMIP [Brandenburg, 2013a], and the FMLP [Block et al., 2007]. For the OLP-F, each task system's schedulability was tested under global G-FIFO scheduling [Leontyev and Anderson, 2007]. For the remaining protocols, s-oblivious schedulability tests were performed under G-EDF scheduling [Devi and Anderson, 2008].[6] For each scenario, we assessed *acceptance ratios*, which give the percentage of task

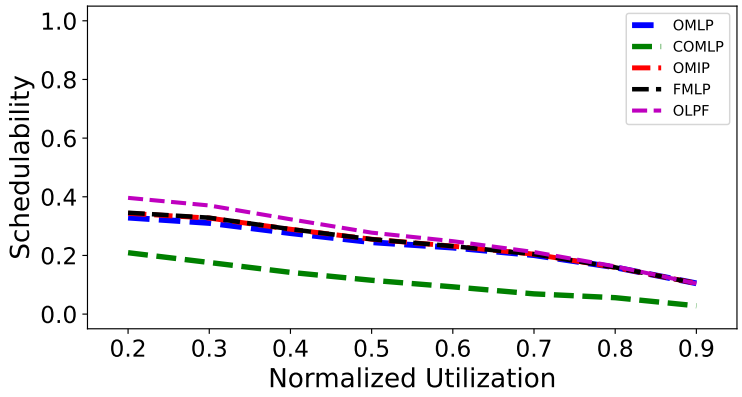
[6]The same schedulability test also applies for a wider class of global schedulers including G-FIFO.



(a) Experiment 1 with moderate periods, medium requests, $p^{acc} = 0.1$, $n_r = M/4$.



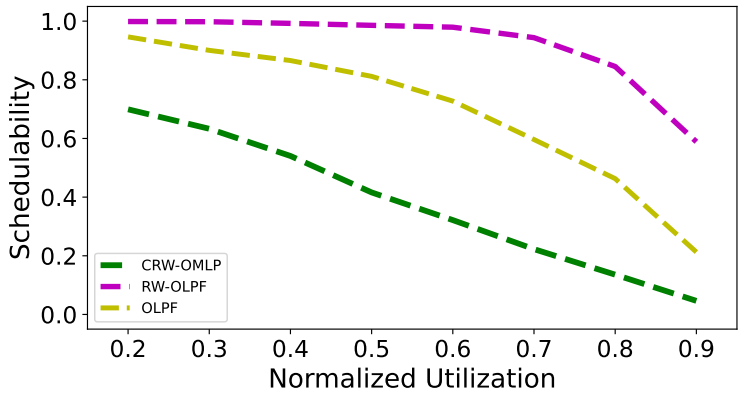
(b) Experiment 1 with short periods, short requests, $p^{acc} = 0.25$, $n_r = M/2$.



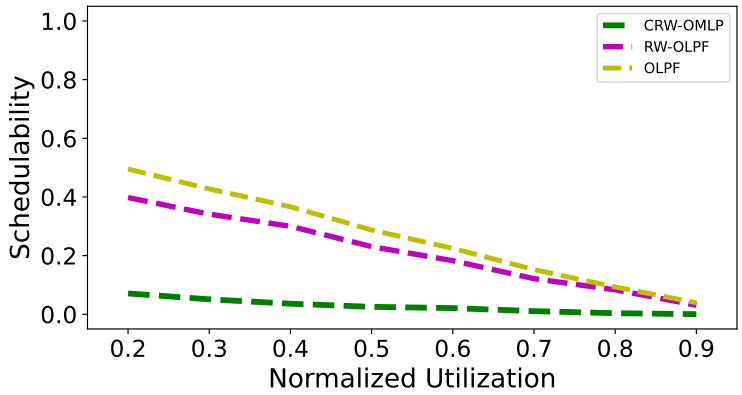
(c) Experiment 1 with moderate periods, long requests, $p^{acc} = 0.1$, and $n_r = M/4$.

Figure 5.10: Experiment 1 results.

systems that were schedulable under each locking protocol. We present a representative selection of our results in Figure 5.10.



(a) Experiment 2 with long periods, medium requests, $p^{acc} = 0.2$, $p^{write} = 0.7$, $n_r = M$.



(b) Experiment 2 with long periods, long requests, $p^{acc} = 0.5$, $p^{write} = 0.1$, $n_r = M$.

Figure 5.11: Experiment 2 results.

Observation 5.1. *The average improvement under the OLP-F over the OMLP, the C-OMLP, the OMIP, and the FMLP was 20.2%, 14.9%, 16.4%, and 27.5%, respectively.*

This can be seen in insets (a)–(c) of Figure 5.10. Unsurprisingly, schedulability improved under the OLP-F because of lower pi-blocking compared to the other protocols. In some cases, as depicted in Figure 5.10(c), all protocols had similar schedulability. This can occur when the number of request-issuing jobs for each resource is small (*e.g.,* fewer than the number of processors), in which case all protocols have similar pi-blocking bounds.

Experiment 2. This experiment pertains to RW sharing. To generate task systems, we used an additional parameter $p^{write} \in \{0.1, 0.2, 0.3, 0.5, 0.7\}$. We defined each resource access to be a write (resp., read) access with probability p^{write} (resp., $1 - p^{write}$). In this experiment, we considered SRT scheduling with

resource synchronization under the RW-OLP-F, the CRW-OMLP [Brandenburg and Anderson, 2011], and the OLP-F. Each task system's schedulability was tested under global G-FIFO scheduling when the OLP-F and the RW-OLP-F were employed, and under global EDF scheduling otherwise. We have the following observation.

Observation 5.2. *The RW-OLP-F improved schedulability over the CRW-OMLP across all scenarios. The RW-OLP-F had less schedulability than the OLP-F when write accesses were more frequent, i.e., high p^{write} values.*

This can be seen in insets (a) and (b) of Figure 5.11. The improved pi-blocking bound enabled higher schedulability under the RW-OLP-F. The RW-OLP-F had better or equal schedulability than the OLP-F across 90% of the total scenarios. Since the RW-OLP-F has higher write request blocking compared to the OLP-F (which does not have optimal read request blocking), the OLP-F had better schedulability than the RW-OLP-F when p^{write} values were high, *e.g.,* $p^{write} = 0.7$.

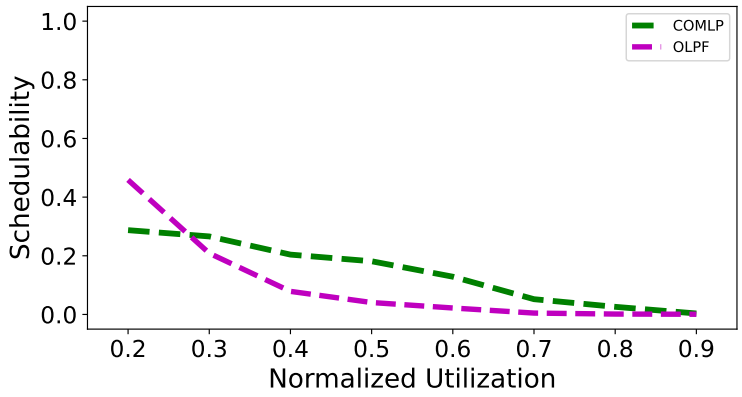
Experiment 3. In this experiment, we considered HRT scheduling under mutex locks. For each task τ_i, we randomly chose a relative deadline between $[T_i, 2T_i]$. We considered partitioned scheduling because of the lack of HRT schedulability tests for global G-FIFO scheduling. We used the *worst-fit* bin-packing heuristic to partition each task system. We compared schedulability under the OLP-F and partitioned G-FIFO scheduling with the partitioned OMLP (the C-OMLP with $c = 1$) and partitioned EDF scheduling.

Observation 5.3. *The partitioned OMLP had better schedulability compared to the OLP-F.*

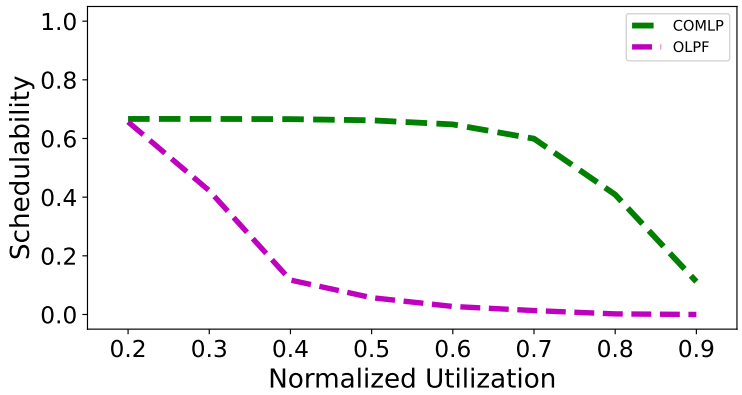
This can be seen in Figure 5.12. Despite having lower pi-blocking and bounded response times, the partitioned OMLP enabled better schedulability because of the optimality of uniprocessor EDF in scheduling HRT workloads. Note that, unlike for EDF, the employed G-FIFO schedulability test was non-exact [Bedarkar et al., 2022].

5.5 Chapter Summary

In this chapter, we have closed a long-standing open problem concerning pi-blocking optimality. In particular, we have presented lower-bound results showing that the factor of two present in the pi-blocking bounds of most real-time multiprocessor mutex protocols that are asymptotically optimal under s-oblivious analysis is fundamental when non-FIFO JLFP scheduling is used. In presenting these results, we have



(a) Experiment 3 with long periods, medium requests, $p^{acc} = 0.5$, $n_r = M/4$.



(b) Experiment 3 with long periods, short requests, $p^{acc} = 0.1$, $n_r = 2M$.

Figure 5.12: Experiment 3 results.

assumed global scheduling. As global scheduling is a special case of clustered scheduling, our results are applicable to locking protocols that target clustered scheduling as well.

For clustered FIFO scheduling, we have presented optimal suspension-based multiprocessor locking protocols for mutex and k-exclusion, and an almost-optimal protocol for RW synchronization. In particular, we have shown that the s-oblivious lower bound of $M - 1$ request lengths for mutex locks is indeed tight under FIFO scheduling. We have also provided s-oblivious lower-bound results on read-request blocking for RW locks.

CHAPTER 6: SOFT REAL-TIME SCHEDULING OF GANG TASKS[1]

In this chapter, we consider the SRT scheduling of rigid gang tasks on identical multiprocessors. Such tasks consist of multiple threads that are co-scheduled. These co-scheduling constraints can cause a system to have idle processors even when a task is ready to execute, complicating the design and analysis of scheduling algorithms for rigid gang tasks. For SRT gang scheduling, this becomes even more complicated, as multiple jobs of a task can be present concurrently.

Consequently, the lone work on SRT gang scheduling [Dong et al., 2021] gives a sufficient condition for SRT-schedulability for gang tasks under G-EDF scheduling. Thus, the exact SRT-feasibility condition is unknown for gang tasks (unlike for sequential and DAG tasks). Motivated by this, we consider the SRT-feasibility problem for preemptive gang scheduling, which asks whether a given gang task system can be scheduled such that each instance of a task has bounded response time. We give necessary and sufficient conditions for SRT feasibility, based on which we show that the SRT-feasibility problem is NP-hard. To the best of our knowledge, this is the first intractability result regarding SRT scheduling where only bounded response times are required. Furthermore, we give server-based scheduling policies for gang tasks and corresponding schedulability conditions for bounded response times. We also show that G-EDF is non-optimal in scheduling SRT gang tasks and give an improved condition for achieving bounded response times under G-EDF.

Organization. In the rest of this chapter, we first cover needed background (Section 6.1). Next, we discuss the SRT-feasibility of gang tasks (Section 6.2) and introduce our server-based scheduling policies (Section 6.3). Then, we discuss G-EDF scheduling of gang tasks (Section 6.4). Finally, we present our experiments (Section 6.5), and conclude with a summary (Section 6.6).

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

Ahmed, S. and Anderson, J. (2023b), Soft Real-Time Gang Scheduling, *Proceedings of the 44th IEEE Real-Time Systems Symposium*, pages 331-343.

6.1 System Model

We consider a system Γ of N sporadic rigid gang tasks that are globally scheduled on M identical processors. Each gang task τ_i releases a potentially infinite sequence of jobs $\tau_{i,1}, \tau_{i,2}, \ldots$. Each sporadic (resp., periodic) gang task τ_i has a *period* T_i, which is the minimum (resp., exact) separation time between any two consecutive job releases of τ_i. The *relative deadline* of τ_i is denoted by D_i. We consider implicit deadlines, meaning that $D_i = T_i$ holds for each τ_i. However, deadlines are soft, meaning that jobs are allowed to miss their deadlines. Task τ_i has a WCET of C_i. Each task τ_i has a *degree of parallelism* m_i, which is the number of simultaneously available processors required to execute any job of τ_i. Thus, the *worst-case execution requirement* (WCER) of each job of τ_i can be represented by a rectangle of area $m_i \times C_i$ in a schedule. We let $C_{max} = \max_i\{C_i\}$, $C_{min} = \min_i\{C_i\}$, and $T_{max} = \max_i\{T_i\}$. Jobs of τ_i are sequential, meaning that no two jobs of τ_i can execute in parallel. We also assume that jobs are preemptive.

Note the difference between the degree of parallelism m_i of a gang task and the parallelization level P_i of a task under the rp model. The former pertains to the number of processors required by a single job of a gang task, while the latter refers to how many successive jobs of a task are allowed to execute in parallel under the rp model. In this chapter, we assume that such concurrent execution of successive jobs of a task is prohibited, *i.e.,* a job cannot commence execution until all prior jobs of the task finish execution. Thus, for each task τ_i, $P_i = 1$ holds.

The *utilization* of τ_i is $u_i = (C_i \times m_i)/T_i$. Note that u_i can exceed 1.0 for a gang task τ_i. The *total utilization* of task system Γ is $U_{tot} = \sum_{i=1}^{N} u_i$. The *horizontal utilization* σ_i of τ_i is C_i/T_i. The *hyperperiod* H is the LCM of all periods. We let h_i denote H/T_i.

Similar to sequential tasks, a periodic gang task τ_i has an offset Φ_i that denotes the release time of the first job of τ_i. For brevity, we denote a periodic (resp., sporadic) gang task by (Φ_i, T_i, C_i, m_i) (resp., (T_i, C_i, m_i)). We summarize all introduced notation in Table 6.1.

Parallelism-induced idleness. When scheduling gang tasks, *parallelism-induced idleness* may occur. Formally, such idleness is defined as follows.

Definition 6.1. A time instant t is *parallelism-induced* idle if there is an idle processor at time t, and a job $\tau_{i,j}$ is pending but unscheduled at time t due to the lack of m_i available processors. ◀

The following example illustrates parallelism-induced idleness.

Table 6.1: Notation summary for Chapter 6.

Symbol	**Meaning**
Γ	Task system
N	Number of tasks
M	Number of processors
Γ^{kH}	Definition 6.2
τ_i	i^{th} task
T_i	Period of τ_i
C_i	WCET of τ_i
Φ_i	Offset of τ_i
u_i	Utilization of τ_i
m_i	Degree of parallelism of τ_i
S_i^H	Server associated with τ_i
U_{tot}	Utilization of Γ
T_{max}	$\max_i\{T_i\}$
C_{max}	$\max_i\{C_i\}$
C_{min}	$\min_i\{C_i\}$
H	Hyperperiod
h_i	H/T_i
$\tau_{i,j}$	j^{th} job of τ_i
$r(\tau_{i,j})$	Release time of $\tau_{i,j}$
$f(\tau_{i,j})$	Completion time of $\tau_{i,j}$
$d(\tau_{i,j})$	PP of $\tau_{i,j}$
(Φ_i, T_i, C_i, m_i)	Periodic task τ_i
(T_i, C_i, m_i)	Sporadic task τ_i
$\mathcal{S}$	An arbitrary schedule
$\mathcal{I}$	Ideal schedule
$\mathsf{A}(\tau_i, t, t', \mathcal{S})$	Allocation of τ_i in $\mathcal{S}$
$\mathsf{A}(\Gamma, t, t', \mathcal{S})$	Allocation of Γ in $\mathcal{S}$
$\mathsf{lag}(\tau_i, t, \mathcal{S})$	lag of τ_i in $\mathcal{S}$ (6.3)
$\mathsf{LAG}(\tau_i, t, \mathcal{S})$	LAG of Γ in $\mathcal{S}$ (6.5)
$\mathcal{U}_b$	Definition 6.5

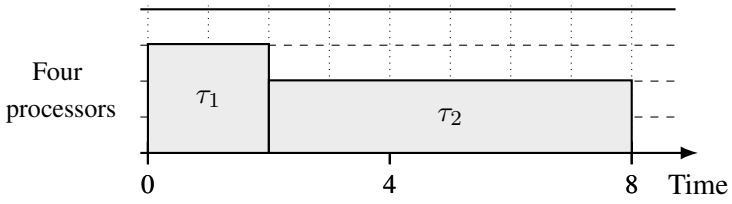


Figure 6.1: Two gang tasks on four processors. The number inside each execution block denotes the degree of parallelism. Both tasks release a job at time 0.

Example 6.1. In Figure 6.1, there is an idle processor during the time interval $[0, 2)$. Although τ_2 has a pending job during this interval, it cannot execute, as the number of available processors is less than m_2. Thus, there is parallelism-induced idleness during $[0, 2)$. ◀

A sporadic gang task system can have many instantiations where release times and execution times of all jobs are specified. Recall from Chapter 1 that

$$\text{SRT-feasibility of } \Gamma \iff \text{Bounded response times for all instantiations of } \Gamma \text{ by some scheduler.} \quad (6.1)$$

6.2 SRT-Feasibility of Gang Tasks

In this section, we consider the problem of determining the SRT-feasibility of a gang task system. In this section, we assume the following, which we justify in Lemma 6.2 in the context of SRT-feasibility.

Assumption 6.1. Each job of any task τ_i executes for its WCET C_i.

Lemma 6.1. *If all jobs of an instantiation Γ_c of Γ have bounded response times under a scheduler when Assumption 6.1 is satisfied, then for any instantiation Γ'_c that differs from Γ_c only in job execution times, all jobs have bounded response times under some scheduler.*

Proof. Let $\mathcal{S}$ be a schedule of Γ_c in which all jobs have bounded response times. Using $\mathcal{S}$, we construct a schedule $\mathcal{S}'$ of Γ'_c, in which all jobs of Γ'_c have bounded response times. In $\mathcal{S}'$, every job $\tau_{i,j}$ is scheduled whenever it is scheduled in $\mathcal{S}$ until it completes in $\mathcal{S}'$. If a job executes for less than its WCET and finishes at time t' in $\mathcal{S}'$, then $\mathcal{S}'$ keeps the processors the job occupies in $\mathcal{S}$ idle at time instants after t'. Thus, each job has a bounded response time, as it finishes no later in $\mathcal{S}'$ than in $\mathcal{S}$. □

Lemma 6.2. *If every instantiation of Γ satisfying Assumption 6.1 has bounded response times under some algorithm, then Γ is SRT-feasible.*

Proof. Assume that Γ is not SRT-feasible. Then, there exists an instantiation Γ'_c of Γ that is not SRT-schedulable by any algorithm (by (6.1)). By the assumption of the lemma, Γ'_c does not satisfy Assumption 6.1. Let Γ_c be the instantiation of Γ such that Γ_c satisfies Assumption 6.1, and it only differs from Γ'_c in terms of job execution times. Since Γ_c satisfies Assumption 6.1, it is SRT-schedulable by some algorithm (by the lemma's assumption). Thus, by Lemma 6.1, Γ'_c is SRT-schedulable by some algorithm, a contradiction. □

6.2.1 Necessary Condition for SRT-Feasibility

We give a necessary condition for SRT-feasibility in Lemma 6.3, which utilizes the following definition.

Definition 6.2. Given the *sporadic* task system $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}$, let $\Gamma^{kH} = \{\tau_1^{kH}, \tau_2^{kH}, \ldots, \tau_N^{kH}\}$ be a set of implicit-deadline *periodic* gang tasks such that k is a positive integer and $\tau_i^{kH} = (0, kH, kh_iC_i, m_i)$.◀

Lemma 6.3. *If Γ is SRT-feasible, then there is a positive integer k such that the periodic task system Γ^{kH} is HRT-feasible.*

Proof. Let Γ_c be an instantiation of Γ such that each task periodically releases its jobs starting from time 0. Since Γ is SRT-feasible, there is a schedule $\mathcal{S}$ of Γ_c, in which each task τ_i has bounded tardiness (and response times). Let x_i be τ_i's tardiness in $\mathcal{S}$. Let $e_i(t) \geq 0$ be the remaining execution time of all jobs of τ_i released *before* time t in $\mathcal{S}$.

We now consider the $e_i(t)$ values at times $0, H, 2H, \ldots$. In Γ_c, each task τ_i releases a job at time $t \in \{0, H, 2H, \ldots\}$. By the definition of $e_i(t)$, at any time $t \in \{0, H, 2H, \ldots\}$, $e_i(t)$ does not include the execution time of τ_i's job released at time t. Since τ_i's tardiness is x_i, at any time $t \in \{0, H, 2H, \ldots\}$, τ_i's jobs released before time t have at most x_i time units of execution remaining at time t. (Note that τ_i's jobs must execute in sequence, so if its tardiness is x_i, then all of its tardy jobs at a hyperperiod boundary must be complete within x_i time units beyond that boundary.) Thus, at any time $t \in \{0, H, 2H, \ldots\}$, $0 \leq e_i(t) \leq x_i$ holds for all i.

Therefore, since $e_i(t)$ is an integer, $e_i(t)$ can take at most $x_i + 1$ distinct values at any time $t \in \{0, H, 2H, \ldots\}$. Thus, the tuple $(e_1(t), e_2(t), \ldots, e_n(t))$ takes on one of $X = \prod_{i=1}^{N}(x_i + 1)$ distinct values at any time $t \in \{0, H, 2H, \ldots\}$. Therefore, there must be a pair of integers $a < b \leq X + 1$ such that $(e_1(aH), e_2(aH), \ldots, e_n(aH)) = (e_1(bH), e_2(bH), \ldots, e_n(bH))$ holds. Let $k = b - a$.

Since τ_i releases jobs periodically in Γ_c, it releases $(b - a) \cdot \frac{H}{T_i} = kh_i$ jobs in $[aH, bH)$. Thus, by Assumption 6.1, during $[aH, bH)$, τ_i executes for $e_i(aH) + kh_iC_i - e_i(bH) = kh_iC_i$ time units. Let $\mathcal{S}_k$ be

the portion of schedule $\mathcal{S}$ during $[aH, bH)$. Using $\mathcal{S}_k$, we can create an HRT-feasible schedule $\mathcal{S}^{kH}$ of Γ^{kH}. $\mathcal{S}^{kH}$ mimics $\mathcal{S}_k$ with the exception that τ_i^{kH} executes in $\mathcal{S}^{kH}$ instead of τ_i whenever τ_i is scheduled in $\mathcal{S}_k$. Since every task τ_i is scheduled for kh_iC_i time units in $\mathcal{S}_k$, τ_i^{kH} is scheduled for its WCET in $\mathcal{S}^{kH}$. Thus, there exists an HRT-feasible schedule of Γ^{kH}. Note that tasks in Γ^{kH} have implicit deadlines with the same period, and release their first jobs synchronously at time 0. □

6.2.2 Sufficient Condition for SRT-Feasibility

In this section, we give a sufficient SRT-feasibility condition for gang tasks as shown in Lemma 6.4.

Lemma 6.4. *If there is a periodic task system Γ^{kH}, as defined in Definition 6.2, that is HRT-feasible, then the corresponding sporadic task system Γ is SRT-feasible.*

Note that we do not require the same value of k in both Lemmas 6.3 and 6.4. To prove Lemma 6.4, we give a server-based scheduling policy for Γ based on an HRT-feasible schedule of Γ^H. For ease of notation, we prove the lemma for Γ^H. We begin by defining reservation servers. Recall that a similar concept was introduced in Chapter 4 for DAG tasks.

Reservation servers. For each task τ_i, we define a periodic *reservation server* S_i^H. We denote the set of all servers as Γ_s^H. Each server S_i^H has a period $T_i^H = H$, a *horizontal budget* $C_i^H = h_iC_i$, and a degree of parallelism of $m_i^H = m_i$. The *total budget*, also called the *budget*, of S_i^H is $m_iC_i^H$. Thus, by Definition 6.2 (with $k = 1$), S_i^H and τ_i^H have the same period and degree of parallelism.

> **Replenishment Rule.** For any non-negative integer ℓ, the budget of S_i^H is replenished to $m_iC_i^H$ at time ℓH.
>
> **Consumption Rule.** S_i^H consumes budget at the rate of m_i execution units per unit of time when it is scheduled until its budget is exhausted.

Scheduling servers. Servers are scheduled according to the following rule.

P. Let $\mathcal{S}^H$ be an HRT schedule of Γ^H where each job of any task Γ^H executes for its WCET. The servers in Γ_s^H are scheduled according to $\mathcal{S}^H$, *i.e.,* server S_i^H is scheduled at time t if and only if τ_i^H is scheduled at time t in $\mathcal{S}^H$.

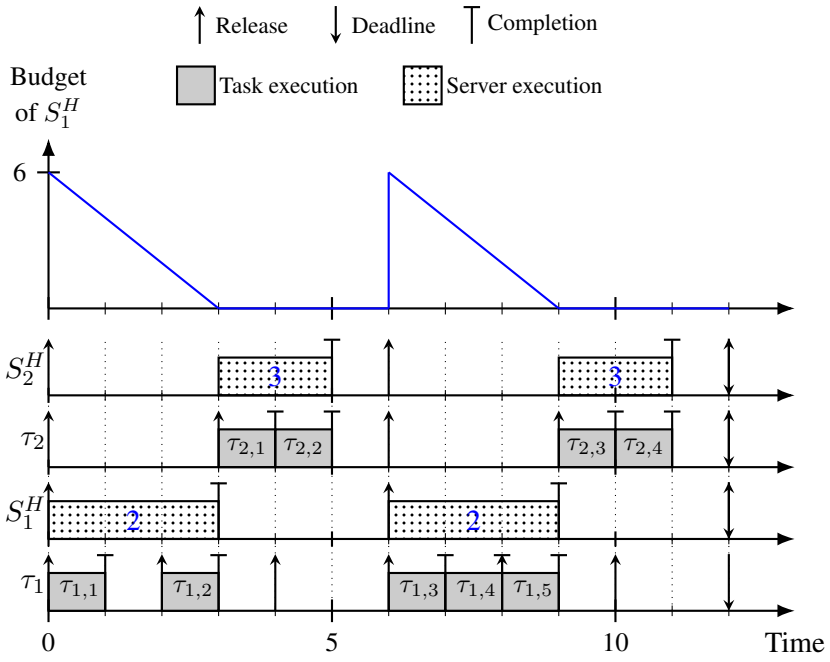


Figure 6.2: Example server-based scheduling. The numbers inside server execution boxes denote m_i values.

Example 6.2. Assume that Γ consists of two gang tasks $\tau_1 = (2, 1, 2)$ and $\tau_2 = (3, 1, 3)$ that are scheduled on a four-processor platform. Since the task periods are 2.0 and 3.0 time units, we have $H = 6$, $h_1 = 6/2 = 3$, and $h_2 = 6/3 = 2$. Thus, by Definition 6.2, Γ^H consists of periodic tasks $\tau_1^H = (0, 6, 3, 2)$ and $\tau_2^H = (0, 6, 2, 3)$. As shown in Figure 6.2, there is an HRT-feasible schedule of Γ^H, which, by Rule P, is also the server schedule of S_1^H and S_2^H.

At time 0, the budget of S_1^H is replenished to $m_2 C_2^H = 2 \cdot 3 = 6$. During the time interval $[0, 3)$, S_1^H is scheduled; hence, its budget is consumed at the rate of 2.0 units per unit of time during $[0, 3)$. Thus, S_1^H's budget is exhausted at time 3. ◀

For ease of notation, we also denote the server schedule by $\mathcal{S}^H$. By Rule P, we have the following lemma, which shows that S_i^H has sufficient budget to be scheduled whenever τ_i^H is scheduled in $\mathcal{S}^H$.

Lemma 6.5. *If τ_i^H is scheduled in $\mathcal{S}^H$ at time t, then S_i^H has at least m_i units of budget remaining at time t.*

Proof. Assume that t is the first time instant when S_i^H has less than m_i units of remaining budget, but τ_i^H is scheduled. Let ℓ be the non-negative integer such that $t \in [\ell H, (\ell + 1)H)$. By the Replenishment Rule, S_i^H's budget is $m_i C_i^H$ at time ℓH. By the Consumption Rule, S_i^H's budget is consumed at the rate of m_i units per unit of time when it is scheduled. Thus, the remaining budget at time t is an integer multiple of m_i. Therefore, at time t, S_i^H's budget is at most 0.

By the Consumption Rule, S_i^H is scheduled for at least $\frac{m_i C_i^H}{m_i} = C_i^H$ time units during $[\ell H, t)$. Thus, by Rule P, τ_i^H is scheduled for at least C_i^H time units during $[\ell H, t)$. By Definition 6.2, $\tau_i^H \in \Gamma^H$ releases its $(\ell+1)^{st}$ job at time ℓH. Thus, by the definition of $\mathcal{S}^H$, the $(\ell+1)^{st}$ job of τ_i^H completes by time t, as τ_i^H is scheduled for at least C_i^H time units during $[\ell H, t)$. Since $t < (\ell+1)H$, there is no ready job of τ_i^H at time t. Thus, τ_i^H cannot be scheduled at time t. Contradiction. □

We now prove the following lemma, which we will later use to prove Lemma 6.4.

Lemma 6.6. *For any non-negative integer ℓ, server S_i^H is scheduled for $C_i^H = h_i C_i$ time units during any time interval $[\ell H, (\ell+1)H)$ in $\mathcal{S}^H$.*

Proof. By Lemma 6.5, S_i^H has at least m_i units of remaining budget at any time when τ_i^H is scheduled. Therefore, S_i^H can be scheduled whenever τ_i^H is scheduled.

By Definition 6.2, $\tau_i^H \in \Gamma^H$ releases its $(\ell+1)^{st}$ job at time ℓH. By the definition of $\mathcal{S}^H$ and Definition 6.2, the $(\ell+1)^{st}$ job of τ_i^H is scheduled for its WCET of $h_i C_i$ time units and completes by time $(\ell+1)H$ in $\mathcal{S}^H$. Thus, S_i^H is scheduled for $h_i C_i$ time units during $[\ell H, (\ell+1)H)$ in $\mathcal{S}^H$. □

Scheduling tasks on servers. Jobs of the sporadic tasks in Γ are scheduled on servers via the following rules.

R1. Jobs of τ_i are scheduled on server jobs of S_i^H.

R2. If server S_i^H is scheduled and job $\tau_{i,j}$ is ready at time t, then $\tau_{i,j}$ is scheduled on the processors on which S_i^H is scheduled at time t.

Example 6.2 (Continued)**.** Consider the scheduling of τ_1 and τ_2 in Figure 6.2. At time 0, τ_1 has a ready job $\tau_{1,1}$ and S_1^H is scheduled. Thus, by Rule R2, $\tau_{1,1}$ is scheduled at time 0. At time 1, there is no pending job of task τ_1. Thus, despite S_1^H being scheduled at time 1, no job of τ_1 is scheduled at time 1. ◀

We now show that each task τ_i has a bounded response time if $\mathcal{S}^H$ is an HRT-feasible schedule of Γ^H. We first show, in Lemma 6.7, that the jobs released during $(\ell H, (\ell+1)H]$ complete execution by time $(\ell+2)H$. Using this lemma, we will then derive a response-time bound of τ_i in Lemmas 6.8 and 6.9.

Lemma 6.7. *If servers are scheduled according to Rule P and tasks are scheduled according to Rules R1 and R2, then, for any non-negative integer ℓ, any job $\tau_{i,j}$ released during $(\ell H, (\ell+1)H]$ completes execution at or before time $(\ell+2)H$.*

Proof. Assume otherwise. Let ℓ be the smallest non-negative integer such that there is a job $\tau_{i,j}$ released during $(\ell H, (\ell+1)H]$ that completes execution after time $(\ell+2)H$. Thus, any job released during time interval $((\ell-1)H, \ell H]$ completes execution at or before time $(\ell+1)H$. Therefore, no job released at or before time ℓH is pending at or after time $(\ell+1)H$.

Let e_i be the remaining execution time of τ_i's jobs that are released during $(\ell H, (\ell+1)H]$ at time $(\ell+1)H$. Since τ_i releases its job sporadically, at most $H/T_i = h_i$ jobs of τ_i are released during $(\ell H, (\ell+1)H]$. Therefore, $e_i \leq h_i C_i$. By Lemma 6.6, S_i^H is scheduled for $h_i C_i$ time units during $[(\ell+1)H, (\ell+2)H)$. Since jobs of τ_i execute sequentially and $L \leq h_i C_i$, by Rule R2, all job released during $(\ell H, (\ell+1)H]$ must complete execution by time $(\ell+2)H$, a contradiction. □

Lemma 6.8. *Let $\tau_{i,j}$ be the q^{th} job of τ_i among τ_i's jobs that are released during $(\ell H, (\ell+1)H]$ where $q \leq h_i$ and ℓ is a non-negative integer. If servers are scheduled according to Rule P and tasks are scheduled according to Rules R1 and R2, then $\tau_{i,j}$'s response time is at most $2H - (h_i - q)C_i - (q-1)T_i$.*

Proof. We first prove that S_i^H is scheduled for at least qC_i time units during $[(\ell+1)H, (\ell+2)H-(h_i-q)C_i)$. Assume otherwise. During $[(\ell+2)H-(h_i-q)C_i, (\ell+2)H)$, S_i^H can be scheduled for at most $(h_i-q)C_i$ time units. Thus, S_i^H is scheduled during $[(\ell+1)H, (\ell+2)H)$ for less than $qC_i + (h_i-q)C_i = h_i C_i$ time units, contradicting Lemma 6.6.

By Lemma 6.7, no job released at or before time ℓH is pending after time $(\ell+1)H$. The total execution time of the first q jobs of τ_i released during $(\ell H, (\ell+1)H]$ is at most qC_i. Therefore, by Rule R2, $\tau_{i,j}$ completes execution at or before time $(\ell+2)H-(h_i-q)C_i$, as S_i^H is scheduled for at least qC_i time units during $[(\ell+1)H, (\ell+2)H-(h_i-q)C_i)$.

Since τ_i releases jobs sporadically, we have $r_{i,j} \geq \ell H + (q-1)T_i$. Therefore, $\tau_{i,j}$'s response time is at most $(\ell+2)H-(h_i-q)C_i-\ell H-(q-1)T_i = 2H-(h_i-q)C_i-(q-1)T_i$ time units. □

Lemma 6.9. *If servers are scheduled according to Rule P and tasks are scheduled according to Rules R1 and R2, then task τ_i's response time is at most $2H-(h_i-1)C_i$.*

Proof. Let $\tau_{i,j}$ be an arbitrary job of τ_i. Assume that $\tau_{i,j}$ is released during $(\ell H, (\ell+1)H]$ where ℓ is a non-negative integer and $\tau_{i,j}$ is the q^{th} job among τ_i's jobs that are released during $(\ell H, (\ell+1)H]$. By Lemma 6.8, $\tau_{i,j}$'s response time is at most $2H-(h_i-q)C_i-(q-1)T_i = 2H-(h_i-1)C_i+(q-1)(C_i-T_i)$. Since $q \geq 1$ and $C_i \leq T_i$, $\tau_{i,j}$'s response time is at most $2H-(h_i-1)C_i$. Thus, the lemma holds. □

By Lemma 6.9, Γ is SRT-feasible. This proves Lemma 6.4.

The response-time bound in Lemma 6.9 can be exponential with respect to the task count. However, systems with pseudo-harmonic periods, where $H = T_{max}$ holds, the response-time bound is less than $2T_{max}$.

We now prove the following theorem using the conditions derived in Sections 6.2.1 and 6.2.2.

Theorem 6.1. *Determining the SRT-feasibility of a set of gang tasks is NP-hard.*

Proof. The proof is via reduction from the partition problem.

The partition problem. Given a set $A = \{a_1, a_2, \cdots, a_p\}$ of positive integers with $\sum_{i=1}^{p} a_i = 2B$, the partition problem asks whether A can be partitioned into two equal-sum subsets A_1 and A_2, *i.e.,* $\sum_{a \in A_1} = \sum_{a \in A_2} = B$.

Reduction. Let $A = \{a_1, a_2, \cdots, a_p\}$ with $\sum_{i=1}^{p} a_i = 2B$ be an arbitrary instance of the partition problem. We construct an instance of the SRT-feasibility problem as follows. Let Γ be a set of p sporadic gang tasks to be scheduled on B processors. Task $\tau_i \in \Gamma$ has a period of 2.0 time units, a WCET of 1.0 time unit, and $m_i = a_i$.

We now prove that A can be partitioned into two equal-sum subsets if and only if there exists a schedule $\mathcal{S}$ of Γ on B processors where each task has a bounded response time.

Sufficiency. Assume that A can be partitioned into two equal-sum subsets A_1 and A_2. We will prove that Γ^H is HRT-feasible, which, by Lemma 6.4, implies that each task in Γ has a bounded response time under some scheduling algorithm. Since each task's period is 2.0 time units, we have $H = 2.0$. Therefore, by Definition 6.2, Γ^H consists of p tasks such that $\tau_i^H = (0, 2, 1, m_i)$. Since tasks in Γ^H are implicit-deadline periodic tasks, it is sufficient to show that there exists a schedule such that the first jobs of all tasks in Γ^H complete by time 2.0. We construct such an HRT-feasible schedule $\mathcal{S}^H$ as follows. $\mathcal{S}^H$ schedules tasks corresponding to subset A_1 (resp., A_2) during time interval $[0, 1)$ (resp., $[1, 2)$). Since $\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i = B$ and $m_i = a_i$ for all i, exactly B processors execute jobs of Γ^H at any time during $[0, 2)$. Since $A_1 \cup A_2 = A$ and $A_1 \cap A_2 = \emptyset$, each task in Γ^H is scheduled for exactly 1.0 time unit in $\mathcal{S}^H$. Thus, in $\mathcal{S}^H$, the first jobs of all tasks in Γ^H complete by time 2.0.

Necessity. Assume that there is a schedule $\mathcal{S}$ of Γ on B processors where each task in Γ has a bounded response time. Then, by Lemma 6.3 and Definition 6.2, there exists a positive integer k and an HRT-feasible task system Γ^{kH} consisting of tasks $\tau_i = (0, 2k, k, m_i)$. Since tasks in Γ^{kH} are periodic and have implicit

deadlines, there is a schedule $\mathcal{S}^{kH}$ of Γ^{kH} on B processors where the first jobs of all tasks in Γ^{kH} complete at or before time $2k$, *i.e.,* each task executes for k time units during time interval $[0, 2k)$.

We first show that there is no idle processor in $\mathcal{S}^{kH}$ at any time instant during $[0, 2k)$. Assume otherwise. Since there is at least one idle processor during a unit-sized time interval, the total execution of tasks in Γ^{kH} is at most $2kB - 1$ units. The total execution requirement of the first jobs of tasks in Γ^{kH} is $\sum_{i=1}^{p} k \cdot m_i = k \sum_{i=1}^{p} m_i = k \sum_{i=1}^{p} a_i = 2kB$. Thus, at least one task's first job does not complete execution by time $2k$ in $\mathcal{S}^{kH}$ and $\mathcal{S}^{kH}$ cannot be an HRT-feasible schedule of Γ^{kH}, a contradiction.

We now show that there exists a partition of A into two equal-sum subsets. Let Q be the set of tasks that are scheduled during $[0, 1)$ in $\mathcal{S}^{kH}$. Since all B processors are busy during $[0, 1)$, we have $\sum_{\tau_i^{kH} \in Q} m_i = B$. Let A_1 be the subset of A that consists of elements a_i corresponding to tasks in Q. Thus, $\sum_{a_i \in A_1} a_i = B$. Since $\sum_{a_i \in A} a_i = 2B$, we have $\sum_{a_i \in A \setminus A_1} a_i = B$. Thus, there exists a partition of two equal-sum subsets of A. □

6.3 Schedulability Under Server-Based Scheduling

In the previous section, we gave a server-based approach for scheduling gang tasks. We showed that if servers can be scheduled to meet their deadlines, then the gang tasks in Γ have bounded response times under the server-based scheduling policy. However, we relied on an HRT schedule of the servers (Rule P). Unfortunately, obtaining such a schedule is NP-hard in the strong sense [Kubale, 1987]. In this section, we provide some scheduling policies and corresponding exact HRT-schedulability tests for servers, which provide a sufficient SRT-feasibility tests for gang tasks according to Lemma 6.4.

Referring to the server-based scheme used to prove this lemma, it is important to note that the HRT-schedulability of the servers in Γ_s^H under a given scheduling policy can be different from HRT-schedulability of the tasks in Γ^H under that policy. This is because a server S_i^H is required to be scheduled for exactly C_i^H time units during $[0, H)$, as this ensures that τ_i receives a sufficient processor allocation during $[0, H)$. In contrast, for Γ^H, τ_i^H can execute for less than its WCET C_i^H, which can cause timing anomalies by causing some jobs to miss their deadlines. Thus, for servers, only the case where S_i^H is scheduled for exactly C_i^H time units during $[0, H)$ needs to be considered, which may not be sufficient for the HRT-schedulability of Γ^H under the same scheduling.

Algorithm 6.1 FP-scheduling of servers.

Variables:

$\mathcal{O}$: A priority ordering

Sched(t) : Set of jobs to be scheduled at time t

```
1: procedure FP
2:     M' ← M
3:     Order servers according to O
4:     for each S_i^H ∈ Γ^H do
5:         if m_i ≤ M' and S_i^H's remaining budget > 0 then
6:             Sched(t) ← Sched(t) ∪ {S_i^H}
7:             M' ← M' − m_i
```

6.3.1 FP Scheduling of Servers

Under FP scheduling, each server has a fixed priority. At any time instant, the highest-priority servers that can execute together (without requiring more than M processors) are scheduled as in Algorithm 6.1. As all servers are replenished synchronously every H time units, G-FIFO and implicit-deadline G-EDF scheduling (each with *consistent* tie-breaking) are equivalent to FP when scheduling servers.

Determining server priorities. We consider the following heuristics for determining server priorities.

- **Parallelism-decreasing order.** S_i^H has higher priority than S_j^H if $m_i \geq m_j$, with ties being broken consistently.
- **Utilization-decreasing order.** S_i^H has higher priority than S_j^H if $u_i \geq u_j$, with ties being broken consistently.

Schedulability test. The HRT-schedulability of the servers under FP scheduling can be determined by simulating the server schedule over the time interval $[0, H)$. The time complexity for this is polynomial with respect to the task and processor counts. This is because no server is replenished within $(0, H)$, so the servers are scheduled non-preemptively. Thus, scheduling decisions are taken only at time 0 and when a server exhausts its budget. Hence, there are $O(N)$ time instants when scheduling decisions are made. Further, each such decision is of polynomial time complexity.

6.3.2 Least-Laxity-First Scheduling of Servers

Under *least-laxity* (LLF) scheduling, servers with smaller *laxity* have higher priorities. A server's laxity corresponds to the amount of time it can be delayed without violating its deadline. Formally, for a server

S_i^H, letting $C_i^H(t)$ (resp., $D_i^H(t)$) to denote its remaining budget (resp., remaining time to its deadline) at time t, its laxity $L_i^H(t)$ at time t is $L_i^H(t) = D_i^H(t) - C_i^H(t)$. Thus, LLF scheduling also functions like Algorithm 6.1 with $\mathcal{O}$ denoting LLF ordering.

Schedulability test. Similar to FP scheduling, the HRT-schedulability of servers under LLF scheduling can be checked by simulating the server schedule during $[0, H)$. However, unlike FP scheduling, the simulation may take $O(H)$ time, as server priorities may change during runtime.

6.3.3 ILP-Based Scheduling of Servers

Finally, we show that a server schedule can be obtained by solving an integer linear program (ILP), specified as follows.

Variables. For each server S_i^H, we define H variables $x_{i,1}^H, x_{i,2}^H, \ldots, x_{i,H}^H$. $x_{i,t}^H$ is 1 if S_i^H is scheduled during time interval $[t-1, t)$ and 0 otherwise.

Constraint 1. S_i^H is scheduled for $h_i C_i$ time units (its horizontal budget—see the discussion after Lemma 6.4) in $[0, H)$:

$$\forall i :: \sum_{t=1}^{H} x_{i,t}^H = h_i C_i.$$

Constraint 2. At most M processors are occupied at any time:

$$\forall t :: \sum_{i=1}^{n} m_i \cdot x_{i,t}^H \leq M.$$

Note that m_i is a constant.

Translating from a valid assignment of values to the $x_{i,t}^H$ variables to a correct server schedule is straightforward. Note that this method provides an exact sever feasibility test. Unfortunately, it has exponential time complexity.

6.4 Schedulability Under G-EDF

In this section, we consider the preemptive scheduling of gang tasks by G-EDF, which functions as shown in Algorithm 6.2. Under G-EDF, ready jobs with earlier deadlines have higher priorities. We assume that deadline ties are broken arbitrarily but consistently (*e.g.,* by task index). When considering a ready job

Algorithm 6.2 G-EDF job selection policy.

Variables:
Ready(t) : Set of ready jobs at time t
Sched(t) : Set of jobs to be scheduled at time t

```
1: procedure G-EDF
2:     M' ← M
3:     Order jobs in Ready(t) in deadline-increasing order
4:     for each τ_{i,j} ∈ Ready(t) do
5:         if m_i ≤ M' then
6:             Sched(t) ← Sched(t) ∪ {τ_{i,j}}
7:             M' ← M' − m_i
```

$\tau_{i,j}$ under G-EDF, if m_i is larger than the number of remaining available processors, then $\tau_{i,j}$ is skipped (line 5 in Algorithm 6.2). Note that, for scheduling sequential tasks, such scenarios do not arise.

6.4.1 Non-SRT-Optimality Under G-EDF

In this section, we show that G-EDF is not optimal in scheduling SRT gang tasks.

Theorem 6.2. *G-EDF is not SRT-optimal for scheduling gang tasks.*

Proof. We give a gang task system and a release sequence for it where a task has an unbounded response time under G-EDF. Let Γ be a gang task system consisting of seven tasks that are scheduled on six processors. Each task τ_i has a WCET of 7.0 time units and a period of 21.0 time units. Let $m_1 = m_3 = m_5 = 2$ and $m_2 = m_4 = m_6 = m_7 = 3$.

Feasibility. Consider Γ^H from Definition 6.2. Since all task periods are 21, each τ_i^H has the same period, WCET, and degree of parallelism as τ_i. Figure 6.3 shows an HRT-feasible schedule of Γ^H. Thus, by Lemma 6.4, Γ is SRT-feasible.

Unschedulability under G-EDF. Figure 6.4 shows a G-EDF schedule for Γ where each task τ_i releases its first job at time $i-1$ and subsequent jobs periodically, *i.e.,* its j^{th} job is released at time $i-1+(j-1)T_i$. The G-EDF prioritization policy causes at least one idle processor during $[0, 21)$, as a task with $m_i = 3$ is scheduled alongside a task with $m_i = 2$. A similar scenario occurs during the time interval $[22, 43)$. This causes the response time of the second job of each task to be larger than its first job. At times 49 and 50, the third jobs of τ_1 and τ_2, respectively, are scheduled. Thus, the schedule during time $[1, 50)$ starts to repeat at time 50, causing each task's response times to grow unboundedly.

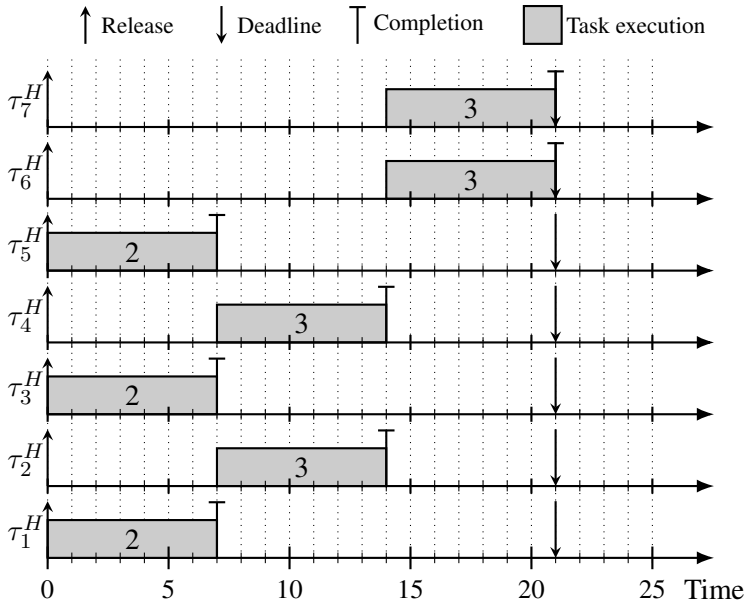


Figure 6.3: An HRT-feasible schedule of Γ^H in Theorem 6.2. The numbers inside execution boxes denote m_i values.

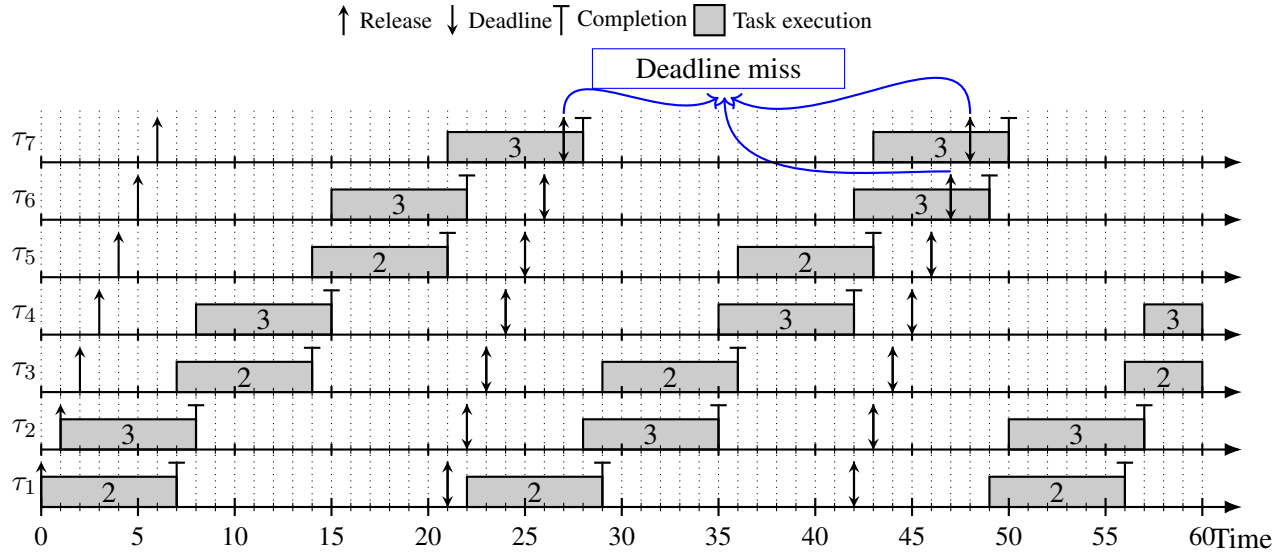


Figure 6.4: G-EDF schedule of Γ in Theorem 6.2. The numbers inside execution boxes denote m_i values.

Since G-EDF cannot ensure bounded response times for an SRT-feasible task system, it is not SRT-optimal for gang tasks by Definition 1.3. □

Note that, according to the proof of Theorem 6.2, G-EDF is not SRT-optimal for gang scheduling even when each task's m_i value is at most three. Since all task deadlines are the same in the system considered in

Theorem 6.2, the same proof also shows the non-SRT-optimality of G-FIFO schedulers for scheduling gang tasks.

6.4.2 A G-EDF Schedulability Test

We now give a schedulability test for G-EDF. Similar to Chapters 3 and 4, we use a lag-based reasoning to derive our test.

Allocation. The cumulative processor capacity (as defined in Definition 3.1) allocated to a job $\tau_{i,j}$, task τ_i, task system Γ, and a set of jobs Ψ, in a schedule $\mathcal{S}$ over an interval $[t, t')$ is denoted by $\mathsf{A}(\tau_{i,j}, t, t', \mathcal{S})$, $\mathsf{A}(\tau_i, t, t', \mathcal{S})$, $\mathsf{A}(\Gamma, t, t', \mathcal{S})$, and $\mathsf{A}(\Psi, t, t', \mathcal{S})$, respectively. Thus,

$$\mathsf{A}(\tau_i, t, t', \mathcal{S}) = \sum_j \mathsf{A}(\tau_{i,j}, t, t', \mathcal{S}),$$

$$\mathsf{A}(\Gamma, t, t', \mathcal{S}) = \sum_{i=1}^{N} \mathsf{A}(\tau_i, t, t', \mathcal{S}),$$

and

$$\mathsf{A}(\Psi, t, t', \mathcal{S}) = \sum_{\tau_{i,j} \in \Psi} \mathsf{A}(\tau_{i,j}, t, t', \mathcal{S}).$$

Ideal schedule. Let $\widehat{\pi}_1, \widehat{\pi}_2, \ldots, \widehat{\pi}_N$ be N processors with speeds $u_1, u_2, \ldots, u_N$, respectively. In an *ideal schedule* $\mathcal{I}$, each task τ_i is partitioned to execute on processor $\widehat{\pi}_i$. Each job starts execution as soon as it is released and completes execution by its deadline in $\mathcal{I}$. For task τ_i (resp., task system Γ), $\mathsf{A}(\tau_i, t, t', \mathcal{I})) \leq u_i(t' - t)$ (resp., $\mathsf{A}(\Gamma, t, t', \mathcal{I}) \leq U_{tot}(t' - t)$). In $\mathcal{I}$, parallelism constraints of gang tasks may not be maintained. Recall from Chapter 4 that precedence constraints were not maintained in the ideal schedule defined for DAG tasks.

lag and LAG. Similar to Chapters 3 and 4, we now define lag and LAG. The lag of job $\tau_{i,j}$ at time t in a schedule $\mathcal{S}$ is

$$\mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) = \mathsf{A}(\tau_{i,j}, 0, t, \mathcal{I}) - \mathsf{A}(\tau_{i,j}, 0, t, \mathcal{S}). \tag{6.2}$$

The lag of a task τ_i at time t in a schedule $\mathcal{S}$ is

$$\mathsf{lag}(\tau_i, t, \mathcal{S}) = \sum_j \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) = \mathsf{A}(\tau_i, 0, t, \mathcal{I}) - \mathsf{A}(\tau_i, 0, t, \mathcal{S}). \tag{6.3}$$

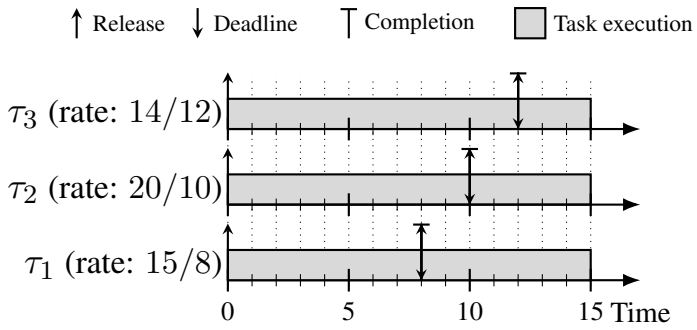


Figure 6.5: An ideal schedule.

Since $\mathsf{lag}(\tau_i, 0, \mathcal{S}) = 0$, for $t' \geq t$ we have

$$\mathsf{lag}(\tau_i, t', \mathcal{S}) = \mathsf{lag}(\tau_i, t, \mathcal{S}) + \mathsf{A}(\tau_i, t, t', \mathcal{I}) - \mathsf{A}(\tau_i, t, t', \mathcal{S}). \tag{6.4}$$

The LAG of a task system Γ in a schedule $\mathcal{S}$ at time t is

$$\mathsf{LAG}(\Gamma, t, \mathcal{S}) = \sum_{\tau_i \in \Gamma} \mathsf{lag}(\tau_i, t, \mathcal{S}) = \mathsf{A}(\Gamma, 0, t, \mathcal{I}) - \mathsf{A}(\Gamma, 0, t, \mathcal{S}). \tag{6.5}$$

Similarly, the LAG of a set of jobs Ψ is

$$\begin{aligned} \mathsf{LAG}(\Psi, t, \mathcal{S}) &= \sum_{\tau_{i,j} \in \Psi} \mathsf{lag}(\tau_{i,j}, t, \mathcal{S}) \\ &= \sum_{\tau_{i,j} \in \Psi} (\mathsf{A}(\tau_{i,j}, 0, t, \mathcal{I}) - \mathsf{A}(\tau_{i,j}, 0, t, \mathcal{S})). \end{aligned} \tag{6.6}$$

Since $\mathsf{LAG}(\Psi, 0, \mathcal{S}) = 0$, for $t' \geq t$ we have

$$\mathsf{LAG}(\Psi, t', \mathcal{S}) = \mathsf{LAG}(\Psi, t, \mathcal{S}) + \mathsf{A}(\Psi, t, t', \mathcal{I}) - \mathsf{A}(\Psi, t, t', \mathcal{S}). \tag{6.7}$$

Example 6.3. Consider three gang tasks $\tau_1 = (8, 5, 3)$, $\tau_2 = (10, 4, 5)$, and $\tau_3 = (12, 7, 2)$ that are scheduled on six processors. Figures 6.5 and 6.6 show an ideal schedule $\mathcal{I}$ and a G-EDF schedule $\mathcal{S}$, respectively, of these tasks. Since τ_2's utilization is $(5 \cdot 4)/10 = 20/10 = 2$, it executes at a rate of 2.0 in $\mathcal{I}$. Task τ_2 receives an allocation of $1 \cdot 5 = 5$ (resp., $6 \cdot 2 = 12$) units during $[0, 6)$ in $\mathcal{S}$ (resp., $\mathcal{I}$). Therefore, $\mathsf{lag}(\tau_2, 6, \mathcal{S}) = 12 - 5 = 7$. ◀

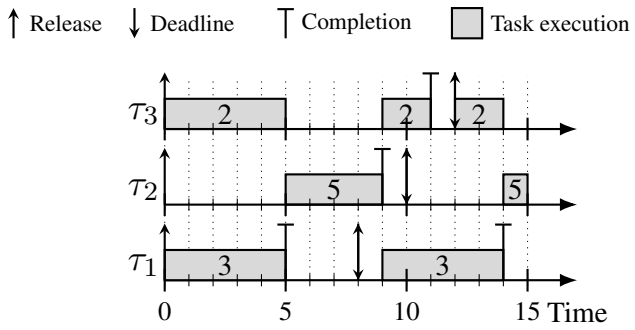


Figure 6.6: A G-EDF schedule. The numbers inside execution boxes denote m_i values.

We now define a notation for the maximum number of processors that can be idle due to parallelism-induced idleness.

Definition 6.3. For a task τ_i, let Δ_i denote the maximum possible number of idle processors at any time instant when τ_i has a ready job that cannot execute. Let $\Delta_{max} = \max_i\{\Delta_i\}$. ◀

Example 6.4. Consider four gang tasks with $m_1 = 3, m_2 = 4, m_3 = 5$, and $m_4 = 6$ to be scheduled on ten processors. If τ_2 has a pending job at time t that cannot execute, then at least seven processors are busy at time t. No combination of other tasks can occupy exactly seven processors. However, if τ_1 and τ_3 execute on $m_1 + m_3 = 8$ processors at time t, then τ_3 cannot execute at time t. Thus, the maximum number of idle processors when τ_3 cannot execute is $\Delta_2 = 10 - 8 = 2$. ◀

Dong *et al.* gave an $O(M^2N)$ time dynamic-programming algorithm to compute the Δ_i value of a task [Dong et al., 2021]. Using Δ_{max}, they established the following sufficient condition for bounded response times of gang tasks under G-EDF, which is the only-known SRT-schedulability test for gang tasks under G-EDF.

Theorem 6.3 ([Dong et al., 2021])**.** *If $U_{tot} \leq M - \Delta_{max}$, then each task in Γ has a bounded response time under G-EDF.*

Consider the task system shown in Figure 6.1. Since $M = 4$, $m_1 = 3$, and $m_2 = 2$, we have $\Delta_1 = 2$ and $\Delta_2 = 1$. The total utilization of the task system is $\frac{3 \cdot 2}{8} + \frac{2 \cdot 6}{8} = \frac{18}{8} = 2.25$, which is larger than $M - \Delta_{max} = 4 - 2 = 2$. Thus, the system is not SRT-schedulable under G-EDF by Theorem 6.3.

We now give an improved sufficient condition for the SRT-schedulability of gang tasks under G-EDF. We first introduce some necessary terms.

Definition 6.4. Let M_p denote the minimum possible number of busy processors whenever at least p tasks in Γ have pending jobs. ◀

Example 6.4 (Continued)**.** Assume that at least three tasks have pending jobs at time t. With four tasks, there are four possibilities of having three pending jobs at time t. Since $m_1 = 3$, $m_2 = 4$, $m_3 = 5$, and $m_4 = 6$, if τ_1, τ_2, and τ_3 have pending jobs at time t, then at least seven processors are busy among all ten processors. Similarly, if τ_1, τ_2, and τ_4 have pending jobs, then at least seven processors are busy. If τ_1, τ_3, and τ_4 have pending jobs, then at least eight processors are busy. Finally, if τ_2, τ_3, and τ_4 have pending jobs, then at least nine processors are busy. Thus, the minimum number of busy processors when at least three tasks have pending jobs is $M_3 = 7$. ◀

Note that $M_N \geq M - \Delta_{max}$ holds. This is because $M - \Delta_{max}$ refers to the minimum possible number of busy processors when a certain task τ_i's job cannot execute despite being ready due to the unavailability of a sufficient number of processors. These scheduled tasks along with τ_i form a subset of all N tasks. Perhaps, if all N tasks have pending jobs, then the minimum possible number of busy processors (M_N) would not decrease.

Before showing how to compute M_p, we first give a sufficient SRT-schedulability condition for G-EDF assuming M_p values are given. This condition is comprised of two sub-conditions involving task utilizations u_i and horizontal utilizations σ_i, which we define next.

Definition 6.5. Let $\mathcal{U}_b = \sum_{b \text{ smallest}} u_i$, *i.e.,* $\mathcal{U}_b$ denote the sum of the b smallest u_i values. ◀

$$\exists\, b \in \mathbb{N}_0 : b < N \wedge U_{tot} \leq (M - \Delta_{max} + \mathcal{U}_b) \wedge U_{tot} \leq M_{N-b} \tag{6.8}$$

$$\forall\, i : \sigma_i \leq 1 \wedge m_i \leq M \tag{6.9}$$

Specifically, we will prove the following theorem.

Theorem 6.4. *If* (6.8) *and* (6.9) *hold, then τ_i's response time is at most $T_i + x + C_i$ where*

$$x \geq \max\{0, \frac{\sum_{(N-b-1)\ largest}\ m_k C_k - C_{min}}{M - \Delta_{max} + \mathcal{U}_{b+1} - U_{tot}}\}. \tag{6.10}$$

Here, $\sum_{(N-b-1)\ largest}\ m_k C_k$ is the sum of the $N - b - 1$ largest values of $m_k C_k$ among all k.

Note that the denominator in (6.10) is positive if (6.8) is met. This is because $\mathcal{U}_{b+1} > \mathcal{U}_b$, implying $M - \Delta_{max} + \mathcal{U}_{b+1} > M - \Delta_{max} + \mathcal{U}_b \geq U_{tot}$. Also, if a task system satisfies the schedulability condition in Theorem 6.3, then it also satisfies (6.8), *i.e.,* Theorem 6.4. This can be shown by considering $b = 0$, for which $U_{tot} \leq M - \Delta_{max} \leq M_N$ holds . The last inequality holds because by Definition 6.3, at least $M - \Delta_{max}$ processors are busy if a task has a pending but unscheduled job.

Example 6.5. Consider seven gang tasks to be scheduled on ten processors by G-EDF. Let $\tau_1 = (10, 1, 9)$ and $\tau_i = (10, 1, 2)$ for all $i > 1$. Thus, $U_{tot} = \frac{9 \cdot 1}{10} + 6 \times \frac{2 \cdot 1}{10} = \frac{21}{10} = 2.1$. Also, we have $\Delta_1 = 8$, as τ_1 cannot execute if one of the remaining tasks is scheduled. Thus, $\Delta_{max} = 8$ and $M - \Delta_{max} = 2$. Since $U_{tot} > M - \Delta_{max} = 2$, the system is deemed SRT-unschedulable by Theorem 6.3.

Now consider the condition in (6.8). For $b = 1, \mathcal{U}_b = 2/10 = 0.2$, so $M - \Delta_{max} + \mathcal{U}_b = 2 + 0.2 = 2.2$ holds, which is larger than U_{tot}. Also, at least nine processors are busy at time t if at least $N - 1$ tasks have pending jobs at time t. Thus, $M_{N-1} = 9 > U_{tot}$, and (6.8) is satisfied. Therefore, the system is SRT-schedulable by Theorem 6.4. ◀

We now prove Theorem 6.4. Our proof strategy is similar to the LAG-based approach pioneered by Devi and Anderson [Devi and Anderson, 2005] for ordinary sporadic tasks, and later adapted for gang tasks by Dong *et al.* [Dong et al., 2021]. The LAG-based analysis in [Dong et al., 2021] relies on determining an upper bound on LAG by considering lag values at the latest time instant t_0 at or before the deadline t_d of a job of interest such that at least $M - \Delta_{max}$[2] processors are busy during $[t_0, t_d)$. However, this may not capture the "opportunistic" execution of lower-priority jobs (*e.g.,* the execution of τ_3 during $[0, 5)$ in Figure 6.6 despite having lower priority than τ_2). By defining t_0 using the number of tasks with pending jobs, instead of busy processors, we can account for such lower-priority job execution. This may result in a larger LAG upper bound at t_0, as more tasks need to be considered, causing larger response-time bounds.

We prove Theorem 6.4 by induction on job priorities. Assume that (6.8) and (6.9) hold for Γ and let b be the smallest non-negative integer for which (6.8) is met. Let $\mathcal{S}$ be a G-EDF schedule of Γ. We consider an arbitrary job $\tau_{i,j}$ and inductively prove that its response time is no more than $T_i + x + C_i$ in $\mathcal{S}$. Thus, we assume the following.

Assumption 6.2. The response time of each job with higher priority than $\tau_{i,j}$ is at most $T_i + x + C_i$ in $\mathcal{S}$.

[2]For sporadic tasks, $\Delta_{max} = 0$. Thus, all processors are busy.

We assume that a job may execute for less than its WCET. Each job $\tau_{k,\ell}$'s execution time is denoted by $C_{k,\ell} \leq C_k$. Let $t_d = d(\tau_{i,j})$ and $t_f = f(\tau_{i,j})$. We assume that $t_f > t_d$ holds, otherwise $\tau_{i,j}$'s response time is at most $D_i = T_i$. We now denote the jobs considered in Assumption 6.2 as follows.

Definition 6.6. Let ψ be the set of jobs that have higher priorities than $\tau_{i,j}$. Let $\Psi = \psi \cup \{\tau_{i,j}\}$. ◀

Under G-EDF scheduling, $\tau_{i,j}$ can only be delayed by the jobs in $\Psi \setminus \{\tau_{i,j}\}$. Note that jobs not in Ψ can be scheduled before $\tau_{i,j}$ due to the lack of enough processors to schedule $\tau_{i,j}$ (line 5 in Algorithm 6.2). However, such jobs will be preempted (if needed) as soon as there are enough processors to schedule $\tau_{i,j}$. The following lemma gives an upper bound on lag values.

Lemma 6.10 ([Dong et al., 2021])**.** *For any task τ_k and a time instant $t \leq t_d$, $\mathsf{lag}(\tau_k, t, \mathcal{S}) \leq m_k(x\lambda_k + C_k)$ holds.*

Definition 6.7. A time instant t is called b-*busy* if at least $N - b$ tasks have pending jobs (hence, at most b tasks have no pending jobs) in Ψ at t, and b-*non-busy* otherwise. A time interval is called b-busy (resp., b-non-busy) if each instant in the interval is b-busy (resp., b-non-busy). ◀

The number of busy processors in a b-busy time instant t depends on the m_i values and the deadlines of the pending jobs at time t. Also, the number of busy processors may vary throughout a b-busy interval because of job releases and completions.[3] However, by Definition 6.4, the number of busy processors is lower bounded by M_{N-b} at any busy instant, as there are at least $N - b$ pending jobs.

Definition 6.8. Let t_0 be the earliest time instant such that $[t_0, t_d)$ is a b-busy interval. Let τ^* be the set of tasks with jobs in Ψ that are pending at time $t_0 - 1$. Note that $\tau^* = \emptyset$, if $t_0 = 0$. ◀

Using Lemma 6.10, we upper bound the LAG of Ψ at t_0.

Lemma 6.11. $\mathsf{LAG}(\Psi, t_0, \mathcal{S}) \leq \sum_{\tau_k \in \tau^*} (m_k (x\lambda_k + C_k))$.

Proof. By (6.6), we have

$$\begin{aligned}\mathsf{LAG}(\Psi, t_0, \mathcal{S}) &= \sum_{\tau_{k,\ell} \in \Psi} \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}) \\ &= \sum_{\tau_k \in \Gamma} \sum_{\tau_{k,\ell} \in \Psi} \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S})\end{aligned}$$

[3]Such variance does not occur for sporadic tasks, as each task's $m_i = 1$.

$$= \sum_{\tau_k \in \tau^*} \sum_{\tau_{k,\ell} \in \Psi} \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}) + \sum_{\tau_k \notin \tau^*} \sum_{\tau_{k,\ell} \in \Psi} \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}). \tag{6.11}$$

We now prove two claims, motivated by (6.11), depending on whether a task is in τ^* or not.

Claim 6.1. *For any $\tau_k \notin \tau^*$, $\sum_{\tau_{k,\ell} \in \Psi} \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}) \leq 0$.*

Proof. Since $\tau_k \notin \tau^*$, τ_k has no pending jobs in Ψ at time $t_0 - 1$. Let $\tau_{k,p} \in \Psi$ be the latest job of τ_k released at or before time $t_0 - 1$ (hence, before time t_0). By the definition of $\mathcal{I}$, for each job $\tau_{k,\ell} \in \Psi$ with $\ell \leq p$, we have $\mathsf{A}(\tau_{k,\ell}, 0, t_0, \mathcal{I}) \leq C_{k,\ell}$. Additionally, since $\tau_{k,p}$ completes execution by time t_0 in $\mathcal{S}$, we have $\mathsf{A}(\tau_{k,\ell}, 0, t_0, \mathcal{S}) = C_{k,\ell}$ for each such job $\tau_{k,\ell}$. Thus, for all $\ell \leq p$, we have $\mathsf{A}(\tau_{k,\ell}, 0, t_0, \mathcal{I}) - \mathsf{A}(\tau_{k,\ell}, 0, t_0, \mathcal{S}) \leq 0$. Therefore, by (6.2) we have

$$\forall \ell \leq p : \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}) \leq 0. \tag{6.12}$$

No job $\tau_{k,\ell} \in \Psi$ with $\ell > p$ can execute before time t_0 in both $\mathcal{I}$ and $\mathcal{S}$. Thus, for $\ell > p$, we have $\mathsf{A}(\tau_{k,\ell}, 0, t_0, \mathcal{I}) = \mathsf{A}(\tau_{k,\ell}, 0, t_0, \mathcal{S}) = 0$. Thus, we have $\forall \ell > p : \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}) = 0$. Together with (6.12), this implies the claim. □

Claim 6.2. *For any $\tau_k \in \tau^*$ and job $\tau_{k,\ell} \notin \Psi$, $\mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}) \geq 0$.*

Proof. Since each task executes sequentially, job $\tau_{k,\ell} \notin \Psi$ cannot execute before all jobs of τ_k in Ψ complete their execution. Since τ_k has a pending job in Ψ at time $t_0 - 1$, $\tau_{k,\ell}$ cannot be scheduled at or before time $t_0 - 1$ in $\mathcal{S}$. Thus, $\mathsf{A}(\tau_{k,\ell}, 0, t_0, \mathcal{S}) = 0$ holds. Since $\mathsf{A}(\tau_{k,\ell}, 0, t_0, \mathcal{I}) \geq 0$ holds, by (6.2), the claim follows. □

By Claim 6.1 and (6.11), we have $\mathsf{LAG}(\Psi, t_0, \mathcal{S}) \leq \sum_{\tau_k \in \tau^*} \sum_{\tau_{k,\ell} \in \Psi} \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S})$, which is at most $\sum_{\tau_k \in \tau^*} \left(\sum_{\tau_{k,\ell} \in \Psi} \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}) + \sum_{\tau_{k,\ell} \notin \Psi} \mathsf{lag}(\tau_{k,\ell}, t_0, \mathcal{S}) \right) = \sum_{\tau_k \in \tau^*} \mathsf{lag}(\tau_k, t_0, \mathcal{S})$, by Claim 6.2. Therefore, by Lemma 6.10, $\mathsf{LAG}(\Psi, t_0, \mathcal{S}) \leq \sum_{\tau_k \in \tau^*} m_k(x\lambda_k + C_k)$. □

Finally, we give an upper bound on LAG of Ψ at time t_d.

Lemma 6.12. $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) \leq \sum_{\tau_k \in \tau^*} (m_k (x\lambda_k + C_k))$.

Proof. Since $[t_0, t_d)$ is a b-busy interval, by Definitions 6.4 and 6.7, at least M_{N-b} processors are busy executing jobs in Ψ during $[t_0, t_d)$ in $\mathcal{S}$. Thus, $\mathsf{A}(\Psi, t_0, t_d, \mathcal{S}) \geq M_{N-b}(t_d - t_0)$ holds. By (6.7), we have

$$\begin{aligned}
\mathsf{LAG}(\Psi, t_d, \mathcal{S}) &= \mathsf{LAG}(\Psi, t_0, \mathcal{S}) + \mathsf{A}(\Psi, t_0, t_d, \mathcal{I}) - \mathsf{A}(\Psi, t_0, t_d, \mathcal{S}) \\
&\leq \{\text{Since } \mathsf{A}(\Psi, t_0, t_d, \mathcal{I}) \leq U_{tot}(t_d - t_0) \text{ and } \mathsf{A}(\Psi, t_0, t_d, \mathcal{S}) \geq M_{N-b}(t_d - t_0)\} \\
&\quad \mathsf{LAG}(\Psi, t_0, \mathcal{S}) + U_{tot}(t_d - t_0) - M_{N-b}(t_d - t_0) \\
&\leq \{\text{Since } U_{tot} \leq M_{N-b} \text{ by (6.8)}\} \\
&\quad \mathsf{LAG}(\Psi, t_0, \mathcal{S}) \\
&\leq \{\text{By Lemma 6.11}\} \\
&\quad \sum_{\tau_k \in \tau^*} \left(m_k \left(x\lambda_k + C_k\right)\right).
\end{aligned}$$

□

Let W be the total remaining workload of Ψ at time t_d in $\mathcal{S}$. Using Lemma 6.12, we upper bound W in the lemma below.

Lemma 6.13. $W \leq \sum_{\tau_k \in \tau^*} \left(m_k \left(x\lambda_k + C_i\right)\right)$.

Proof. By the definition of $\mathcal{I}$, all jobs in Ψ finish execution by time t_d in $\mathcal{I}$. The completed workload of the jobs in Ψ at time t_d is $\mathsf{A}(\Psi, 0, t_d, \mathcal{S})$. Thus, the remaining workload of Ψ at time t_d in $\mathcal{S}$ is $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) \leq \sum_{\tau_k \in \tau^*} \left(m_k \left(x\lambda_k + C_k\right)\right)$. □

The following lemma gives a lower bound on W if $\tau_{i,j}$'s response time exceeds $T_i + x + C_i$. The lemma can be proven by considering an interval $[t_d, t_d + t_y)$ during which at least $M - \Delta_{max}$ processors execute jobs in Ψ.

Lemma 6.14 ([Dong et al., 2021])**.** *If $W \leq (M - \Delta_{max})x + C_i$ holds, then $\tau_{i,j}$'s response time is at most $T_i + x + C_i$.*

The next lemma shows that Theorem 6.4 holds.

Lemma 6.15. *$\tau_{i,j}$'s response time is at most $T_i + x + C_i$.*

Proof. Assume that $\tau_{i,j}$'s response time is more than $T_i + x + C_i$. Then, by Lemma 6.14, $W > (M - \Delta_{max})x + C_i$ holds. By Lemma 6.13, we have

$$(M - \Delta_{max})x + C_i < \sum_{\tau_k \in \tau^*} m_k \left(x\lambda_k + C_k\right),$$

which implies

$$\begin{aligned}
x &< \frac{\sum_{\tau_k \in \tau^*} m_k C_k - C_i}{M - \Delta_{max} - \sum_{\tau_k \in \tau^*} m_k \lambda_k} \\
&= \frac{\sum_{\tau_k \in \tau^*} m_k C_k - C_i}{M - \Delta_{max} - \sum_{\tau_k \in \tau^*} u_k} \\
&\leq \{\text{Since } |\tau^*| \leq N - b - 1 \text{ and } C_i \geq C_{min}\} \\
&\quad \frac{\sum_{(N-b-1) \text{ largest}} m_k C_k - C_{min}}{M - \Delta_{max} - \sum_{(N-b-1) \text{ largest}} u_k} \\
&= \frac{\sum_{(N-b-1) \text{ largest}} m_k C_k - C_{min}}{M - \Delta_{max} - U + \sum_{(b+1) \text{ smallest}} u_k} \\
&= \{\text{By Definition 6.5}\} \\
&\quad \frac{\sum_{(N-b-1) \text{ largest}} m_k C_k - C_{min}}{M - \Delta_{max} + \mathcal{U}_{b+1} - U},
\end{aligned}$$

which contradicts (6.10). □

Discussion. The response-time bound given in Theorem 6.4 is smaller for large b values. Thus, to compute a small response-time bound, the largest b value that satisfies (6.8) should be picked.

Computing M_p. We now show how to compute the value of M_p. We begin by giving the following property.

Property 6.1. *Let M_p^e denote the minimum possible number of busy processors whenever* exactly p *tasks in* Γ *have pending jobs. Then, for any $p < N$, $M_p^e \leq M_{p+1}^e$.*

By Property 6.1, we have $M_p = M_p^e$. Thus, we compute M_p by determining M_p^e. To compute M_p^e, we first index tasks in the non-decreasing order by m_i, *i.e.,* $m_i \leq m_{i+1}$.

Property 6.2. *If M' processors are busy at time t, then, for any unscheduled task τ_i with pending jobs, $M' + m_i > M$ holds.*

Algorithm 6.3 Finding M_p.

Variables:
$F[i, \ell, m]$ is initially NULL
$B[i, \ell, m]$ precomputed true/false values
m_i values in non-decreasing order

```
1:  procedure FIND_M_p(i, ℓ, m)
2:      if F[i, ℓ, m] ≠ NULL then
3:          return F[i, ℓ, m]
4:      if ℓ < 0 ∨ m < 0 then
5:          return ∞
6:      if i = N then
7:          F[i, ℓ, m] ← ∞
8:          if ℓ = 0 and m_i ≤ m then
9:              F[i, ℓ, m] ← m_i
10:         if (ℓ = 0 and m_i > m) or ℓ = 1 then
11:             F[i, ℓ, m] ← 0
12:         return F[i, ℓ, m]
13:     x_1 ← FIND_M_p(i + 1, ℓ − 1, m)
14:     x_2 ← FIND_M_p(i + 1, ℓ, m − m_i) + m_i
15:     for each x ∈ {m − m_i + 1, ··· , m} do
16:         if B[i + 1, ℓ, x] = true then
17:             x_3 ← x
18:             break
19:     F[i, ℓ, m] ← min(x_1, x_2, x_3)
20:     return F[i, ℓ, m]
```

We give a dynamic-programming algorithm to compute M_p^e that satisfies Property 6.2 as shown in Algorithm 6.3. Algorithm 6.3 uses a precomputed array B, which we compute via dynamic programming. We first describe how B is computed.

Let $B[i, \ell, m]$ be *true* if there exists a subset $\Gamma_{i,\ell}$ of $(N - i + 1) - \ell$ tasks with pending jobs (*i.e.,* exactly ℓ tasks with no pending jobs) in $\{\tau_i, \tau_{i+1}, \cdots, \tau_N\}$ such that $(\exists\, \Gamma_{i,\ell}^e \subseteq \Gamma_{i,\ell} : \sum_{\tau_k \in \Gamma_{i,\ell}^e} m_k = m)$ holds, and *false* otherwise. Informally, if $B[i, \ell, m]$ is true and if there are exactly ℓ tasks in $\{\tau_i, \tau_{i+1}, \cdots, \tau_N\}$ that have no pending jobs at time t, then there is a way to select jobs from the remaining $(N - i + 1) - \ell$ tasks to occupy exactly m processors.

We can compute the array B in $O(N^2 M)$ time via dynamic programming using the following recurrence.

$$B[i,\ell,m] = \begin{cases} true & \text{if } i = N \ \wedge ((m = m_N \wedge \ell = 0) \vee (m = 0 \wedge \ell \leq 1)) \\ false & \text{if } \ell < 0 \vee (i = N \ \wedge ((m = m_n \wedge \ell \neq 0) \vee (m = 0 \wedge \ell > 1) \\ & \qquad \vee (m \notin \{0, m_n\}))) \\ B[i+1,\ell,m-m_i] & \\ \ \vee B[i+1,\ell,m] & \text{otherwise} \\ \ \vee B[i+1,\ell-1,m] & \end{cases} \tag{6.13}$$

The first two cases in (6.13) cover the base cases. For $i = N$, $B[N, \ell, m]$ is only true if $\ell = 0$ (τ_N has a pending job) and $m = m_N$, or $\ell \leq 1$ and $m = 0$. For $i < N$, $B[i, \ell, m]$ is computed via the third case in (6.13). $B[i+1, \ell, m - m_i]$ (resp., $B[i+1, \ell, m]$) holds when τ_i has a pending job that executes on m_i processors (resp., does not execute), ℓ tasks in $\{\tau_{i+1}, \cdots, \tau_N\}$ have no pending jobs, and those tasks with pending jobs occupy $m - m_i$ (resp., m) processors. $B[i+1, \ell-1, m]$ holds when τ_i has no pending jobs, $\ell - 1$ tasks in $\{\tau_{i+1}, \cdots, \tau_N\}$ have no pending jobs, and the tasks with pending jobs occupy m processors.

Using the array B, procedure FIND_$M_p(i, \ell, m)$ in Algorithm 6.3 determines the minimum number of busy processors when $\{\tau_i, \tau_{i+1}, \cdots, \tau_N\}$ are scheduled on m processors and ℓ tasks among them have no pending jobs. To compute M_p, we invoke FIND_$M_p(1, N - p, M)$. We now describe the Algorithm 6.3. Lines 2–3 check whether the subproblem is already computed and lines 4–12 cover the base cases. Line 13 makes a recursive call to determine the minimum number of busy processors when τ_i has no pending jobs (thus, $\ell - 1$ tasks among $\{\tau_{i+1}, \ldots, \tau_N\}$ have no pending jobs). Line 14 considers the case where τ_i has a pending and scheduled job (thus, ℓ tasks among $\{\tau_{i+1}, \ldots, \tau_N\}$ have no pending jobs). Lines 15–18 consider the case where τ_i has pending but unscheduled jobs. In this case, at least $m - m_i + 1$ processors must be busy. Thus, for each $x \in \{m - m_i + 1, \cdots, m\}$, we consult the array B to determine the lowest possible x value for which $B[i+1, \ell, x]$ is true. Note that if $B[i+1, \ell, x]$ is true, then any unscheduled task τ_k with $k > i$ satisfies Property 6.2 because $m_i \leq m_k$. Finally, the minimum among the three cases is returned.

Since $i \leq N$, $\ell \leq N$, and $m \leq M$ holds, FIND_M_p is called at most $O(N^2 M)$ times. In each call, lines 15–18 take $O(M)$ time with the precomputed array B. Thus, the total time to compute M_p is $O(N^2 M^2)$. Since M_p can be computed in polynomial time, (6.8) can also be checked in polynomial time.

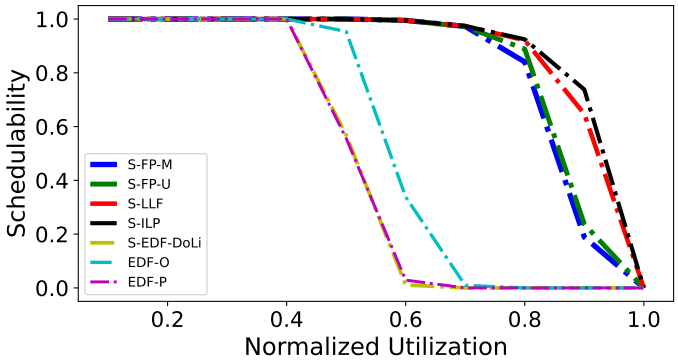
6.5 Experimental Evaluation

In this section, we provide the results of a schedulability study we conducted to evaluate our proposed approaches.

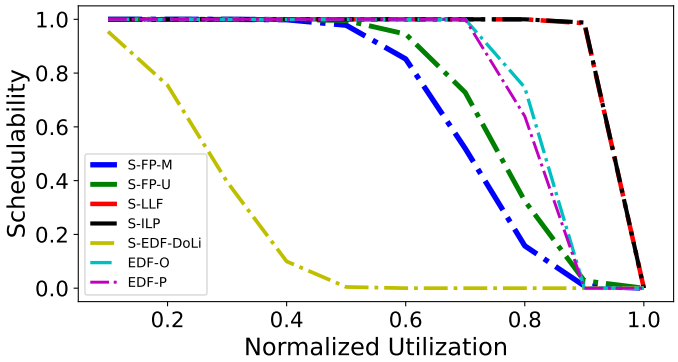
Our task-system generation method is similar to that used in prior gang-task-related schedulability studies [Dong and Liu, 2019, 2022; Dong et al., 2021]. We generated task systems randomly for platforms with $M = 16$ or $M = 32$ processors. Motivated by automotive use cases, we chose task periods from $\{2, 5, 10, 20, 50, 100, 200, 1000\}$ms [Kramer et al., 2015]. We considered *light*, *medium*, or *heavy* horizontal task utilizations, which are uniformly distributed in $[0.01, 0.1]$, $[0.1, 0.3]$, and $[0.3, 1]$, respectively. We set each task's WCET C_i to $T_i \cdot \sigma_i$ rounded to the next microsecond. We considered *small*, *moderate*, or *heavy* degrees of parallelism, for which m_i values were uniformly distributed in $[1, \frac{M}{4}]$, $[\frac{M}{4}, \frac{5M}{8}]$, and $[\frac{5M}{8}, \frac{7M}{8}]$, respectively. We varied the *normalized utilization*, *i.e.,* U_{tot}/M, from 0.1 to 1.0 with a step size of 0.1. For each combination of M, horizontal task utilization, degree of parallelism, and normalized utilization, we generated 1,000 task systems. We generated each such task system by creating tasks until the system's normalized utilization exceeded the desired value, and then reducing the last task's utilization so that the normalized utilization equaled the desired value. We call each combination of M, horizontal task utilization, and degree of parallelism a *scenario*.

We assessed the SRT-schedulability of each task system under both G-EDF and the server-based scheduling policies given in Section 6.2. For scheduling servers, we considered FP scheduling with parallelism-decreasing priorities (S-FP-M), FP scheduling with utilization-decreasing priorities (S-FP-U), LLF scheduling (S-LLF), and ILP-based scheduling (S-ILP). To assess the efficacy of the schedulability tests of servers given in Section 6.2, we also determined the schedulability of servers under G-EDF using methods in [Dong and Liu, 2019] (EDF-DOLI). For G-EDF scheduling of gang tasks, we determined schedulability by the **p**rior method (denoted EDF-P) from Dong *et al.*, *i.e.,* Theorem 6.3, and by **o**ur method (denoted EDF-O), *i.e.,* Theorem 6.4. For each scenario, we computed *acceptance ratios*, which give the percentage of task systems that were schedulable under each approach. We present a representative selection of our results in Figures 6.7 and 6.8.

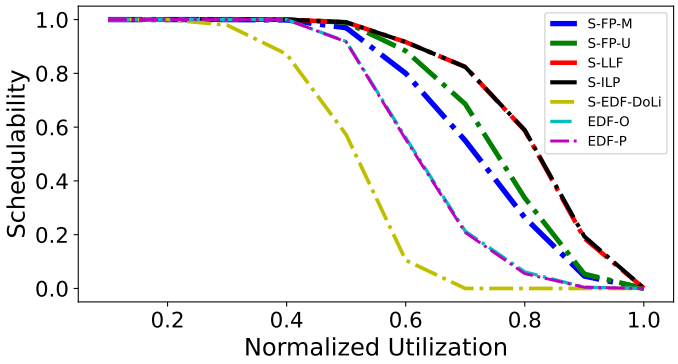
Observation 6.1. *In all scenarios,* S-LLF *had a higher acceptance ratio than* S-FP-M*,* S-FP-U*,* EDF-O*, and* EDF-P*. For most scenarios,* S-FP-M *and* S-FP-U *had higher acceptance ratios than* EDF-P*. The*



(a) Acceptance ratio for medium horizontal utilizations and moderate degree of parallelism.



(b) Acceptance ratio for heavy horizontal utilizations and small degree of parallelism.



(c) Acceptance ratio for heavy horizontal utilizations and moderate degree of parallelism.

Figure 6.7: Schedulability results.

average improvement in S-LLF, S-FP-M, S-FP-U, *and* EDF-O *over* EDF-P *was 37.65%, 26.37%, 28.79%, and 8.32%, respectively.*

This can be seen in Figure 6.7(a)–(c). Being a JLDP scheduling algorithm, LLF can schedule more task systems than the other approaches. In most scenarios, server-based FP scheduling outperformed G-EDF, while utilization-decreasing priority ordering outperformed parallelism-decreasing priority ordering. As expected, more task systems were schedulable by Theorem 6.4 than by Theorem 6.3.

Observation 6.2. *For scenarios with a small degree of parallelism,* EDF-O *and* EDF-P *had higher acceptance ratios than* S-FP-M *and* S-FP-U.

This can be seen in Figure 6.7(b). When the m_i values are small, Δ_{max} (Definition 6.3) is also small. This causes more task systems to be schedulable under G-EDF by Theorem 6.3.

Observation 6.3. *The average improvement in* S-ILP *over* S-LLF*,* S-FP-M*,* S-FP-U*,* EDF-O*, and* EDF-P *was 0.16%, 9.10%, 7.05%, 27.29%, and 37.88%, respectively. In all scenarios,* EDF-DoLi *had smaller acceptance ratios than* S-ILP*,* S-LLF*,* S-FP-M*, and* S-FP-U.

Figure 6.7 shows this. Server-based LLF scheduling scheduled most task systems that were schedulable under ILP-based scheduling. In contrast, many SRT-feasible task systems were deemed unschedulable by the other considered approaches. EDF-DoLi was deemed more pessimistic than S-ILP, S-LLF, S-FP-M, and S-FP-U, as it is applicable to HRT-scheduling of sporadic gang tasks.

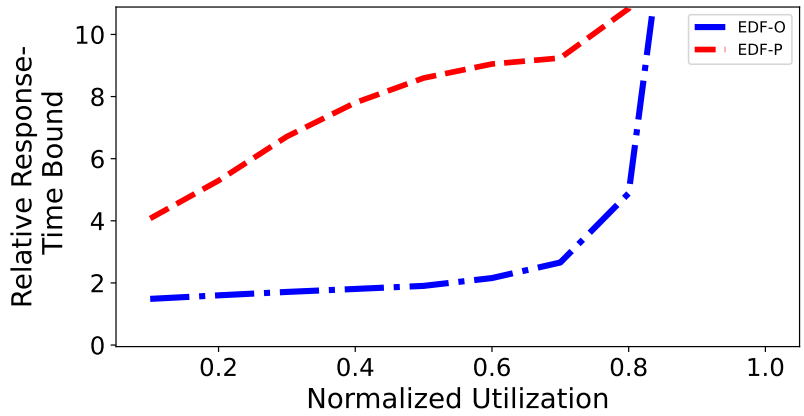
To compare the response-time bounds derived under G-EDF (Theorem 6.4) with those from [Dong et al., 2021], we computed *relative response-time bounds* for all task systems that are SRT-schedulable according to the corresponding G-EDF schedulability tests. A task's relative response time is computed by dividing its response time by the maximum period, *i.e.,* T_{max}. When computing relative response-time bounds using Theorem 6.4, we selected the largest value of b for which (6.8) was satisfied.

Observation 6.4. *On average, relative response-time bounds in* EDF-O *were 47.53% smaller than* EDF-P.

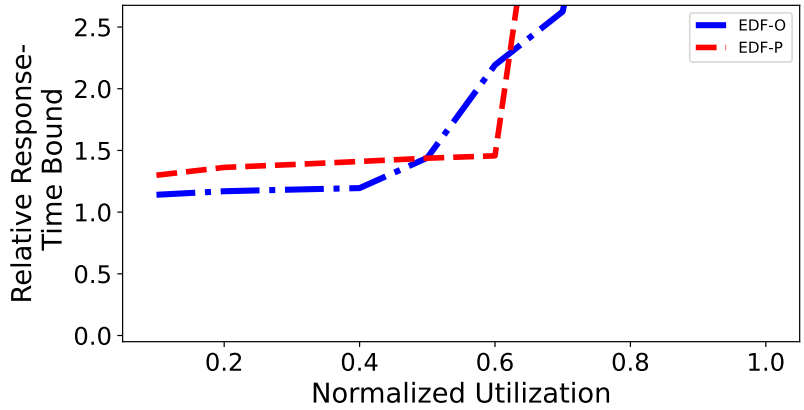
Figure 6.8 shows this. This improvement was due to frequently observed large b values that contributed to a less pessimistic accounting of *carry-on* workloads.

6.6 Chapter Summary

In this chapter, we have considered the SRT-feasibility problem for systems of gang tasks. We have presented a necessary and a sufficient condition for the SRT-feasibility of such systems. Based on these



(a) Relative response-time bounds under G-EDF for $M = 32$, heavy horizontal utilizations, and small degree of parallelism.



(b) Relative response-time bounds under G-EDF for $M = 16$, medium horizontal utilizations, and moderate degree of parallelism.

Figure 6.8: Response-time bound results.

conditions, we have shown that the SRT-feasibility problem for gang task systems is NP-hard. We have also provided server-based scheduling policies for gang tasks and corresponding SRT-schedulability tests for gang tasks based on exact HRT-schedulability tests for the servers. Finally, we have shown that G-EDF is not SRT-optimal for gang tasks and provided an SRT-schedulability test for gang tasks under G-EDF. We have provided experimental evaluations that demonstrate the benefits of our approaches.

CHAPTER 7: SCHEDULING GANG TASKS WITH PRECEDENCE CONSTRAINTS[1]

In this chapter, we consider scheduling processing graphs of gang tasks on heterogeneous platforms consisting of different types of *compute elements* (CEs) such as CPUs, GPUs, FPGAs, *etc.* In such a task model, gang tasks have precedence constraints among them and each gang task is assigned to a particular CE on which it executes. This task model generalizes various systems. We describe two such examples below.

Scheduling on multicore+GPU platforms. Processing graphs scheduled on multicore platforms augmented with GPUs represent a special case of our task model. In such systems, graph nodes that execute on GPUs are modeled as gang tasks, while nodes that execute on CPUs are sequential tasks (a special case of gang tasks). It is worth noting that, for such applications, the corresponding processing graphs can be constructed at different granularities [Yang et al., 2018]. Below, we discuss a few such approaches.

In the first approach, all GPU accesses are hidden inside a CPU node by treating them as suspension times on the CPU. This essentially converts the schedulability problem into a CPU-only scheduling problem. The key benefit of this approach is that it allows the use of existing CPU-only schedulability tests for such systems. Figure 7.1(a) illustrates a DAG that contains a GPU-accessing node. The shaded region indicates the suspension time on the CPU due to the GPU access.

In the second approach, the graph may explicitly contain GPU nodes, but at a coarser level of granularity. Such coarse-grained graphs may include GPU *kernels*—pieces of code that execute on the GPU in parallel across thousands or even millions of threads—as GPU nodes. In the case of NVIDIA GPUs, these GPU nodes execute on one or more *streaming multiprocessors* (SMs), which are mini multicore processors inside the GPU. For analyzing such coarse-grained graphs, the SMs can be treated as black boxes; *e.g.,* details of how threads are scheduled within SMs need not be considered in the analysis. Figure 7.1(b) illustrates a coarse-grained GPU-accessing DAG. The GPU-accessing node is decomposed into a GPU node, representing the GPU kernel, and CPU nodes responsible for launching the kernel and reading the results from the GPU.

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

Ahmed, S., Massey, D., and Anderson, J. (2025), Scheduling Processing Graphs of Gang Tasks on Heterogeneous Platforms, *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 362–374.

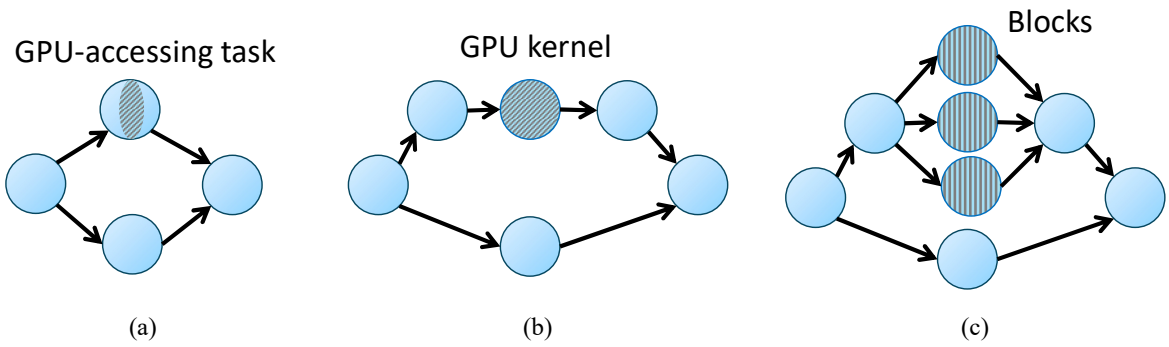


Figure 7.1: Illustration of GPU-accessing DAGs when (a) GPU accesses are treated as CPU suspension time, explicitly at a (b) coarser granularity, and (c) finer granularity.

Finally, fine-grained graphs can be constructed by decomposing GPU kernels into *blocks*. Blocks are groups of kernel threads, with their number and size specified in the user code. When fine-grained graphs are analyzed, the schedulability analysis requires knowledge of block WCETs and how blocks execute within SMs. Note that block WCET estimates can be obtained via measurements using the `globaltimer` performance-counter register [Yang et al., 2018]. Interested readers are referred to [Yang et al., 2018] for more details. Figure 7.1(c) shows a fine-grained GPU-accessing DAG, in which the GPU kernel is decomposed into multiple thread blocks that can execute concurrently on the GPU.

Time partitioning in component-based systems. Large systems are often decomposed into components that interact with one another through pre-defined interfaces. Software certification processes for safety-critical applications typically require these components to be partitioned in time and space (*e.g.,* ARINC 653 in avionics [Prisaznuk, 2008]). Hierarchical scheduling techniques can enable time partitioning among different software components of a component-based system. Under these techniques, the top-level scheduler allocates a gang-like set of processors to each component for certain time intervals. Since dataflow dependencies may be present between components due to their interaction through interfaces, designing the top-level scheduler reduces to the problem of scheduling processing graphs of gang tasks.

Scheduling. In this chapter, we consider *work-conserving* and *semi-work-conserving* scheduling of processing graphs of gang tasks. GEL schedulers form a subset of work-conserving schedulers; therefore, our results also apply to GEL schedulers. We introduce semi-work-conserving schedulers because scheduling on an NVIDIA GPU is semi-work-conserving when all GPU work is submitted from the same address space [Amert

et al., 2017]. We illustrate semi-work-conserving schedulers and their connection to GPU scheduling in Section 7.2.

Organization. After covering needed background (Section 7.1), we discuss the considered scheduling algorithms in detail (Section 7.2), provide techniques to account for *parallelism-induced* idleness for gang tasks that form DAGs (Section 7.3), give our response-time bound for a DAG (Section 7.4), present techniques to support multiple DAGs (Section 7.5), present our experiments (Section 7.6), and conclude (Section 7.7).

7.1 System Model

We consider a task system Γ consisting of N DAG tasks $\{G^1, G^2, \ldots, G^N\}$. For ease of notation, we use DAG indices only when relevant. Each DAG task G releases a potentially infinite sequence of DAG jobs $G_1, G_2, \ldots$. The release and completion time of G_j are denoted by $r(G_j)$ and $f(G_j)$, respectively. DAG jobs of G are released sporadically with period T. We assume that each DAG G has a constrained relative deadline $D \leq T$, *i.e.,* DAG job G_j must finish execution by time $r(G_j) + D$.

Each DAG G is represented as a tuple (V, E), where V and E are sets of nodes and directed edges, respectively. The set V consists of n rigid *gang* tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. We assume that tasks are indexed according to a topological order of G. Each gang task τ_i has a (WCET) C_i and a *degree of parallelism* m_i that denotes the number of simultaneously available processors required to execute any job (instance) of τ_i. Thus, the *worst-case execution requirement* (WCER) of each job of τ_i is $m_i \times C_i$. A directed edge from τ_i to τ_k represents a precedence constraint between the *predecessor* task τ_i and the *successor* task τ_k. The set of predecessors (resp., successors) of τ_i is denoted by $pred(\tau_i)$ (resp., $succ(\tau_i)$). We assume that each DAG G has a unique *source* task τ_1 with no incoming edges and a unique *sink* task τ_n with no outgoing edges. The *utilization* of τ_i is $u_i = (C_i \times m_i)/T$, which can exceed 1.0. The utilization of DAG task G is $U = \sum_{i=1}^{n} u_i$. The *total utilization* of Γ is $U_{tot} = \sum_{G \in \Gamma} U$.

DAGs in Γ are scheduled on μ *compute elements* (CEs). A CE might be a CPU or some specialized hardware accelerator. The k^{th} CE consists of $\mathcal{M}_k$ identical processors. Each task τ_i of a DAG G has a parameter $\gamma_i \in \{1, 2, \ldots, \mu\}$ that represents the CE on which τ_i executes.

Example 7.1. Figure 7.2 shows a DAG of ten tasks on two CEs. Tasks τ_2, τ_3, τ_4, and τ_9 are assigned to one CE, while the remaining tasks execute on the other CE. τ_7's degree of parallelism is $m_7 = 3$ and its WCET is $C_7 = 4$. ◀

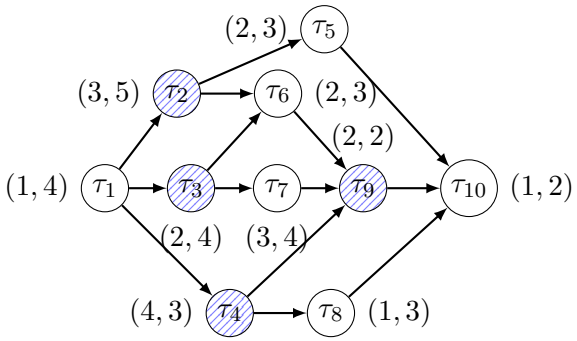


Figure 7.2: A DAG G. Solid and hatched circles represent tasks allocated to two different CEs. Tuples circles represent (m_i, C_i).

Each DAG job G_j is composed of the j^{th} job $\tau_{i,j}$ of each task τ_i. The release time and finish time of job $\tau_{i,j}$ are denoted by $r(\tau_{i,j})$ and $f(\tau_{i,j})$, respectively. The j^{th} job $\tau_{1,j}$ of the source task τ_1 is released when G_j is released, *i.e.,* $r(G_j) = r(\tau_{1,j})$. The j^{th} job of each non-source task is released once the j^{th} job of each of its predecessors finishes, *i.e.,* $r(\tau_{i,j}) = \max_{\tau_k \in pred(\tau_i)}\{f(\tau_{k,j})\}$. Job $\tau_{i,j}$ is *ready* to execute during $[r(\tau_{i,j}), f(\tau_{i,j}))$. DAG job G_j completes when $\tau_{n,j}$ completes. The *response time* of G_j is $R(G_j) = f(\tau_{n,j}) - r(\tau_{1,j})$. G's response time is $R(G) = \sup_j\{R(G_j)\}$.

A *path* $\lambda = \{v_1, v_2, \ldots, v_k\}$ is an ordered set of tasks of G (*i.e.,* $v_i \in V$ for each $1 \leq i \leq k$) such that $v_i \in pred(v_{i+1})$ holds. (We use the symbol v to simplify indexing nodes of λ.) When job indices are irrelevant, we also use λ to denote the ordered set of the j^{th} jobs of tasks in $\{v_1, v_2, \ldots, v_k\}$. A path is a *complete path* if it contains the source and sink nodes. We define the *length* of a path as follows:

$$len(\lambda) = \sum_{\tau_i \in \lambda} C_i. \tag{7.1}$$

If a path exists from τ_i to τ_k, then τ_i (resp., τ_k) is called an *ancestor* (resp., *descendant*) of τ_k (resp., τ_i). The j^{th} job of an ancestor (resp., descendant) task of τ_i is an ancestor (resp., descendant) job of $\tau_{i,j}$. The set of ancestors (resp., descendants) of τ_i is denoted as $anc(\tau_i)$ (resp., $desc(\tau_i)$). The set of ancestors (resp., descendants) of $\tau_{i,j}$ is denoted as $anc(\tau_{i,j})$ (resp., $desc(\tau_{i,j})$). {We use $dep(\tau_i)$ (resp., $dep(\tau_{i,j})$) to denote $anc(\tau_i) \cup desc(\tau_i)$ (resp., $anc(\tau_{i,j}) \cup desc(\tau_{i,j})$). For any subset $V' \subseteq V$ of tasks, we define its *volume* as follows:

$$vol(V') = \sum_{\tau_i \in V'} m_i C_i. \tag{7.2}$$

Example 7.1 (Continued)**.** In Figure 7.2, task τ_6 has two predecessors τ_2 and τ_3. Tasks $\{\tau_1, \tau_3, \tau_7, \tau_9, \tau_{10}\}$ form a complete path with length 16. Task τ_6's ancestors (resp., descendants) are $anc(\tau_6) = \{\tau_1, \tau_2, \tau_3\}$ (resp., $desc(\tau_6) = \{\tau_9, \tau_{10}\}$). Finally, $vol(anc(\tau_6)) = 1 \cdot 4 + 3 \cdot 5 + 2 \cdot 4 = 27$. ◀

We summarize all introduced notation in Table 7.1.

7.2 Scheduling

In this section, we describe the scheduling policies under which we give the response-time bounds in Section 7.4.

7.2.1 Federated Scheduling

To schedule multiple DAGs, we use the *federate* scheduling techniques discussed in Section 2.2. Under federated scheduling each DAG G is allocated a dedicated set of processors from each CE. Let M_p denote the number of processors of the p^{th} CE assigned to G. Thus, all jobs of task τ_i with $\gamma_i = p$ are scheduled on the M_p processors of the p^{th} CE assigned to G. We require $M_p \geq \max_{\tau_i:\gamma_i=p}\{m_i\}$, otherwise, a task will never be scheduled. M_p can be zero if no task of G requires the p^{th} CE.

Heavy vs. light DAGs. For scheduling DAGs of sequential tasks (*i.e.,* tasks with $m_i = 1$) on a single CE, federated scheduling approaches differentiate between *heavy* (DAGs with $\sum_{\tau_i \in V} C_i > D$) and *light* (DAGs with $\sum_{\tau_i \in V} C_i \leq D$) DAG tasks [Li et al., 2014]. Each heavy DAG requires parallel execution of its nodes, so it is allocated enough processors to meet its deadline. In contrast, all light DAGs share a set of processors, where they are scheduled as sequential tasks.

When DAG tasks are scheduled on multiple CEs, federated scheduling techniques become more nuanced. This is because whether a DAG should be treated as heavy or light should be determined on a per-CE basis. For example, a DAG may have many nodes executing on a CE, while only a few on another. In such a case, the DAG may be treated heavy on one and light on another [Lin et al., 2023]. Furthermore, if nodes of a DAG assigned to a CE are scheduled sequentially on the CE, jobs of those nodes may exhibit self-suspending behavior with respect to the CE, as illustrated in the following example.

Example 7.2. Assume that three nodes, τ_1, τ_3, and τ_6, are assigned to a CE. Figure 7.3 shows a schedule of these nodes when they execute sequentially. All three nodes have different m_i values. The duration between

Table 7.1: Notation summary for Chapter 7.

Symbol	**Meaning**
Γ	Task system
N	Number of DAG tasks
μ	Number of CEs
G	A DAG
V	Nodes of G
E	Edges of G
T	Period of G
D	Rel. deadline of G
τ_i	i^{th} task of G
C_i	WCET of τ_i
γ_i	Assigned CE of τ_i
m_i	τ_i's degree of parallelism
$pred(\cdot)$	Set of predecessors
$succ(\cdot)$	Set of successors
$anc(\cdot)$	Set of ancestors
$desc(\cdot)$	Set of descendants
λ	path of G
V'_p	Tasks of V' on the p^{th} CE
$len(\lambda)$	$\sum_{\tau_i \in \lambda} C_i$
$vol(V')$	$\sum_{\tau_i \in V'} m_i C_i$
$R(\cdot)$	Response time
U_{tot}	Utilization of Γ
$\mathcal{M}_p$	Processor count on p^{th} CE
G_j	j^{th} DAG job of G
$\tau_{i,j}$	j^{th} job of τ_i^v
$r(\tau_{i,j})$	Release time of job or server job
$f(\tau_{i,j})$	Completion time of $\tau_{i,j}^v$

the execution of τ_1 and τ_3, when other nodes execute on different CEs, can be regarded as self-suspension times on τ_1's CE. ◀

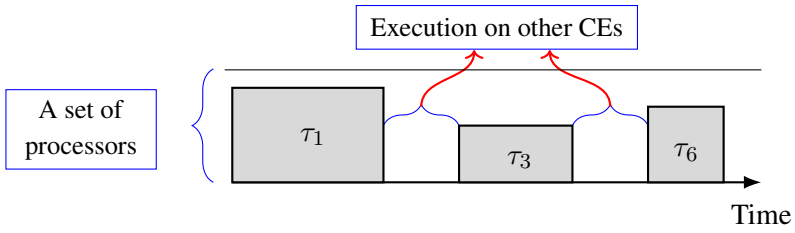


Figure 7.3: Scheduling DAG nodes sequentially on a CE.

Thus, if some DAGs are scheduled sequentially on a shared set of processors, deriving their response-time bounds may require analyzing self-suspending *bundled* gang tasks. A bundled gang task consists of a chain of multiple rigid gang subtasks (a special case of our task model), where the m_i values of two subtasks may differ [Wasly and Pellizzoni, 2019]. Thus, the execution in Figure 7.3 can be viewed as bundled tasks with self-suspension between two consecutive subtasks. We defer providing such an analysis to future work, restricting our focus on the case where each DAG receives a dedicated number of processors on each CE.

7.2.2 Scheduling DAGs on Allocated Processors

We consider *work-conserving* and *semi-work-conserving* approaches for scheduling each DAG on its allocated processors. Among these two, work-conserving scheduling has been widely studied for sequential and traditional DAG tasks (DAG tasks with sequential nodes). Work-conserving scheduling for gang tasks differs from such scheduling of sporadic or DAG tasks, as illustrated below.

Work-conserving scheduling. For gang tasks, work-conserving schedulers do not allow a set of processors on a CE to remain idle if a ready gang job can use them. Specifically, under work-conserving schedulers, a job $\tau_{i,j}$ is ready but unscheduled at any time if and only if the number of idle processors of the γ_i^{th} CE is insufficient to schedule $\tau_{i,j}$. Thus, any work-conserving scheduler satisfies the following:

WC. Under a work-conserving scheduler, among the processors of the p^{th} CE that are allocated to G, there are M'_p idle processors at time t if and only if, for each ready but unscheduled job $\tau_{i,j}$ with $\gamma_i = p$ at time t, $m_i > M'_p$ holds.

Algorithm 7.1 shows pseudocode for an example preemptive work-conserving scheduler. Note that non-preemptive schedulers can also be work-conserving. At any scheduling-decision point, Algorithm 7.1 iterates through all ready jobs to schedule as many jobs as possible in an order. If a job cannot fit on the

Algorithm 7.1 Work-conserving scheduling.

Variables:

Ready(t) : Set of ready jobs at time t

Sched(t) : Set of jobs to be scheduled at time t

```
1: procedure AN EXAMPLE WORK-CONSERVING SCHEDULING
2:     M' ← M
3:     Order jobs in Ready(t) according to the scheduling policy
4:     for each τ_{i,j} ∈ Ready(t) do
5:         if m_i > M' then
6:             continue /* Use break for semi-work-conserving */
7:         Sched(t) ← Sched(t) ∪ {τ_{i,j}}
8:         M' ← M' − m_i
```

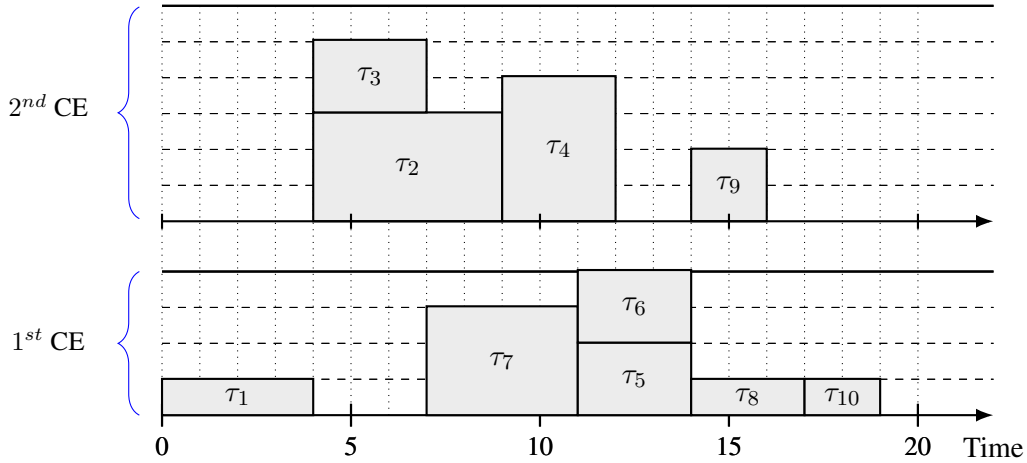


Figure 7.4: A work-conserving schedule of G in Figure 7.2.

available processors, the scheduler attempts to schedule the next job in the order (line 6). Note that the G-EDF scheduling policy considered in Chapter 6 is similar to Algorithm 7.1.

Example 7.3. Figure 7.4 shows a work-conserving schedule of a DAG job of G in Figure 2.4 on two CEs consisting of four and six processors, respectively. At time 4, τ_1's job completes, causing the release of the jobs of τ_2, τ_3, and τ_4. At time 4, jobs of τ_2 and τ_3 are scheduled on the 2^{nd} CE, but τ_4's job cannot be scheduled on the remaining one available processor. τ_3's job executes less than τ_3's WCET and completes at time 7. This causes τ_7 to release its job at time 7 on the 1^{st} CE. At time 7, τ_4's job still cannot be scheduled on the three available processors of the 2^{nd} CE. ◀

Semi-work-conserving scheduling. We now introduce the semi-work-conserving scheduling. A semi-work-conserving scheduler may allow some processors to remain idle even if an unscheduled ready job

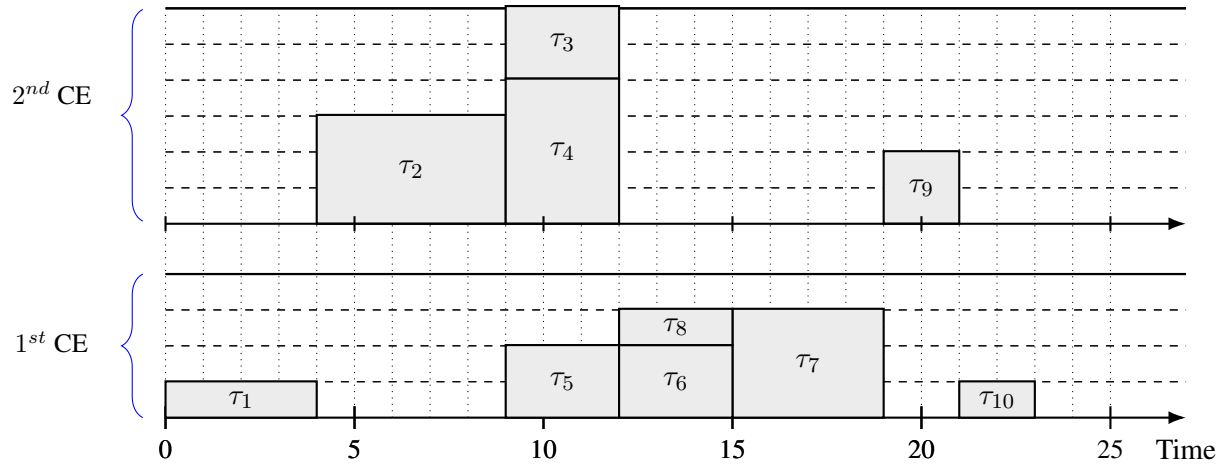


Figure 7.5: A semi-work-conserving schedule of G in Figure 2.4.

could fit there. However, in such a case, there must be another ready but unscheduled job that cannot fit on those processors. Specifically, under a semi-work-conserving scheduler, at any time, a job $\tau_{i,j}$ is ready but unscheduled if and only if the number of idle processors is insufficient to schedule a ready but unscheduled job $\tau_{k,\ell}$. Thus, the following holds.

SC. Under a semi-work-conserving scheduler, among the processors of the p^{th} CE that are allocated to G, there are M'_p idle processors at time t if and only among the ready but unscheduled jobs there exists one $\tau_{k,\ell}$ with $\gamma_k = p$ at time t for which $m_k > M'_p$.

Thus, under semi-work-conserving schedulers, when a job $\tau_{i,j}$ with $\gamma_i = p$ is ready but unscheduled, the number of idle processors M'_p allocated to G on the p^{th} CE can be larger than m_i. However, there must be another job $\tau_{k,\ell}$ with $\gamma_k = p$ such that m_k is larger than M'_p. Algorithm 6.2 can be converted into a semi-work-conserving scheduling algorithm by replacing the statement **continue** at line 6 by **break**.

Example 7.4. Figure 7.5 depicts a semi-work-conserving schedule of a DAG job of G in Figure 2.4 on two CEs consisting of four and six processors, respectively. At time 4, τ_1's job completes, causing the release of the jobs of τ_2, τ_3, and τ_4. At time 4, τ_2's job is scheduled on the 2^{nd} CE. Assume that the scheduler attempts to schedule τ_4's job on the three available processors first. Since it cannot fit there, jobs for τ_3 and τ_4 are not scheduled. After τ_2's job is completed, τ_4's job is scheduled, and the only remaining ready job on the 2^{nd} CE is also scheduled. ◀

Semi-work-conserving scheduling in GPUs. When GPU-accessing tasks share the same address space, NVIDIA GPUs schedule tasks in a semi-work-conserving manner. Recall that processors in NVIDIA GPUs are clustered into SMs. A CUDA-using[2] program launches a *kernel* to be executed on GPU. Each kernel consists of *blocks* of multiple threads that are co-scheduled on an SM. Note that all threads of a block must be scheduled on an SM, *i.e.,* its threads cannot be distributed to multiple SMs.

When a kernel is launched, it moves through a pipeline to enter into a FIFO (EE) *queue*.[3] The blocks of the kernel at the head of the queue are scheduled on SMs. When all blocks at the head of the queue are scheduled, the kernel is removed from the queue, and the new head's blocks are scheduled until no more blocks can fit any SMs. Thus, if a block of the kernel at the head of the queue cannot fit on any SMs, no blocks of the non-head kernels are scheduled even if they can fit on remaining processors on an SM, satisfying SC. Readers interested in the details of scheduling on NVIDIA GPUs are referred to [Amert et al., 2017; Bakita and Anderson, 2024].

7.3 Parallelism-Induced Idleness

Recall from Chapter 6 that rigid gang tasks can cause *parallelism-induced idleness*. Quantifying such idleness was an important step in Chapter 6 to derive G-EDF response-time bounds for sporadic gang tasks. In this section, we quantify such idleness by determining Δ_i (Definition 6.3) values for gang tasks forming a DAG under both work-conserving and semi-work-conserving scheduling. For work-conserving and semi-work-conserving scheduling, we define them as follows.

Definition 7.1. For each task τ_i of G, let Δ_i^W (resp., Δ_i^S) denote the maximum possible number of idle processors, among the M_{γ_i} processors allocated to G on the γ_i^{th} CE, when a job of τ_i is ready but unscheduled under any work-conserving (resp., semi-work-conserving) scheduler. ◀

Since Δ_i^W and Δ_i^S values depend only on G's tasks that execute on the γ_i^{th} CE, we introduce the following notation.

Definition 7.2. For any set $V' \subseteq V$ of tasks, V'_p denotes the tasks in V' that execute on the p^{th} CE, *i.e.,* $V'_p = \{\tau_i \in V' : \gamma_i = p\}$. Thus, V_p denotes all tasks of G that execute on the p^{th} CE. ◀

[2]Although other GPU-programming APIs exist, CUDA is commonly used in real-time systems.

[3]CUDA also provides CUDA *streams* that adds an additional queueing prior to EE queues. Using per-job streams, such queueing can be obviated [Yang et al., 2018].

7.3.1 Work-Conserving Schedulers

Recall from Chapter 6 that for independent sporadic tasks, a dynamic programming algorithm to determine Δ_i^W is known for G-EDF scheduling [Dong and Liu, 2019], which is also applicable to any work-conserving scheduler. This algorithm calculates the smallest number of occupied processors on the γ_i^{th} CE by jobs of a subset of other tasks, but leaving fewer than m_i available processors (thus, maximizing the number of idle processors). However, considering subsets of all tasks other than τ_i can be pessimistic for DAG tasks, as not all tasks can have ready jobs simultaneously with τ_i.

To accurately determine Δ_i^W values, we give an algorithm that considers only those sets of tasks that can have ready jobs simultaneously with τ_i. A set of tasks can simultaneously have ready jobs if no pairs of tasks in that set have ancestor-descendant relationships. To formally define such tasks, we introduce the notation $par(V')$ for any set $V' \subseteq V$, which is *true* if jobs of all tasks in V' can be ready simultaneously, and *false* otherwise:

$$par(V') = \bigwedge_{\tau_i \in V'} \left(\tau_i \notin \bigcup_{\tau_k \in V' \setminus \{\tau_i\}} dep(\tau_k) \right).$$

Using $par(V')$, Δ_i^W can be determined by the following recurrence relation.

$$\Delta_i^W = \begin{cases} 0 & \text{if } \forall\, V' \subseteq V_{\gamma_i} : par(V' \cup \{\tau_i\}) \\ & :: \textstyle\sum_{\tau_k \in V'} m_k \leq M_{\gamma_i} - m_i \\ \max\{m' \in \{0, \cdots, m_i - 1\} : (\exists\, V' \subseteq V_{\gamma_i} \setminus \{\tau_i\} \ : & \text{otherwise} \\ \quad par(V' \cup \{\tau_i\}) \ \wedge \textstyle\sum_{\tau_j \in V'} m_j = M_{\gamma_i} - m')\} & \end{cases} \tag{7.3}$$

The first case in (7.3) sets Δ_i^W to zero, as no set V' of tasks satisfying $par(V')$ can occupy at least $M_{\gamma_i} - m_i$ processors. Thus, τ_i is never ready but unscheduled due to the unavailability of required number of processors. The second case sets Δ_i^W value by determining a subset of tasks V' satisfying $par(V' \cup \{\tau_i\})$, which occupy the smallest number of processors on τ_i's CE and leave fewer than m_i processors for a job of τ_i.

Example 7.5. Consider the DAG G in Figure 7.6 that is scheduled on a CE of ten processors. Only jobs of τ_2, τ_3, τ_5, and τ_6 can be ready when τ_4 has a ready job, as they are neither ancestors nor descendants

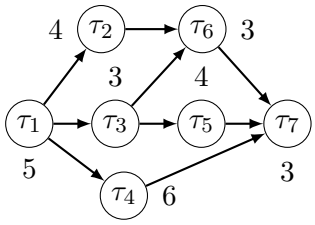


Figure 7.6: A DAG. Numbers outside circles denote m_i values.

of τ_4. However, since τ_3 is a predecessor of τ_6, jobs of τ_3 and τ_6 cannot be simultaneously ready. Thus, $par(\{\tau_3, \tau_4, \tau_6\})$ is *false*. In contrast, $par(\{\tau_2, \tau_4, \tau_5\})$ is *true*, as they can have ready jobs at the same time. Among all sets $V' \subset V \setminus \{\tau_4\}$ such that $par(V' \cup \{\tau_4\}) =$ *true*, tasks $\{\tau_2, \tau_3\}$ require the least number of processors, leaving less than $m_4 = 6$ processors for τ_4. Thus, by (7.3), $\Delta_4^W = 10 - m_2 - m_3 = 10 - 7 = 3$.◀

Computing Δ_i^W by (7.3)**.** We now give a dynamic programming algorithm to compute Δ_i^W values according to the second case of (7.3). Note that the first case is applicable when there is no m' value satisfying the second case. Thus, we only focus on the second case of (7.3). To compute Δ_i^W, we consider the set of tasks $V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\})$. We fill a two-dimensional dynamic-programming table with entries $\Delta_i^W(\tau_j, m)$ where $\tau_j \in V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\})$ and $m \in \{0, 1, \ldots, M_{\gamma_i}\}$. The entry $\Delta_i^W(\tau_j, m)$ stores a boolean value, which is *true* if τ_j and a subset of tasks in $V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\})$ with task indices at most $j - 1$ can execute in parallel and occupy exactly m processors in τ_i's CE, and *false* otherwise.

Example 7.5 (Continued)**.** Consider the DAG in Figure 7.6. For any m, $\Delta_4(\tau_3, m)$ depends on tasks in $V \setminus (dep(\tau_4) \cup \{\tau_4\})$ with task indices at most 3. Such tasks are $\{\tau_2, \tau_3\}$. $\Delta_4(\tau_3, m)$ is true for m values that can be occupied by only τ_3 or both τ_2 and τ_3. Thus, $\Delta_4(\tau_3, 5)$ and $\Delta_4(\tau_3, 9)$ are *true*, as five (resp., nine) processors can be occupied by τ_3 (resp., τ_2 and τ_3). ◀

The recurrence relation in (7.4) determines the $\Delta_i^W(\tau_j, m)$ values. The first case in (7.4) represents a base case; $\Delta_i^W(\tau_j, m_j)$ is *true* as τ_j can occupy m_j processors. The second case considers all tasks $\tau_j \in V_{\gamma_i}$ that cannot execute in parallel with any task $\tau_k \in V_{\gamma_i}$ with $k < j$ and sets the entries corresponding to $m \neq m_j$ as *false*. The final case considers all tasks with task indices smaller than j that are not τ_j's ancestors to determine the existence of a set of tasks occupying exactly $m - m_j$ processors. Note that case three

precludes checking whether τ_k is a descendant of τ_j, as task indexing follows a topological ordering.

$$\Delta_i^W(\tau_j, m) = \begin{cases} true & \text{if } m = m_j \\ false & \text{if } m \neq m_j \wedge (\forall k < j : \\ & \quad \tau_k \notin V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\}) \\ \bigvee_{k<j \wedge \tau_k \in V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\} \cup anc(\tau_j))} \Delta_i^W(\tau_k, m - m_j) & \text{otherwise} \end{cases} \tag{7.4}$$

Finally, to compute Δ_i^W, we determine the smallest m value, say m', larger than $M_{\gamma_i} - m_i$ for which a τ_j exists with $\Delta_i^W(\tau_j, m) = true$. We then set $\Delta_i^W = M_{\gamma_i} - m'$.

Algorithm 7.2 presents pseudocode for computing the entries of the table Δ_i^W according to (7.4). Line 1 creates the table Δ_i^W and initializes every entry to *false*. Consequently, the second case of(7.4) does not need to be explicitly computed. The outer loop in lines 3–14 iterates over each task τ_j and computes the entries of the row $\Delta_i^W(\tau_j, \cdot)$. Lines 4–5 handle the case where $\tau_j \in dep(\tau_i) \cup \{\tau_i\}$, for which all entries $\Delta_i^W(\tau_j, \cdot)$ remain $false$ by the second case of (7.4). The inner loop in lines 6–14 fills in the entries $\Delta_i^W(\tau_j, m)$. Lines 7–8 cover the first case of (7.4), while lines 9–14 apply the third case to determine the value of $\Delta_i^W(\tau_j, m)$. Finally, lines 15–18 identify and return the smallest value of m (if any) greater than $M_{\gamma_i} - m_i + 1$ for which there exists a task τ_j such that $\Delta_i^W(\tau_j, m) = true$. If no such m exists, then the value returned is 0.

Running time. We can compute $dep(\tau_i)$ for all τ_i in $O(|V|^2)$ time [Purdom, 1970]. Computing each entry of the dynamic programming table corresponding to the first case requires $O(1)$ time. Using pre-computed $dep(\tau_i)$, computing an entry corresponding to the second and third cases takes $O(|V|)$ time. Thus, the total running time to compute the dynamic programming table takes $O(|V|^2 M_{\gamma_i})$ time. Computing the Δ_i^W value from the table requires an additional $O(|V| M_{\gamma_i})$ time for scanning all entries in the table. Thus, the time complexity for computing Δ_i^W is $O(|V|^2 M_{\gamma_i})$. Finally, computing Δ_i^W values for all tasks requires $O(|V|^3 \max\{M_{\gamma_i}\})$ time.

7.3.2 Semi-Work-Conserving Schedulers

When a job of τ_i is ready but unscheduled under semi-work-conserving schedulers, the number of idle processors on τ_i's CE can exceed $m_i - 1$. This is because, by (SC), if a task τ_k with $m_k > m_i$ cannot be scheduled due to the unavailability of m_k processors, it may cause τ_i to be unscheduled too. Thus, in such a

Algorithm 7.2 Compute Δ_i^W.

```
1: procedure COMPUTE Δ_i^W
2:     Δ_i^w = n × M_{γ_i} table, each cell initialized to false
3:     for j = 1 to n do
4:         if τ_j ∈ dep(τ_i) ∪ {τ_i} then
5:             continue
6:         for m = 1 to M_{γ_i} do
7:             if m = m_j then
8:                 Δ_i^W(τ_j, m) ← true
9:             for k = 1 to j − 1 do
10:                if τ_k ∈ dep(τ_i) ∪ {τ_i} ∪ anc(τ_j) then
11:                    continue
12:                if m > m_j and Δ_i^W(τ_k, m − m_j) = true then
13:                    Δ_i^W(τ_i, m) ← true
14:                    break
15:    for m = M_{γ_i} − m_i + 1 to M_{γ_i} do
16:        for j = 1 to n do
17:            if Δ_i^W(τ_j, m) = true then
18:                return M_{γ_i} − m
19:    return 0
```

case, the number of idle processors is at most Δ_k^W, *i.e.,* the maximum possible number of idle processors when a job of τ_k cannot be scheduled under work-conserving schedulers. Therefore, Δ_i^S depends on Δ_k^W values of all tasks τ_k that can have ready jobs simultaneously with τ_i. Thus, we define Δ_i^S as follows:

$$\Delta_i^S = \max_{\tau_k \in V_{\gamma_i} \setminus dep(\tau_i)} \{\Delta_k^W\}. \tag{7.5}$$

Proof of (7.5). Assume that the number of idle processors is M' at time t when a job of τ_i is ready but unscheduled. By (SC), there exists a task τ_k in $V_{\gamma_i} \setminus dep(\tau_i)$ with a ready but unscheduled job such that $m_k > M'$ holds. Since τ_k has a ready but unscheduled job, only jobs of tasks in $V_{\gamma_i} \setminus dep(\tau_k)$ can occupy processors on the γ_i^{th} CE. By (7.3), the maximum number of idle processors when tasks in $V_{\gamma_i} \setminus dep(\tau_k)$ occupy more than $M_{\gamma_i} - m_k$ processors of the γ_i^{th} CE is at most Δ_k^W. Thus, since $\tau_k \in V_{\gamma_i} \setminus dep(\tau_i)$, we have $M' \leq \Delta_k^W \leq \max_{\tau_\ell \in V_{\gamma_i} \setminus dep(\tau_i)} \{\Delta_\ell^W\}$, which satisfies (7.5). □

From computed Δ_k^W values, it requires an additional $O(|V_{\gamma_i}|)$ time to compute a Δ_i^S value, thus total $O(|V_{\gamma_i}|^2)$ time to compute such values for all nodes. Including computation times for Δ_k^W values, running time to compute all Δ_i^S is $O(|V|^3 \max_k\{M_k\} + |V|^2) = O(|V|^3 \max_k\{M_k\})$.

Example 7.6. Consider a semi-work-conserving schedule of the DAG G in Figure 7.6 on ten processors. Consider Δ_5^S for τ_5. Only jobs of τ_2, τ_4, and τ_6 can be ready when τ_5 has a ready job. Thus, by (7.5), $\Delta_5^S = \max\{\Delta_2^W, \Delta_4^W, \Delta_6^W\}$. ◀

Δ_i^W and Δ_i^S values for GPUs. Recall that NVIDIA GPUs cluster their processors into SMs and a job of gang tasks must execute on a single SM. The above-mentioned approaches to compute Δ_i^W and Δ_i^S values can be applied for such a case by first determining such values assuming a single SM and then multiplying the values by the number of SMs allocated to the DAG. Thus, if c SMs, each containing M_p/c processors, are allocated to DAG G and $\Delta_{i,c}^W$ is the value computed by (7.3) assuming M_p/c processors, then $\Delta_i^W = c \cdot \Delta_{i,c}^W$.

7.4 Response-Time Bound

In this section, we give a response-time bound for a DAG G of gang tasks that are scheduled on μ CEs under a work-conserving or a semi-work-conserving scheduler. Since G has constrained deadlines, we consider a single DAG job to derive our response-time bound. For notational convenience, we omit job indices, *e.g.,* τ_i denotes both a task and its job. Our analysis technique is the same for work-conserving and semi-work-conserving schedulers. Specifically, replacing Δ_i^W by Δ_i^S from our response-time bound under work-conserving schedulers yields our response-time bound under semi-work-conserving schedulers. Thus, we give a response-time bound under an arbitrary schedule, as assumed in the following definition.

Definition 7.3. Let $\mathcal{S}$ be a schedule of DAG job G on μ CEs where, for all p, M_p processors of the p^{th} CE are assigned to G. For each task τ_i, let $\Delta_i = \Delta_i^W$ (resp., $\Delta_i = \Delta_i^S$), if $\mathcal{S}$ is work-conserving (resp., semi-work-conserving). ◀

Note that, in $\mathcal{S}$, jobs of G may execute for less than their WCETs; our bound is also valid in such a case. Our analysis relies on an *envelope* path of G in $\mathcal{S}$, as defined below.

Definition 7.4. In $\mathcal{S}$, a path of jobs $\{v_1, v_2, \cdots, v_k\}$ of G is an *envelope path* if and only if the following conditions hold.

(i) $v_1 = \tau_1 \wedge v_k = \tau_n$,

(ii) $\forall\, i \in \{1, 2, \ldots, k-1\} : f(v_i) = r(v_{i+1})$,

(iii) $\forall\, i \in \{1, 2, \ldots, k-1\} : v_i \in pred(v_{i+1})$.

We denote an envelope path of G in $\mathcal{S}$ by λ^e. ◀

Example 7.7. We can determine an envelope path by traversing the schedule backward (from sink to source). In Figure 7.4, τ_{10} is released when τ_8 finishes. Similarly, τ_8 is released when τ_4 finishes. Iteratively doing this until τ_1 is reached, an envelope path in Figure 7.4 is $\{\tau_1, \tau_4, \tau_8, \tau_{10}\}$. ◀

Note that there can be multiple envelope paths of G in $\mathcal{S}$. This can happen when multiple predecessor jobs of τ_i complete at the same time. For any task on the envelope path λ^e of G in $\mathcal{S}$, we have the following lemma.

Lemma 7.1. *Let τ_i be a job of λ^e. At any time $t \in [r(\tau_i), f(\tau_i))$, if τ_i is not scheduled, then at least $M_{\gamma_i} - \Delta_i$ processors of the γ_i^{th} CE are busy in $\mathcal{S}$.*

Proof. Follows from Definitions 6.3 and 7.3. □

Let A^e be the union of all intervals when jobs of λ^e execute in $\mathcal{S}$. Also, let A^{ne} be the union of all intervals when no jobs of λ^e execute in $\mathcal{S}$. Thus, $A^e \cap A^{ne} = \emptyset$ holds, and we have

$$|A^e| + |A^{ne}| = f(\tau_n) - r(\tau_1) = R(G). \tag{7.6}$$

Thus, response time $R(G)$ of job G can be upper bounded by upper bounding $|A^e|$ and $|A^{ne}|$. To upper bound $|A^{ne}|$, we define interfering workload for each task on a path.

Definition 7.5. For any τ_i, we let $I(\tau_i) = V_{\gamma_i} \setminus (dep(\tau_i) \cup \{\tau_i\})$. For any set $V' \subseteq V$, we define $I(V') = \bigcup_{\tau_i \in V'} I(\tau_i)$. ◀

Example 7.7 (Continued)**.** In Figure 7.4, for envelope path $\lambda^e = \{\tau_1, \tau_4, \tau_8, \tau_{10}\}$, $A^e = [0, 4) \cup [9, 12) \cup [14, 17) \cup [17, 19)$, when jobs of λ^e executes. In contrast, $A^{ne} = [4, 9) \cup [12, 14)$. By Definition 7.5, $I(\tau_4) = \{\tau_2, \tau_3\}$. ◀

When a job τ_i is ready but unscheduled, the jobs that execute of τ_i's CE are neither an ancestor nor a descendant of τ_i. Thus, we have the following lemma.

Lemma 7.2. *Let τ_i be a job of λ^e. For any time $t \in [r(\tau_i), f(\tau_i))$, if τ_i is not scheduled at time t, then jobs that are scheduled on the γ_i^{th} CE at time t are in $I(\tau_i)$.*

Proof. No jobs in $dep(\tau_i)$ are ready during $[r(\tau_i), f(\tau_i))$. Also, only jobs τ_k with $\gamma_k = \gamma_i$ can be scheduled on τ_i's CE. Thus, the lemma holds. □

Next, we give an upper bound on $|A^{ne}|$. We begin by introducing some necessary notation.

Definition 7.6. Let $\tau_k(V')$ be the task with the k^{th}-highest Δ_i value among the tasks of V', and $\Delta_k(V')$ is its Δ_i value. We assume that ties are broken by task indices. ◀

By Definition 7.6, for an ℓ-node path λ of G, $\{\tau_1(\lambda), \tau_2(\lambda), \ldots, \tau_\ell(\lambda)\}$ is an ordered set of tasks of λ in descending order of Δ_i values. Moreover, since λ_p denotes the set of tasks on λ that execute on the p^{th} CE (by Definition 7.2), $\tau_k(\lambda_p)$ is the task with the k^{th}-largest Δ_i value among all tasks on λ that execute on the p^{th} CE.

Definition 7.7. For any set $V' \subseteq V$ of tasks, let $I_i^{cup}(V') = \bigcup_{j=1}^{i} I(\tau_j(V'))$ and $I_i^{diff}(V') = I(\tau_i(V')) \setminus I_{i-1}^{cup}(V')$. ◀

Thus, by Definitions 7.7 and 7.5, $I_i^{cup}(V')$ consists of all tasks that may interfere with any task in $\{\tau_1(V'), \tau_2(V') \ldots, \tau_i(V')\}$. In contrast, $I_i^{diff}(V')$ consists of tasks that may interfere with task $\tau_i(V')$ but not with any task in $\{\tau_1(V'), \tau_2(V') \ldots, \tau_{i-1}(V')\}$. Thus, $\bigcup_{j=1}^{i} I_j^{diff}(V')$ also consists of all tasks that may interfere with any task in $\{\tau_1(V'), \ldots, \tau_i(V')\}$. Therefore, we have

$$I_i^{cup}(V') = \bigcup_{j=1}^{i} I_j^{diff}(V'). \tag{7.7}$$

Moreover, for any $i \neq j$, $I_i^{diff}(V')$ and $I_j^{diff}(V')$ are disjoint:

$$\forall i \neq j : I_i^{diff}(V') \cap I_j^{diff}(V') = \emptyset. \tag{7.8}$$

Example 7.8. Assume that the DAG in Figure 7.6 is scheduled by a work-conserving scheduler on a CE of ten processors. Let $V' = \{\tau_5, \tau_6\}$. According to (7.3), $\Delta_5 = 1$ and $\Delta_6 = 0$. Thus, by Definition 7.6, $\tau_1(V') = \tau_5$ and $\tau_2(V') = \tau_6$. By Definition 7.5, $I(\tau_5) = \{\tau_2, \tau_4, \tau_6\}$ and $I(\tau_6) = \{\tau_4, \tau_5\}$. Thus, by Def 7.7, $I_2^{cup}(V') = I(\tau_5) \cup I(\tau_6) = \{\tau_2, \tau_4, \tau_5, \tau_6\}$. By Def 7.7, $I_1^{diff}(V') = I(\tau_5) = \{\tau_2, \tau_4, \tau_6\}$ and $I_2^{diff}(V') = I(\tau_6) \setminus I(\tau_5) = \{\tau_5\}$. ◀

Definition 7.8. For any subset of tasks V'_p assigned to the p^{th} CE (Definition 7.2) and $1 \leq j \leq |V'_p|$, we define $F(V'_p, j)$ as follows:

$$F(V'_p, j) = \sum_{i=1}^{j} \frac{vol(I_i^{diff}(V'_p)))}{M_p - \Delta_i(V'_p)}. \tag{7.9}$$

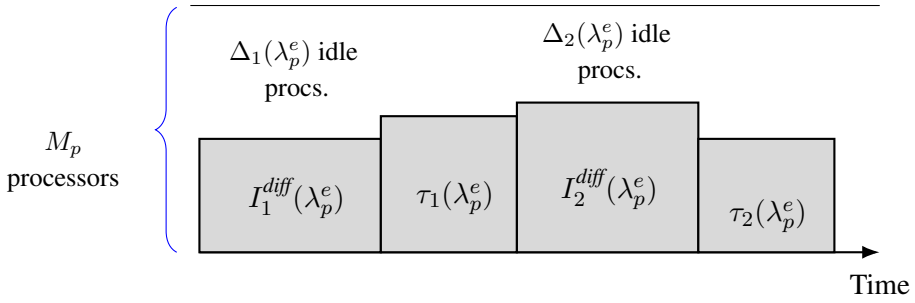


Figure 7.7: Proof of Lemma 7.3.

◀

Intuitively, $F(V'_p, j)$ is computed by considering j nodes in V'_p that have the largest Δ_i values. For each such node τ_i, the interfering workload (the numerator) is computed by considering nodes that interfere with τ_i but not with any node with larger Δ_i values. Dividing the interfering workload by the number of minimum possible busy processors when τ_i is ready but not executing yields a time duration when τ_i cannot execute due to such interfering workload. The $F(V'_p, j)$ values contains the sum these durations for each node.

Using Definition 7.8, in the following lemma, we upper bound $|A^{ne}|$ by considering all tasks in λ_p^e in decreasing order of their Δ_i values, *i.e.,* in the order: $\tau_1(\lambda_p^e), \ldots, \tau_{|\lambda_p^e|}(\lambda_p^e)$. Under such an ordering, we assume that a task that can interfere with multiple tasks on λ_p^e executes when the task with the largest Δ_i value is ready but unscheduled. Figure 7.7 illustrates the idea: a task that can interfere with both $\tau_1(\lambda_p^e)$ and $\tau_2(\lambda_p^e)$ is scheduled when tasks of $I_1^{diff}(\lambda_p^e)$ are executing, *i.e.,* $I_1^{diff}(\lambda_p^e)$ contains that task.

Lemma 7.3. $|A^{ne}| \leq \sum_{p=1}^{\mu} F(\lambda_p^e, |\lambda_p^e|)$ *holds.*

Proof. For any job $\tau_i(\lambda_p^e) \in \lambda_p^e$ with $1 \leq p \leq \mu$, let $A_{i,p}^{ne}$ be the union of intervals when $\tau_i(\lambda_p^e)$ is ready but unscheduled. Note that $\tau_i(\lambda_p^e)$ follows Definition 7.6. Thus, $A_{i,p}^{ne} \subseteq A^{ne}$. By the definition of A^{ne}, we have $\bigcup_{p=1}^{\mu} \bigcup_{i=1}^{|\lambda_p^e|} A_{i,p}^{ne} = A^{ne}$. Moreover, by Definition 7.4, no two jobs of λ^e are ready at the same time. Thus, for any pair of jobs $\tau_i(\lambda_p^e)$ and $\tau_j(\lambda_q^e)$ on λ^e with $\tau_i(\lambda_p^e) \neq \tau_j(\lambda_q^e)$, $A_{i,p}^{ne} \cap A_{j,q}^{ne} = \emptyset$ holds. Therefore, we have

$$|A^{ne}| = \sum_{p=1}^{\mu} \sum_{i=1}^{|\lambda_p^e|} |A_{i,p}^{ne}|. \tag{7.10}$$

We now upper bound A^{ne} by upper bounding $\sum_{i=1}^{|\lambda_p^e|} |A_{i,p}^{ne}|$ in (7.10) for all $1 \leq p \leq \mu$. For any job τ_k on the p^{th} CE, let τ_k execute for $C_{k,i,p}$ time units during $A_{i,p}^{ne}$. By Lemma 7.2, jobs not in $I(\tau_i(\lambda_p^e))$ cannot

execute on the p^{th} CE during $A^{ne}_{i,p}$. Thus, we have

$$\forall \tau_k \notin I(\tau_i(\lambda^e_p)) : \gamma_k = p :: C_{k,i,p} = 0. \tag{7.11}$$

Thus, the total execution on the p^{th} CE during $A^{ne}_{i,p}$ is $\sum_{\tau_k \in I(\tau_i(\lambda^e_p))} m_k C_{k,i,p}$. By Lemma 7.1 and Definition 7.6, at least $M_p - \Delta_i(\lambda^e_p)$ processors of the p^{th} CE are busy during $A^{ne}_{i,p}$. Hence, we can upper bound the length $|A^{ne}_{i,p}|$ as follows:

$$|A^{ne}_{i,p}| \leq \frac{\sum_{\tau_k \in I(\tau_i(\lambda^e_p))} m_k C_{k,i,p}}{M_p - \Delta_i(\lambda^e_p)}.$$

Therefore, we have

$$\sum_{i=1}^{|\lambda^e_p|} |A^{ne}_{i,p}| \leq \sum_{i=1}^{|\lambda^e_p|} \frac{\sum_{\tau_k \in I(\tau_i(\lambda^e_p))} m_k C_{k,i,p}}{M_p - \Delta_i(\lambda^e_p)}. \tag{7.12}$$

By Definition 7.5, $I(\lambda^e_p)$ denotes the set $\bigcup_{\tau_k \in \lambda^e_p} I(\tau_k)$. Using $I(\lambda^e_p)$, we can rearrange (7.12) as follows:

$$\sum_{i=1}^{|\lambda^e_p|} |A^{ne}_{i,p}| \leq \sum_{\tau_k \in I(\lambda^e_p)} \sum_{i=1}^{|\lambda^e_p|} \frac{m_k C_{k,i,p}}{M_p - \Delta_i(\lambda^e_p)}. \tag{7.13}$$

Now, consider a job $\tau_k \in I(\lambda^e_p)$. Let $sm(k)$ be the smallest i value such that $\tau_k \in I(\tau_i(\lambda^e_p))$, *i.e.,* $\forall j < sm(k), \tau_k \notin I(\tau_j(\lambda^e_p))$. By (7.11), for any $j < sm(k)$, τ_k does not execute during $A^{ne}_{j,p}$, *i.e.,* $C_{k,j,p} = 0$. Therefore, by (7.13), we have

$$\begin{aligned}
\sum_{i=1}^{|\lambda^e|} |A^{ne}_{i,p}| &\leq \sum_{\tau_k \in I(\lambda^e_p)} \sum_{i=sm(k)}^{|\lambda^e_p|} \frac{m_k C_{k,i,p}}{M_p - \Delta_i(\lambda^e_p)} \\
&\leq \{\text{By Definition 7.6, } \Delta_i(\lambda^e_p) \geq \Delta_{i+1}(\lambda^e_p)\} \\
&\quad \sum_{\tau_k \in I(\lambda^e_p)} \sum_{i=sm(k)}^{|\lambda^e_p|} \frac{m_k C_{k,i,p}}{M_p - \Delta_{sm(k)}(\lambda^e_p)} \\
&\leq \{\text{Since } \sum_{i=sm(k)}^{|\lambda^e_p|} C_{k,i,p} \leq C_k\} \\
&\quad \sum_{\tau_k \in I(\lambda^e_p)} \frac{m_k C_k}{M_p - \Delta_{sm(k)}(\lambda^e_p)}.
\end{aligned} \tag{7.14}$$

Now, by the definition of $sm(k)$, $sm(k) = i$ holds when τ_k is in $I(\tau_i(\lambda_p^e))$ but not in any $I(\tau_j(\lambda_p^e))$ with $j < i$. Thus, $sm(k) = i$ holds if and only if $\tau_k \in I(\tau_i(\lambda_p^e)) \setminus \bigcup_{j=1}^{i-1} I(\tau_j(\lambda_p^e)) = I_i^{diff}(\lambda_p^e)$ (by Definition 7.7). Thus, by (7.14) and (7.9),

$$\sum_{i=1}^{|\lambda_p^e|} |A_i^{ne}| \leq \sum_{i=1}^{|\lambda_p^e|} \frac{vol(I_i^{diff}(\lambda_p^e))}{M_p - \Delta_i(\lambda_p^e)} = F(\lambda_p^e, |\lambda_p^e|).$$

The lemma holds by applying the above inequality in (7.10). □

Applying Lemma 7.3 in (7.6), we have the following lemma.

Lemma 7.4. *In $\mathcal{S}$, the response time of G is bounded as follows: $R(G) \leq len(\lambda^e) + \sum_{p=1}^{\mu} F(\lambda_p^e, |\lambda_p^e|)$.*

Proof. By the definition of A^e, $|A^e| \leq len(\lambda^e)$ holds. Applying $|A^e| \leq len(\lambda^e)$ and Lemma 7.3 in (7.6), the lemma holds. □

Now, G's response time can be upper bounded by considering all complete paths as an envelope path in Lemma 7.4.

Theorem 7.1. *Let $\Lambda(G)$ be the set of all complete paths of G. G's response time is bounded as follows.*

$$R(G) \leq \max_{\lambda \in \Lambda(G)} \left(len(\lambda) + \sum_{p=1}^{\mu} F(\lambda_p, |\lambda_p|) \right) \tag{7.15}$$

Proof. Follows from Lemma 7.4. □

Unfortunately, even for the special case of sequential nodes (*i.e.,* $\Delta_i = 0$), computing the exact value of the right-hand-side of (7.15) is NP-hard in the strong sense [Han et al., 2019, Theorem 4.2]. Moreover, the variation in the number of idle processors (Δ_i values) during different sub-intervals of A^{ne} (see Figure 7.7) complicates the application of existing approaches to upper bound (7.15), as in [Han et al., 2019], for the sequential case (where the denominator in (7.9) is always M_p). Thus, such approaches can focus only on maximizing the numerator of (7.9).

Upper bounding (7.15)**.** To upper bound (7.15), instead of the tasks of λ_p, we consider the tasks of V_p in order of decreasing Δ_i values. Thus, we consider tasks of V_p in the order: $\tau_1(V_p), \tau_2(V_p), \ldots, \tau_{|V_p|}(V_p)$. Since $\lambda_p \subseteq V_p$, considering the tasks of V_p in such an order assumes that more processors are idle during A^{ne}. This enables upper bounding $R(G)$ without determining the path λ that maximizes (7.15).

We now show how to upper bound $F(\lambda_p, |\lambda_p|)$ in (7.15) using the tasks of V_p in order of decreasing Δ_i values. Since $\lambda_p \subseteq V_p$, the volume of tasks that may interfere with tasks in λ_p (numerator in (7.9)) can be expressed as the volume of tasks that may interfere with a subset of tasks $\tau_1(V_p), \ldots, \tau_k(V_p)$. In Lemma 7.5, we determine such an equivalent interfering workload from a subset of tasks in V_p and divide them by the corresponding Δ_i values to upper bound $F(\lambda_p, j)$ for any j. We first give the following notation.

Definition 7.9. For any set of tasks V', we denote by $vi_k(V')$ the volume of all tasks that may interfere with any tasks in $\{\tau_1(V'), \ldots, \tau_k(V')\}$. Thus, by Definition 7.7, $vi_k(V') = vol(I_k^{cup}(V'))$. We define $vi_0(V') = 0$. Furthermore, by (7.7) and (7.8), $vi_k(V')$ satisfies the following:

$$vi_k(V') = vol\left(\bigcup_{i=1}^{k} I_i^{diff}(V')\right) = \sum_{i=1}^{k} vol\left(I_i^{diff}(V')\right). \tag{7.16}$$

◀

Note that, by Definitions 7.5 and 7.9, the volume of all tasks that may interfere with any task of V' is

$$vol(I(V')) = vi_{|V'|}(V'). \tag{7.17}$$

Lemma 7.5. *For any path λ, CE type p, and $j \leq |\lambda_p|$, let $1 \leq z(j) \leq |V_p|$ be the smallest integer so that $vi_j(\lambda_p) \leq vi_{z(j)}(V_p)$. Let $x(j) = vi_j(\lambda_p) - vi_{z(j)-1}(V_p)$. Then,*

$$F(\lambda_p, j) \leq F(V_p, z(j) - 1) + \frac{x(j)}{M_p - \Delta_{h(V_p, z(j))}}. \tag{7.18}$$

Proof. Note that $vi_j(\lambda_p) = vi_{z(j)-1}(V_p) + x(j)$. Thus, the volume of tasks that interfere with $\{\tau_1(\lambda_p), \tau_2(V_p), \ldots, \tau_j(\lambda_p)\}$ is the same as the sum of $x(j)$ and the volume of tasks that interfere with $\{\tau_1(V_p), \tau_2(V_p), \ldots, \tau_{z(j)-1}(V_p)\}$.

We first consider the case where $vi_j(\lambda_p) = 0$. Then, by (7.16), $vol(I_i^{diff}(\lambda_p)) = 0$ for all $i \leq j$. Thus, by (7.9), $F(\lambda_p, j) = 0$, and the lemma trivially holds.

We now consider $vi_j(\lambda_p) > 0$. Since $z(j)$ is the smallest integer with $vi_j(\lambda_p) \leq vi_{z(j)}(V_p)$, we have

$$vi_{z(j)-1}(V_p) < vi_j(\lambda_p). \tag{7.19}$$

We start by showing the following:

Claim 7.1. $\Delta_{z(j)}(V_p) \geq \Delta_j(\lambda_p)$.

Proof. Assume that $\Delta_{z(j)}(V_p) < \Delta_j(\lambda_p)$. Thus, since $\lambda_p \subseteq V_p$, by Definition 7.6, each task in $\{\tau_1(\lambda_p), \ldots, \tau_j(\lambda_p)\}$ is also in $\{\tau_1(V_p), \ldots, \tau_{z(j)-1}(V_p)\}$. Therefore, by Definitions 7.5 and 7.7, $I_j^{cup}(\lambda_p) \subseteq I_{z(j)-1}^{cup}(V_p)$, which by (7.7) implies that

$$\bigcup_{i=1}^{j} I_i^{diff}(\lambda_p) \subseteq \bigcup_{i=1}^{z(j)-1} I_i^{diff}(V_p)$$
$$\xRightarrow{\text{By (7.16)}} vi_j(\lambda_p) \leq vi_{z(j)-1}(V_p).$$

This contradicts (7.19). Thus, the claim holds. □

We now prove the lemma. We give a proof by induction on index j. Assume that the lemma holds for $j-1$:

$$F(\lambda_p, j-1) \leq F(V_p, z(j-1)-1) + \frac{x(j-1)}{M_p - \Delta_{z(j-1)}(V_p)}. \tag{7.20}$$

By (7.9), we have

$$\begin{aligned} F(\lambda_p, j) &= F(\lambda_p, j-1) + \frac{vol(I_j^{diff}(\lambda_p))}{M_p - \Delta_j(\lambda_p)} \\ &\leq \{\text{By (7.20) and Claim 7.1}\} \\ &\quad F(V_p, z(j-1)-1) + \frac{x(j-1)}{M_p - \Delta_{z(j-1)}(V_p)} + \frac{vol(I_j^{diff}(\lambda_p))}{M_p - \Delta_{z(j)}(V_p)}. \end{aligned} \tag{7.21}$$

To prove the lemma, we now express $vol(I_j^{diff}(\lambda_p))$ in (7.21) using the volume of a subset of tasks in V_p. By (7.16), we have $vol(I_j^{diff}(\lambda_p)) = vi_j(\lambda_p) - vi_{j-1}(\lambda_p)$. Applying the definition of $z(\cdot)$ and $x(\cdot)$ in $vi_j(\lambda_p) - vi_{j-1}(\lambda_p)$, we have $vol(I_j^{diff}(\lambda_p)) = vi_{z(j)-1}(V_p) + x(j) - vi_{z(j-1)-1}(V_p) - x(j-1) = \sum_{i=z(j-1)}^{z(j)-1} vol(I_i^{diff}(V_p)) + x(j) - x(j-1)$. Dividing this equation by $M_p - \Delta_{z(j)}(V_p)$ yields

$$\begin{aligned} \frac{vol(I_j^{diff}(\lambda_p))}{M_p - \Delta_{z(j)}(V_p)} &= \sum_{i=z(j-1)}^{z(j)-1} \frac{vol(I_i^{diff}(V_p))}{M_p - \Delta_{z(j)}(V_p)} + \frac{x(j) - x(j-1)}{M_p - \Delta_{z(j)}(V_p)} \\ &\leq \{\text{By Definition 7.6, for any } i \leq z(j), \Delta_i(V_p) \geq \Delta_{z(j)}(V_p)\} \end{aligned}$$

$$\sum_{i=z(j-1)}^{z(j)-1} \frac{vol(I_i^{diff}(V_p))}{M_p - \Delta_i(V_p)} + \frac{x(j)}{M_p - \Delta_{z(j)}(V_p)} - \frac{x(j-1)}{M_p - \Delta_{z(j-1)}(V_p)}.$$

Applying the above inequality in (7.21), we have

$$\begin{aligned} F(\lambda_p, j) &\leq F(V_p, z(j-1)-1) + \sum_{i=z(j-1)}^{z(j)-1} \frac{vol(I_i^{diff}(V_p))}{M_p - \Delta_i(V_p)} + \frac{x(j)}{M_p - \Delta_{z(j)}(V_p)} \\ &= \{\text{By (7.9)}\} \\ &= F(V_p, z(j)-1) + \frac{x(j)}{M_p - \Delta_{z(j)}(V_p)}. \end{aligned}$$

This completes the proof of the lemma. □

Using (7.18) to upper bound $F(\lambda_p, |\lambda_p|)$ requires determining $y(|\lambda_p|)$ and $x(|\lambda_p|)$. By Lemma 7.18 and (7.21), determining such values requires determining $vi_{|\lambda_p|}(\lambda_p) = vol(I(\lambda_p))$. We upper bound $vol(I(\lambda_p))$ by determining a path $\lambda^{min(p)}$ for which the volume of tasks on the p^{th} CE is the minimum. Since tasks on a path do not contribute to its interfering workload, for any λ, $vol(I(\lambda_p))$ does not exceed $vol(V_p) - vol(\lambda_p^{min(p)})$. Using the following definition, this is shown in Lemma 7.6.

Definition 7.10. For any $1 \leq p \leq \mu$, let $\Lambda_p(G)$ be the set of all complete paths that have at least one task assigned to the p^{th} CE. Let $\lambda^{\min(p)}$ be a path in $\Lambda_p(G)$ with the minimum $vol(\lambda_p^{\min(p)})$, *i.e.,* $\lambda^{\min(p)} = \arg\min_{\lambda \in \Lambda_p(G)}(vol(\lambda_p))$. ◀

Lemma 7.6. *For any path λ and $1 \leq p \leq \mu$, $vol(I(\lambda_p)) \leq vol(V_p) - vol(\lambda_p^{\min(p)})$.*

Proof. If $|\lambda_p| = 0$, then $vol(I(\lambda_p)) = 0$, and the lemma trivially holds. Assuming $|\lambda_p| > 0$, by Definition 7.10, $vol(\lambda_p) \geq vol(\lambda_p^{min(p)})$. Since each task in $I(\lambda_p)$ executes on the p^{th} CE, by Definition 7.5, $I(\lambda_p) \subseteq V_p$. However, by Definition 7.5, $I(\lambda_p)$ does not contain any task $\tau_i \in \lambda_p \subseteq V_p$. Thus, $vol(I(\lambda_p)) \leq vol(V_p) - vol(\lambda_p)$ holds. Since, $vol(\lambda_p) \geq vol(\lambda_p^{min(p)})$, we have $vol(I(\lambda_p)) \leq vol(V_p) - vol(\lambda_p^{\min(p)})$. □

We now define the two terms $h(p)$ and $g(p)$ that can be used, instead of $y(|\lambda_p|)$ and $x(|\lambda_p|)$, to upper bound $F(\lambda_p, |\lambda_p|)$. Using these values, we upper bound $F(\lambda_p, |\lambda_p|)$ in Lemma 7.7 by adding (potentially) more terms in (7.18).

Definition 7.11. For any $1 \leq p \leq \mu$, let $1 \leq h(p) \leq |V_p|$ be the smallest integer with $vol(V_p) - vol(\lambda_p^{\min(p)}) \leq vi_{h(p)}(V_p)$ holds. Also, let $g(p) = vol(V_p) - vol(\lambda_p^{\min(p)}) - vi_{h(p)-1}(V_p)$. ◀

Lemma 7.7. *For any λ and $1 \leq p \leq \mu$, the following holds.*

$$F(\lambda_p, |\lambda_p|) \leq F(V_p, h(p) - 1) + \frac{g(p)}{M_p - \Delta_{h(p)}(V_p)}$$

Proof. Let $y(|\lambda_p|)$ be the smallest integer so that $vi_{|\lambda_p|}(\lambda_p) \leq vi_{y(|\lambda_p|)}(V_p)$ holds. Let $x(|\lambda_p|) = vi_{|\lambda_p|}(\lambda_p) - vi_{y(|\lambda_p|)-1}(V_p)$. By Lemma 7.5, we have

$$F(\lambda_p, |\lambda_p|) \leq F(V_p, y(|\lambda_p|) - 1) + \frac{x(|\lambda_p|)}{M_p - \Delta_{y(|\lambda_p|)}(V_p)}. \tag{7.22}$$

By (7.17), we have $vol(I(\lambda_p)) = vi_{|\lambda_p|}(\lambda_p)$. By Lemma 7.6, $vol(I(\lambda_p)) = vi_{|\lambda_p|}(\lambda_p) \leq vol(V_p) - vol(\lambda_p^{min(p)})$. Therefore, by the definition of $y(|\lambda_p|)$ and $h(p)$, we have $y(|\lambda_p|) \leq h(p)$. We now consider two cases.

Case 1. $y(|\lambda_p|) = h(p)$. Since $vol(I(\lambda_p)) = vi_{|\lambda_p|}(\lambda_p) \leq vol(V_p) - vol(\lambda_p^{min(p)})$, by the definition of $x(|\lambda_p|)$ and $g(p)$, we have $x(|\lambda_p|) \leq g(p)$. Thus, the lemma holds by replacing $y(|\lambda_p|)$ and $x(|\lambda_p|)$ with $h(p)$ and $g(p)$, respectively, in (7.22).

Case 2. $y(|\lambda_p|) < h(p)$. By the definition of $x(|\lambda_p|)$, $x(|\lambda_p|) = vi_{|\lambda_p|}(\lambda_p) - vi_{y(|\lambda_p|)-1}(V_p) \leq vi_{y(|\lambda_p|)}(V_p) - vi_{y(|\lambda_p|)-1}(V_p) = vol(I^{diff}_{y(|\lambda_p|)}(V_p))$. Replacing $x(|\lambda_p|)$ with $vol(I^{diff}_{y(|\lambda_p|)}(V_p))$ and adding additional non-negative terms in (7.22), we have

$$\begin{aligned} F(\lambda_p, |\lambda_p|) &\leq F(V_p, y(|\lambda_p|) - 1) + \frac{vol(I^{diff}_{y(|\lambda_p|)}(V_p))}{M_p - \Delta_{y(|\lambda_p|)}(V_p)} + \sum_{i=y(|\lambda_p|)+1}^{h(p)-1} \frac{vol(I^{diff}_i(V_p))}{M_p - \Delta_i(V_p)} + \frac{g(p)}{M_p - \Delta_{h(p)}(V_p)} \\ &= F(V_p, h(p) - 1) + \frac{g(p)}{M_p - \Delta_{h(p)}(V_p)}. \end{aligned}$$

Thus, the lemma holds. □

Using Lemma 7.7, we have the following theorem.

Theorem 7.2. *G's response time is bounded as follows:*

$$R(G) \leq len(G) + \sum_{p=1}^{\mu} \left(F(V_p, h(p) - 1) + \frac{g(p)}{M_p - \Delta_{h(p)}(V_p)} \right), \tag{7.23}$$

where $len(G)$ denotes length of the longest path of G.

Proof. The theorem follows from applying $len(G) = \max_{\lambda \in \Lambda(G)} len(\lambda)$ and Lemma 7.7 in Theorem 7.1. □

Running time. $len(G)$ and each $vol(\lambda_p^{\min(p)})$ can be computed in $O(V + E)$ time. For DAG, the set $I(\tau_i)$ for all tasks can be computed in $O(|V|^2)$ time. Using precomputed $I(\tau_i)$ sets, for each task, each I^{diff} set (to compute $F(V_p, h(p) - 1)$) can be computed in $O(|V|)$ time. Since each task appears at most once in the response-time bound expression in (7.23), computing all numerators in (7.23) takes $O(|V|^2)$ time. Since computing all Δ_i values takes $O(|V|^3 \max_p\{M_p\})$ time, the total running time is $O(|V|^3 \max_p\{M_p\})$.

7.5 Processor Allocation

In this section, we give an ILP to allocate processors among multiple DAGs. We consider DAGs $G^1, G^2, \cdots, G^N$. For all introduced notation, we use a superscript k to denote the corresponding term for the k^{th} DAG G^k. We also assume that the p^{th} CE has $\mathcal{M}_p$ processors.

For DAG G^k, let $R^k_{p,m}$ denote the value $\left(F(V_p^k, h^k(p) - 1) + \frac{g^k(p)}{m - \Delta_{h^k(p)}(V_p^k)}\right)$ of (7.23) and $len(G^k)$ denote its longest-path length. Using this notation, the ILP is specified as follows.

Variables : For each pair of DAG G^k and p^{th} CE, we define $\mathcal{M}_p$ variables $x^k_{p,1}, x^k_{p,2}, \cdots, x^k_{p,\mathcal{M}_p}$. Variable $x^k_{p,m}$ is 1 if DAG G^k is assigned m processors on the p^{th} CE, and 0 otherwise.

Constraint 1. For each pair of DAG and CE, exactly one $x^k_{p,m}$ is 1 (the DAG receives m processors on that CE):

$$\forall k \in \{1, \cdots, N\}, \forall p \in \{1, \cdots, \mu\} :: \sum_{m=1}^{\mathcal{M}_p} x^k_{p,m} = 1.$$

Constraint 2. The total number of allocated processors per CE is at most the number of processors that CE has:

$$\forall p \in \{1, \cdots, \mu\} :: \sum_{k=1}^{N} \sum_{m=1}^{\mathcal{M}_p} m \cdot x^k_{p,m} \leq \mathcal{M}_p.$$

Constraint 3. On allocated processors, each DAG meets its deadline:

$$\forall k \in \{1, \cdots, N\} : len(G^k) + \sum_{p=1}^{\mu} \sum_{m=1}^{\mathcal{M}_p} R^k_{p,m} x^k_{p,m} \leq D^k.$$

The ILP has $N \sum_{p=1}^{\mu} \mathcal{M}_p$ variables and $N\mu + N + \mu$ constraints.

7.6 Experimental Evaluation

We now present the results of the experiments we conducted to evaluate the response-time bounds of our approach. First, we compared schedulability under work-conserving and semi-work-conserving scheduling for a DAG on an arbitrary number of processors. Second, for multicore+GPU platforms, we compared schedulability under semi-work-conserving scheduling on GPUs with traditional locking-based approaches [Ali et al., 2024]. Finally, we demonstrated the practicality of our approach via a case study on a multicore+GPU platform.

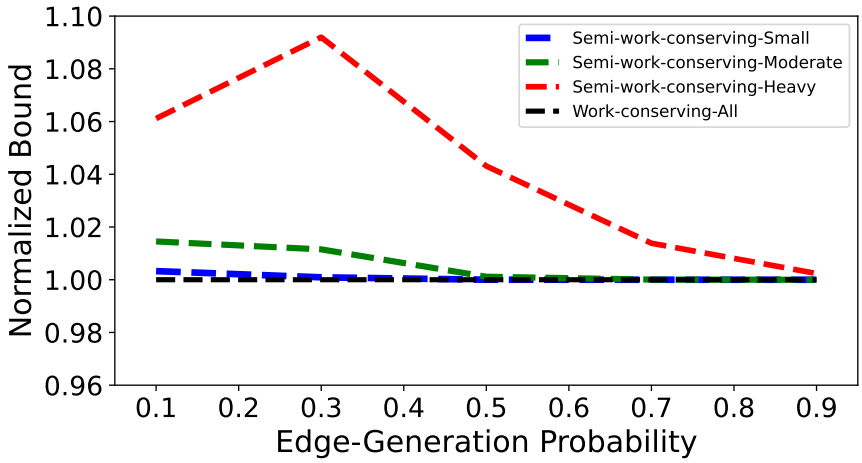
7.6.1 Experiments on Arbitrary Number of CEs

In this experiment, we compared the response-time bounds in Theorem 7.2 under work-conserving and semi-work-conserving schedulers. We generated DAGs following the *Erdős-Rényi method* [Cordeiro et al., 2010]. The number of nodes per DAG was selected from $[20, 120]$. Each task's WCET was chosen from $[50, 100]$. For each pair of nodes (τ_i, τ_j) with $i < j$, an edge from τ_i to τ_j was added if a uniformly generated random number in $[0, 1]$ was at most a predefined *edge-generation probability*. We selected this probability value from $\{0.1, 0.3, 0.5, 0.7, 0.9\}$. A higher edge-generation probability makes DAGs more sequential. As in [Saifullah et al., 2014], additional edges were added to make each DAG weakly connected.

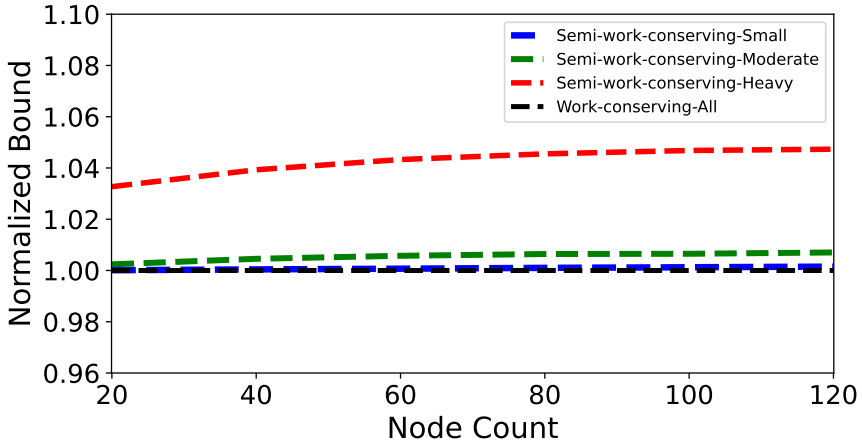
The number of CEs was randomly selected from $[2, 6]$. The number of processors per CE was selected from $\{8, 16, 24, 32\}$, which represent common values in real-world use cases [Akesson et al., 2022; Kato et al., 2018]. Each task was assigned to one of the CEs with uniform probabilities. We considered *small*, *moderate*, or *heavy* degrees of parallelism, for which m_i values were uniformly distributed in $[1, 0.2M_{\gamma_i}]$, $[1, 0.4M_{\gamma_i}]$, and $[1, 0.7M_{\gamma_i}]$, respectively, where M_{γ_i} is the number of processors on τ_i's CE. For each combination of edge-generation probabilities and degrees of parallelism, we generated 1,000 task sets.

We computed the average *normalized response-time bound*, which is the ratio between the response-time bound under semi-work-conserving and work-conserving scheduling. Thus, normalized response-time bounds less than 1.0 imply smaller response-time bounds under semi-work-conserving scheduling than under work-conserving scheduling. The normalized response-time bounds are plotted in Figure 7.8.

Observation 7.1. *For small, moderate, and heavy degrees of parallelism, the average response-time bounds under semi-work-conserving scheduling were* $1.001\times$*,* $1.005\times$*, and* $1.04\times$ *of those under work-conserving scheduling, respectively.*



(a) Normalized bound vs. edge-generation probability.



(b) Normalized bound vs. node count.

Figure 7.8: Results of experiments on arbitrary number of CEs.

For heavy degrees of parallelism, semi-work-conserving scheduling caused larger response-time bounds compared to work-conserving scheduling. This is because the amount of wasted processing capacity (Δ_i values) due to each task can often be larger for semi-work-conserving scheduling (see (7.5)). For smaller edge-generation probabilities, the difference in response-time bounds between work-conserving and semi-work-conserving scheduling increased. (see Figure 7.8(a)). This is because the interfering workload (the summation term in (7.23)) contributed more significantly to the response-time bounds. Increasing the number of nodes slightly increased the normalized response-time bounds (see Figure 7.8(b)). For small degrees of parallelism, the response-time bounds under both work-conserving and semi-work-conserving scheduling were close, as Δ_i values under both scheduling were small. Note that semi-work-conserving scheduling becomes work-conserving when all tasks have $m_i = 1$.

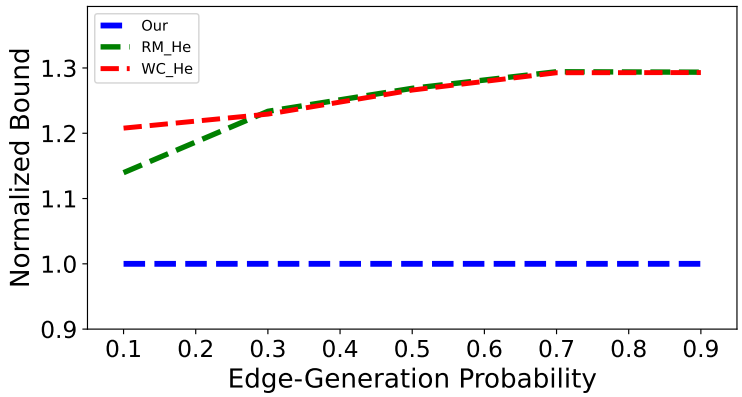
7.6.2 Experiments on Multicore+GPU

In this experiment, we considered systems scheduled on multicore+GPU platforms. We compared our response-time bounds under semi-work-conserving scheduling with a locking-based approach. Under locking-based approach, each GPU access is protected by a lock, *i.e.,* a GPU-accessing CPU task must hold a lock before it can launch its GPU kernel. For the locking-based approach, we considered a recently proposed locking protocol, called the SMLP, which allows multiple jobs to access a GPU simultaneously by allocating SMs among them [Ali et al., 2024]. Under the SMLP, an upper bound on s-oblivious pi-blocking time can be derived under any JLFP scheduling [Ali et al., 2024]. Using such a pi-blocking bound, DAG response-time bounds can be derived by inflating task WCETs and then applying any s-oblivious response-time analysis techniques for the used scheduling algorithm.

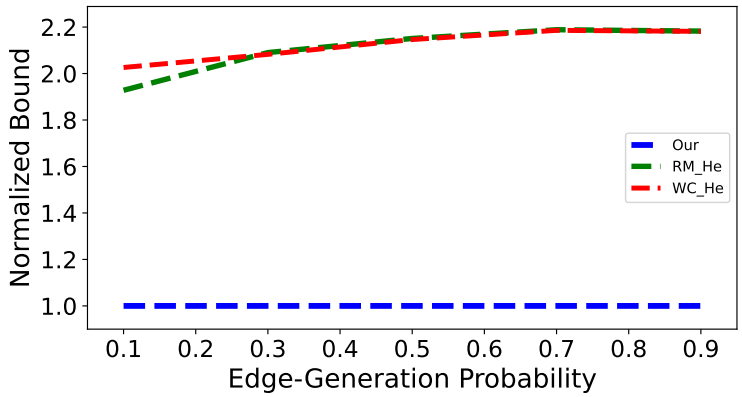
For the locking-based approach, we considered two state-of-the-art response-time bounds: RM-HE [He et al., 2021] and WC-HE [He et al., 2022]. RM-HE considers *prioritized list scheduling* of a DAG of sequential tasks and supports multi-DAG systems by prioritizing different DAGs by the *rate-monotonic* algorithm. WC-HE applies under any work-conserving scheduler, where multiple DAGs are supported by federated scheduling techniques.

Single-DAG systems. To describe task generation, we use NVIDIA-GPU-specific terms. Our GPU-specific task parameter generation was inspired by prior work [Ali et al., 2024; Wang et al., 2024; Patel et al., 2018]. We considered platforms consisting of $\{8, 16, 24, 32\}$ processors and $\{16, 32, 48\}$ SMs, where each SM consists of 2,048 GPU threads. We first generated *coarse-grained* DAG tasks consisting of sequential CPU tasks by the same task-generation method given in Section 7.6.1, where we set $\mu = 1$ and $m_i = 1$ to generate only CPU tasks. We randomly selected some CPU tasks as *GPU-accessing* tasks. We considered *small* ([1–20]%), *moderate* ([20-50]%), and *heavy* ([50–80]%) ratios of GPU-accessing tasks. We generated GPU-access lengths according to the method in [Ali et al., 2024]. The maximum GPU-access lengths were selected uniformly from $[0.1C_i, 0.7C_i]$. By [Ali et al., 2024], a task's maximum GPU-access length occurs when the number of SMs allocated to the task is small. Similar to [Ali et al., 2024], for each GPU-accessing task τ_i, we selected a value ρ_i, not exceeding the number of total SMs, that represents the maximum number of SMs the task can utilize, *i.e.,* GPU-access lengths do not increase if more SMs are allocated.

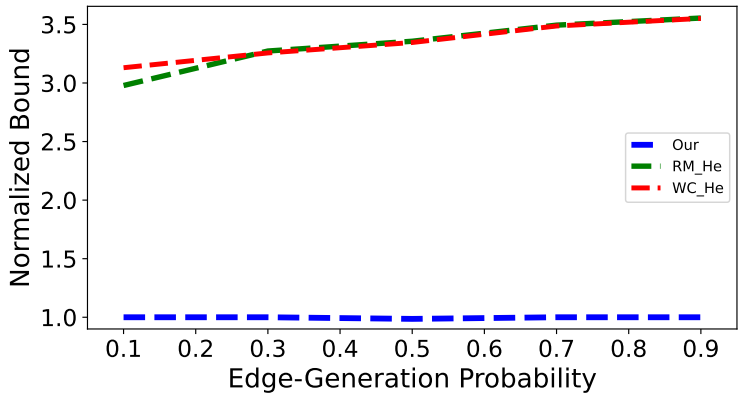
To apply our approach, we then generated *fine-grained* DAGs by splitting the GPU-accessing tasks of *coarse-grained* DAGs. Each GPU-accessing task was split into two CPU tasks and multiple GPU



(a) Normalized bound vs. edge-generation probability for *light* GPU-access ratio.



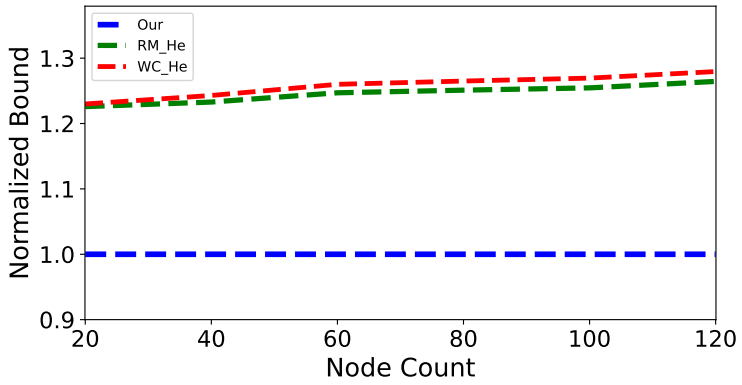
(b) Normalized bound vs. edge-generation probability for *moderate* GPU-access ratio.



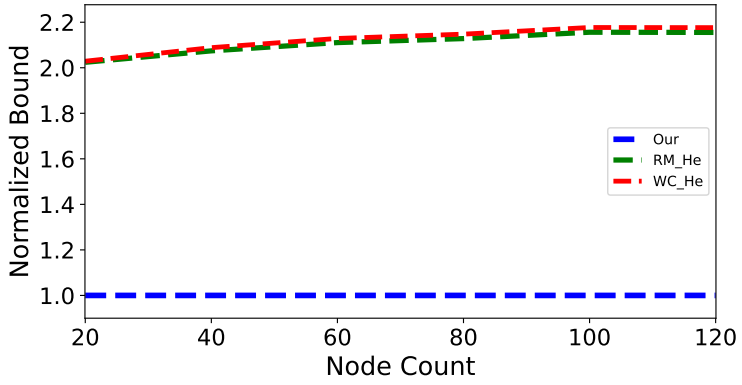
(c) Normalized bound vs. edge-generation probability for *heavy* GPU-access ratio.

Figure 7.9: Normalized bound vs. edge-generation probability.

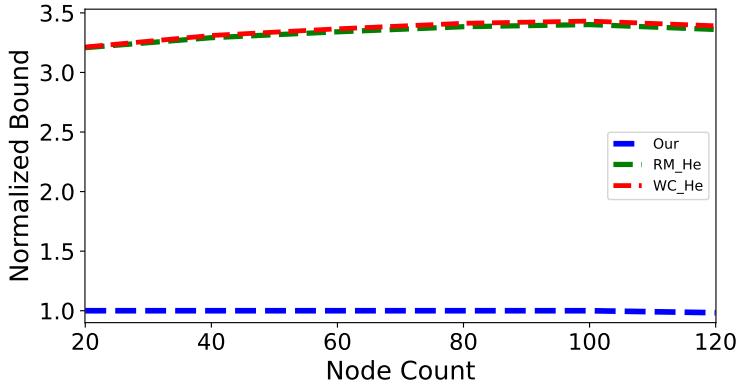
blocks. Each *block* was a gang task, for which we selected *block sizes* (*i.e.,* m_i values) randomly from $\{126, 256, 512, 1028\}$. The number of blocks was determined so that increasing the number by one required



(a) Normalized bound vs. node count for *light* GPU-access ratio.



(b) Normalized bound vs. node count for *moderate* GPU-access ratio.



(c) Normalized bound vs. node count for *heavy* GPU-access ratio.

Figure 7.10: Normalized bound vs. node count.

more the ρ_i SMs. Finally, edges were added from one CPU task to all GPU blocks and from all GPU blocks to the other CPU task. For each combination of processor count, SM count, edge probabilities, and GPU-accessing task ratios, we generated 1,000 task sets. We compared our bound (OUR) under semi-

work-conserving scheduling with RM-HE and WC-HE. Figures 7.9 and 7.10 present these three bounds *normalized* with respect to OUR.

Observation 7.2. *For small, moderate, and heavy GPU-accessing-task ratios, bounds under* RM-HE *(resp.,* WC-HE*) were, on average,* $1.24\times$*,* $2.07\times$*, and* $3.30\times$ *(resp.,* $1.25\times$*,* $2.09\times$*, and* $3.32\times$*), respectively, of those under* OUR.

For systems with many GPU-accessing tasks, OUR gave much smaller response-time bounds than RM-HE and WC-HE. Figure 7.9 shows this by plotting normalized bounds with respect to edge-generation probabilities. The bounds of RM-HE and WC-HE were larger compared to OUR for larger edge-generation probabilities. This is because of pi-blocking-related inflation of WCETs under RM-HE and WC-HE. With large edge-generation probabilities, accumulated inflation of WCETs along the longest path of the DAG became high.

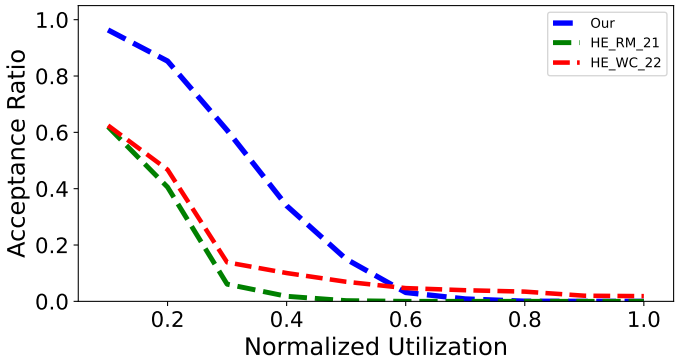
Multi-DAG systems. In this section, we used the above single DAG-generation method iteratively to generate multiple DAGs. For each processor count in $\{8, 16, 24, 32\}$, we generated task systems with coarse-grained DAGs that have *normalized utilizations*, *i.e.,* sum of all DAG utilizations over processor count, from 0.1 to 1 with a step size of 0.1. Similar to [He et al., 2021], we chose DAG G's period uniformly from $[len(G), 6 \cdot len(G)]$, where $len(G)$ is its longest-path length. For each combination of processor count, SM count, edge probabilities, and GPU-accessing task ratios, we generated 1,000 task sets. For each combination, we determined the *acceptance ratio*, which gives the percentage of task systems that were schedulable under each of OUR, RM-HE, and WC-HE. Figure 7.11 presents these acceptance ratios.

Observation 7.3. *For small, moderate, and heavy GPU-accessing-task ratios,* RM-HE *(resp.,* WC-HE*) scheduled* 14%*,* 19%*, and* 30% *(resp.,* 15%*,* 22%*, and* 48%*) of the systems compared to* OUR*, respectively.*

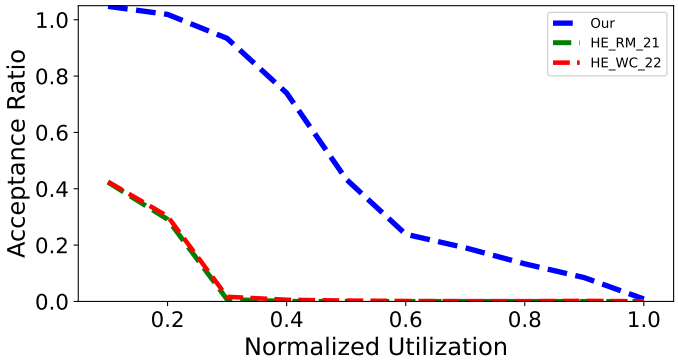
Similar to single-DAG experiments, WCET inflations caused fewer systems to be schedulable under RM-HE and WC-HE than OUR. However, as seen in Figure 7.11(a), WC-HE scheduled some systems that OUR cannot. This is because light DAGs share processors under WC-HE, while our analysis requires allocating a dedicated processor to each light DAG.

7.6.3 Case Study on Multicore+GPU

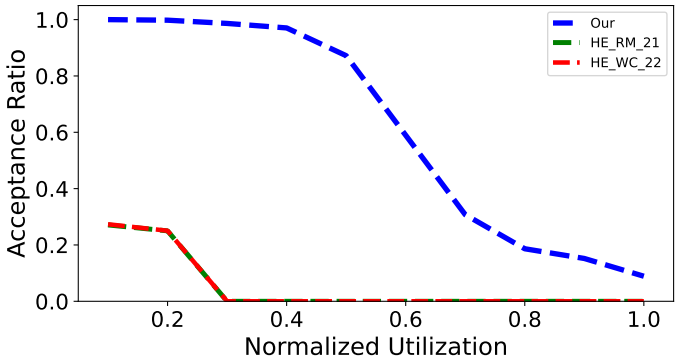
For this case study, we employed the pedestrian-detection algorithm *Histogram of Oriented Gradients (HOG)* [Dalal and Triggs, 2005]. HOG computes gradients over each frame of a video feed via a series of



(a) Acceptance ratio vs. normalized utilizations for *light* GPU-access ratio.



(b) Acceptance ratio vs. normalized utilizations for *moderate* GPU-access ratio.



(c) Acceptance ratio vs. normalized utilizations for *heavy* GPU-access ratio.

Figure 7.11: Results of multi-DAG experiments on multicore+GPU.

CUDA kernels, forming a DAG of gang tasks with sequential CPU and parallel GPU computations. These experiments were conducted on a machine running a modified version of LITMUS^{RT} [Brandenburg, 2011; Calandrino et al., 2006], a Linux-based real-time kernel. The machine had a 3.5-GHz AMD Ryzen 9 3950X 16-Core Processor and one NVIDIA RTX 6000 Ada Generation GPU.

We considered both single- and multi-DAG scenarios. In both scenarios, we ran HOG under the locking-based approach using the OMLP protocol [Brandenburg and Anderson, 2010a; Amert et al., 2021] and under federated scheduling with the default semi-work-conserving scheduler [Bakita and Anderson, 2024]. In the multi-DAG scenario, we ran four parallel HOG instances. We used `libsmctrl` [Bakita and Anderson, 2023] to partition GPU among the four HOG instances under federated scheduling. We measured response times of 1,000 DAG jobs, each processing a video frame at five image-scale levels. In the single-DAG scenario, under the locking-based approach and semi-work-conserving scheduling, the average (resp., maximum) response time was 2.6ms (resp., 8.5ms) and 2.5ms (resp., 7.1ms), respectively. Thus, there was a 16.5% reduction in maximum response time under the semi-work-conserving approach. For multiple DAGs, under the locking-based approach and semi-work-conserving scheduling, the average (resp., maximum) response time was 9.9ms (resp., 143.0ms) and 15.9ms (resp., 31.5ms), respectively.

7.7 Chapter Summary

In this chapter, we have considered the scheduling of DAGs composed of gang tasks on heterogeneous processing platforms. We presented a polynomial-time response-time bound for such DAGs under any scheduler that is either work-conserving or semi-work-conserving. We have also given an ILP formulation to allocate processors among multiple DAGs. We have demonstrated the utility of our approach through schedulability studies and a case study on a multicore+GPU platform.

Acknowledgment. The work presented in this chapter is the result of a collaboration between Shareef Ahmed and Denver Massey. Ahmed derived the response-time bounds and implemented the schedulability studies presented in Sections 7.6.1 and 7.6.2. Denver Massey performed the case study presented in Section 7.6.3.

CHAPTER 8: CONCLUSION

Scheduling and synchronization algorithms play a crucial role in ensuring the temporal correctness of resource-constrained real-time systems. For such systems, these algorithms must be designed and analyzed to utilize processing resources as efficiently as possible. Achieving high resource utilization while maintaining temporal correctness is particularly challenging in today's compute-heavy, highly parallel real-time systems. This dissertation takes a step toward addressing this challenge by providing tighter analyses of scheduling and synchronization algorithms. First, we derived tight and exact response-time bounds for pseudo-harmonic periodic tasks, both with and without precedence constraints, under GEL scheduling. Second, we established an improved pi-blocking lower bound for a class of non-FIFO GEL schedulers, showing that existing asymptotically optimal suspension-based locking protocols incur pi-blocking that exceeds the optimal bound by at most one request length. We also devised an optimal suspension-based locking protocol for FIFO scheduling. Finally, we provided response-time analyses for gang tasks, with and without precedence constraints.

The remainder of this chapter summarizes our results (Section 8.1), describes other related work conducted in parallel but beyond the scope of this dissertation (Section 8.2), and outlines directions for future work (Section 8.3).

8.1 Summary of Results

In this section, we provide a summary of the results presented in this dissertation.

Tight and exact response-time analysis. Existing response-time analyses for the G-EDF scheduling of sequential tasks are not tight. Moreover, these analyses tend to be restrictive for large systems, as the derived response-time bounds typically increase with the number of processors in highly utilized platforms. This pessimism becomes even more pronounced for graph-based tasks, where source-to-sink response-time requirements further amplify the bounds.

In Chapter 3, we presented a tight response-time bound for a class of periodic tasks, called pseudo-harmonic tasks, under GEL schedulers. The derived bound does not depend on the number of tasks or

processors, but only on task parameters. Furthermore, the bound has a closed-form expression and can be computed in linear time. Our tightness result implies that it is unlikely to obtain an empirically tighter bound without resorting to more computationally expensive response-time analysis.

Also in Chapter 3, we showed how to derive the exact response-time bound for a periodic system scheduled by GEL schedulers. Our method requires simulating the system schedule over a finite time interval and can be executed in pseudo-polynomial time for pseudo-harmonic systems. In Chapter 4, we extended these simulation-based techniques to systems with precedence constraints among tasks and self-dependencies among task instances.

Tight pi-blocking lower and upper bounds. Suspension-based locking protocols are common for dealing with mutex resources. Many suspension-based multiprocessor locking protocols provide asymptotically optimal pi-blocking bounds under JLFP schedulers. However, since the inception of the first such protocol 15 years ago [Brandenburg and Anderson, 2010a], locking protocols that achieve *truly* optimal pi-blocking bounds have remained elusive. Additionally, we presented an optimal multiprocessor locking protocol for k-exclusion sharing and a phase-fair locking protocol for reader-writer sharing that exceeds the optimal pi-blocking bound by only two request access lengths. Finally, for mutex sharing under G-EDF scheduling, we showed that some existing locking protocols actually provide pi-blocking bounds that exceed the optimal bound by only a single resource access length, thereby resolving the long-standing mystery regarding a factor of two in existing pi-blocking bounds.

Scheduling gang tasks. The scheduling and analysis of gang tasks have become increasingly important due to their relevance in systems equipped with hardware accelerators such as GPUs. Despite this significance, the scheduling of SRT gang tasks has received relatively little attention. In Chapter 6, we showed that determining the SRT-feasibility of a set of rigid gang tasks on an identical multiprocessor is NP-hard. We also demonstrated that, although G-EDF is SRT-optimal for scheduling sequential and DAG tasks, it is not for scheduling rigid gang tasks. Finally, we presented an SRT-schedulability test for rigid gang tasks under G-EDF that theoretically dominates the existing test by Dong *et al.* [Dong et al., 2021].

In Chapter 7, we presented a response-time analysis for rigid gang tasks with precedence constraints scheduled on a heterogeneous compute platform comprising multiple computational resources. This task model captures the behavior of AI-based systems that form processing graphs and are deployed on multiprocessor platforms augmented with hardware accelerators. We provided response-time analysis under both

work-conserving and semi-work-conserving schedulers. Our choice of scheduler classes is motivated by the observation that common CPU scheduling policies are typically work-conserving, whereas default GPU scheduling policies tend to be semi-work-conserving under certain restrictions. Through empirical evaluation, we demonstrated that accessing GPUs without locks can offer schedulability benefits when our response-time analysis is used.

8.2 Other Work

This section summarizes additional research contributions, beyond the scope of this dissertation, in which the author has been involved.

Semi-clustered scheduling of SRT sporadic tasks. Although existing response-time bounds for sporadic tasks under G-EDF and its variants scale with the number of processors, these bounds tend to be relatively tight when the processor count is small. Based on this observation, we proposed a *semi-clustered* scheduler named SC-EDF [Ahmed and Anderson, 2020]. Under this scheduler, tasks are partitioned into small clusters, where each cluster's size is determined by the total utilization of the tasks within it. A cluster with utilization U_c is allocated $\lfloor U_c \rfloor$ dedicated processors on which only its tasks execute. However, to ensure bounded response times, each cluster also requires additional fractional processing capacity beyond its dedicated processors. This remaining capacity is supplied by a set of globally shared processors accessible by all clusters. To provide this fractional capacity, SC-EDF creates a periodic server for each cluster whose utilization matches or exceeds the additional capacity required. These servers are scheduled using an HRT-optimal Pfair scheduler [Baruah et al., 1995].

SRT-feasibility of rp-sporadic tasks on heterogeneous multiprocessors. SRT-feasibility conditions for scheduling sporadic tasks are known for systems where tasks exhibit no intra-task parallelism, *i.e.,* when all P_i values are one, on heterogeneous multiprocessors. In [Massey et al., 2024], we extended these SRT-feasibility conditions to support sporadic tasks with arbitrary levels of parallelism under various heterogeneous multiprocessor models. To derive such conditions, we introduced a method to analytically transform a task set with arbitrary parallelism into an equivalent set of tasks without intra-task parallelism. Using these transformed task sets, we provided necessary and sufficient conditions for SRT-feasibility under both the uniform and unrelated multiprocessor models.

Non-decomposition-based optimal SRT scheduling of a DAG task. Scheduling algorithms that ensure bounded response times for DAG tasks without incurring capacity loss usually rely on decomposition-based techniques. As noted earlier, such approaches compute the response-time bound of a DAG task by summing pessimistic bounds of individual nodes along a path in the DAG. In [Ahmed and Anderson, 2025], we gave a scheduling algorithm for a DAG task that does not require decomposition-based techniques. The key idea to enable such an approach is to elevate priority of certain jobs of a DAG job by means of *priority boosting*.

Budget management. As seen in Chapters 3–7, schedulability-analysis techniques typically rely on precise WCET estimates of tasks. Unfortunately, on multicore systems, obtaining accurate WCET estimates that are not overly pessimistic via static code analysis is not viable [Wilhelm, 2020]. The only practical alternative is to use *measurement-based* timing analysis, which has been the focus of considerable recent research [Cazorla et al., 2019; Davis and Cucu-Grosjean, 2019]. However, with measurement-based analysis, one can never be certain that the true WCET of a task has been observed. As a result, schedulability analysis based on such estimates can lead to incorrect conclusions, and crucially, whether those conclusions are wrong may be unknowable. Thus, real-time systems should be equipped with runtime mechanisms to deal with *overrunning* jobs that execute for more than their WCET estimantes. One way to deal with such overrunning jobs is to assign budgets (based on WCET estimates) to tasks and enforce these budgets at runtime.

In [Tong et al., 2022], we presented budget-enforcement techniques for sporadic tasks that use mutex resources. In such systems, budget enforcement is required at both the job and CS levels; that is, each job and each CS is assigned a budget and cannot execute beyond it. Enforcing job-level budgets becomes challenging in the presence of CSs, as a job may exhaust its budget while executing within a CS. To allow safe abortion of a job upon budget exhaustion, we used the concept of a *forbidden zone* [Holman and Anderson, 2006], which prevents a job from issuing a resource request unless it has sufficient remaining budget to complete the CS. Importantly, this "sufficient" budget includes not only the execution time of the CS code itself but also the time required for lock acquisition, lock release, and budget-management overhead. We showed how to compute forbidden zones by accounting for these components under both spin-based and suspension-based locking protocols. Additionally, to enforce budgets at the CS level, we designed *abortable CSs* that linearize the computation of a shared-data-accessing CS to a single write instruction.

In [Tong et al., 2023], we presented budget-management techniques for DAG tasks. To address the challenge of job overruns at runtime due to inaccurate WCET estimates, one approach is to enforce per-node

budgets. However, if a job is aborted when it overruns, the entire DAG invocation corresponding to that job typically becomes logically incorrect and must be aborted. This can significantly magnify the effects of individual job overruns. For example, a node failure rate of just 0.1% can lead to a DAG failure rate of 36.8% (assuming independent and identically distributed probabilities) for a graph with 1,000 nodes [Amert et al., 2021]—a scale that is plausible in AI-based applications. To reduce DAG failure rates, we proposed budget reallocation techniques under server-based scheduling, as discussed in Chapter 4. Our approach allows an overrunning job to complete execution using budget from servers of its successor nodes. This is motivated by the observation that not all nodes in a DAG tend to fully consume their budgets, suggesting that unused budget from some nodes can be reallocated to support overrunning jobs. Additionally, we introduced slack-allocation policies that proactively utilize slack from underrunning jobs to help execute their successors when possible.

Simultaneous multithreading in real-time systems. Today, many multicore platforms support multiple (often two) hardware threads per core that can execute tasks concurrently—a feature known as *simultaneous multithreading* (SMT). While SMT can enhance computing capacity, it has been largely avoided in real-time systems research due to concerns about interference between tasks sharing a core. In [Osborne et al., 2020], we demonstrated for the first time how SMT can be leveraged for priority-driven preemptive scheduling of HRT systems. To safely utilize SMT, we proposed an ILP that pairs tasks with the same periods based on their "SMT-friendliness," *i.e.,* tasks are co-scheduled only when their combined execution times rarely exceed the sum of their solo execution times. Once pairing decisions are made, all tasks are partitioned and scheduled under P-EDF scheduling.

In [Bakita et al., 2021], we applied SMT to *mixed-criticality* systems, where tasks may have different *criticalities*. A task's criticality typically reflects the severity of its failure—higher-criticality tasks can cause catastrophic consequences if they fail. As a result, higher-criticality tasks have hard deadlines, while lower-criticality tasks have soft deadlines. To apply SMT in mixed-criticality systems, we determined the pairing of HRT tasks using an ILP similar to the one proposed in [Osborne et al., 2020]. For SRT tasks, static task pairing is not necessary to ensure bounded response times. Instead, we identified a set of "SMT-friendly" tasks and assigned these tasks to clusters of processors, allowing any pair within a cluster to be co-scheduled. We implemented this scheduling approach in an open-source framework called MC^2 *(mixed-criticality on multicore)* [Anderson et al., 2009; Mollison et al., 2010], which includes support for SMT on SMT-capable

multicore platforms. Additionally, features to mitigate cache and memory interference were re-implemented on SMT-capable multicore platforms to evaluate the effect of SMT under various cache allocations.

Flexible scheduling in Robot Operating System 2. The Robot Operating System 2 (ROS) is widely used in autonomous systems due to its large ecosystem and modular design. However, ROS poses challenges for real-time applications because of the implementation of the ROS *executor*. In particular, the ROS executor does not support preemption or user-specified priorities, both of which are fundamental to real-time scheduling. Prior ROS variants have individually supported preemption or prioritization of callbacks, but they impose restrictions on applications that prevent their adoption in real-world workloads. In [Liu et al., 2025], we addressed these limitations by proposing a novel executor framework, ROS$^{\text{RT}}$, which is compatible with any type of ROS application while supporting preemptive, priority-driven scheduling. Furthermore, to enable flexible EDF scheduling in ROS$^{\text{RT}}$, we implemented a custom EDF scheduler using the new Linux scheduling class `SCHED_EXT`. ROS$^{\text{RT}}$ not only enables real-time compatibility but also achieves a significant reduction in publisher-to-subscriber overhead compared to the native ROS executor.

8.3 Future Work

Finally, we conclude by discussing several directions for future work to extend the work presented in this dissertation.

Tight response-time bounds. Although this dissertation presents a tight response-time bound for pseudo-harmonic tasks under G-EDF and its variants, deriving tight bounds that apply to general periodic or sporadic task systems remains an open problem. For heterogeneous multiprocessors, this problem becomes even more challenging. In addition to scaling with the processor count, existing response-time bounds [Devi and Anderson, 2008; Erickson et al., 2014; Valente, 2016] for G-EDF scheduling on heterogeneous multiprocessors can become excessively large when the system includes a task with very low utilization. This behavior is unintuitive: adding a very low-utilization task to a system with many high-utilization tasks should not significantly increase their response times. Finally, whether G-EDF is SRT-optimal on unrelated multiprocessors also remains an open problem [Tang, 2024].

Existing variants of G-EDF that are SRT-optimal on heterogeneous multiprocessors may require frequent migrations to ensure that each job is scheduled on processors with sufficiently high speeds. In practice, the overheads associated with such migrations can be prohibitively costly. Therefore, it may be desirable to

design variants of G-EDF that reduce migration frequency, even at the cost of sacrificing SRT-optimality. Doing so would require deriving corresponding conditions for bounded response times under such schedulers.

Non-decomposition-based SRT DAG scheduling of multiple DAG tasks. Although in [Ahmed and Anderson, 2025], we gave a non-decomposition-based SRT-optimal scheduler for a single DAG, no such scheduler is known for multiple DAG tasks. Devising such a scheduler can yield significant schedulability benefits. However, analyzing multi-DAG systems becomes significantly more complex in the presence of the various features listed in Table 2.3. In particular, self-dependencies are difficult to handle, as a task may be delayed by its own prior jobs even when not all processors are busy. Moreover, how to prioritize different jobs to ensure bounded response times for all DAGs is not well understood.

Tight pi-blocking bounds. In Chapter 5, we established a lower bound of $2M - 2$ request lengths on per-request pi-blocking under a class of non-FIFO GEL schedulers. Deriving a similar lower bound for clustered scheduling (and by extension, partitioned scheduling) is more complex. This is because a locking protocol may not satisfy lock requests uniformly across all clusters; that is, more requests from one cluster may be satisfied within a given time interval compared to others. Since pi-blocking is defined based on per-cluster priorities in clustered scheduling, a lower-bound proof must account for such non-uniform execution scenarios while ensuring that each cluster has enough tasks so that c (cluster size) jobs per cluster can be pi-blocked at any given time. Although the proof becomes more intricate, we believe that a similar lower bound also applies under clustered non-FIFO GEL schedulers. Furthermore, our current lower-bound proof assumes one lock request per job. When jobs issue multiple lock requests, deriving a tight lower bound becomes even more challenging.

Gang scheduling. Compared to sequential and DAG tasks, the scheduling of gang tasks has received less attention in the real-time systems literature. This dissertation also contributes to the study of gang scheduling in various contexts. For example, work-conserving non-preemptive scheduling of gang tasks can cause a *transitive blocking* effect, where a higher-priority job may be repeatedly pi-blocked by lower-priority jobs. While recent work has addressed this issue for HRT systems [Lee et al., 2022a; Dong and Liu, 2022], no existing work considers SRT gang tasks. Moreover, semi-work-conserving schedulers, introduced in Chapter 7, may eliminate transitive blocking and thus warrant further investigation. Gang scheduling under the rp model introduces additional complexity in quantifying parallelism-induced idleness, as a job may be blocked due to the execution of one of its own prior jobs. Finally, gang tasks with precedence constraints

are highly relevant to modern systems, but it remains an open question how to handle all the complexities outlined in Table 2.3 when scheduling such tasks.

BIBLIOGRAPHY

Afshar, S., Nemati, F., and Nolte, T. (2012). Towards Resource Sharing Under Multiprocessor Semi-Partitioned Scheduling. In *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems*, pages 315–318.

Ahmed, S. and Anderson, J. (2020). A Soft-Real-Time-Optimal Semi-Clustered Scheduler with a Constant Tardiness Bound. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Ahmed, S. and Anderson, J. (2021). Tight Tardiness Bounds for Pseudo-Harmonic Tasks Under Global-EDF-Like Schedulers. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*, pages 11:1–11:24.

Ahmed, S. and Anderson, J. (2022). Exact Response-Time Bounds of Periodic DAG Tasks under Server-Based Global Scheduling. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 447–459.

Ahmed, S. and Anderson, J. (2023a). Optimal Multiprocessor Locking Protocols Under FIFO Scheduling. In *Proceedings of the 35th Euromicro Conference on Real-Time Systems*, pages 16:1–16:21.

Ahmed, S. and Anderson, J. (2023b). Soft Real-Time Gang Scheduling. In *Proceedings of the 44th IEEE Real-Time Systems Symposium*, pages 331–343.

Ahmed, S. and Anderson, J. (2024). Open Problem Resolved: The "Two" in Existing Multiprocessor PI-Blocking Bounds Is Fundamental. In *Proceedings of the 36th Euromicro Conference on Real-Time Systems*, pages 11:1–11:21.

Ahmed, S. and Anderson, J. (2025). A Soft-Real-Time Optimal Scheduler for DAG Tasks with Node-Level Self Dependencies. In *Proceedings of the 46th IEEE Real-Time Systems Symposium*. to appear.

Ahmed, S., Massey, D., and Anderson, J. (2025). Scheduling Processing Graphs of Gang Tasks on Heterogeneous Platforms. In *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 362–374.

Akesson, B., Nasri, M., Nelissen, G., Altmeyer, S., and Davis, R. I. (2022). A Comprehensive Survey of Industry Practice in Real-Time Systems. *Real-Time Systems*, 58(3):358–398.

Ali, S. W., Tong, Z., Goh, J., and Anderson, J. (2024). Predictable GPU Sharing in Component-Based Real-Time Systems. In *Proceedings of the 36th Euromicro Conference on Real-Time Systems*, pages 15:1–15:22.

Ali, W., Pellizzoni, R., and Yun, H. (2021). Virtual Gang Scheduling of Parallel Real-Time Tasks. In *Proceedings of the 25th Design, Automation and Test in Europe Conference*, pages 270–275.

Amert, T., Otterness, N., Yang, M., Anderson, J., and Smith, F. D. (2017). GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 104–115.

Amert, T., Tong, Z., Voronov, S., Bakita, J., Smith, F. D., and Anderson, J. (2021). TimeWall: Enabling Time Partitioning for Real-Time Multicore+Accelerator Platforms. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 455–468.

Amert, T., Voronov, S., and Anderson, J. (2019). OpenVX and Real-Time Certification: The Troublesome History. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 312–325.

Anderson, J., Baruah, S., and Brandenburg, B. (2009). Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*.

Anderson, J., Erickson, J. P., Devi, U., and Casses, B. N. (2014). Optimal Semi-Partitioned Scheduling in Soft Real-Time Systems. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Anderson, J. and Srinivasan, A. (2000). Early-Release Fair Scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43.

Anderson, J. and Srinivasan, A. (2004). Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks. *Journal of Computer and System Sciences*, 68(1):157–204.

Andersson, B. and Bletsas, K. (2008). Sporadic Multiprocessor Scheduling with Few Preemptions. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 243–252.

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.

Anssi, S., Kuntz, S., Gérard, S., and Terrier, F. (2013). On the Gap between Schedulability Tests and an Automotive Task Model. *Journal of Systems Architecture*, 59(6):341–350.

Baker, T. P. and Cirinei, M. (2007). Brute-Force Determination of Multiprocessor Schedulability for Sets of Sporadic Hard-Deadline Tasks. In *Proceedings of the 11th International Conference on Principles of Distributed Systems*, pages 62–75.

Bakita, J., Ahmed, S., Osborne, S. H., Tang, S., Chen, J., Smith, F. D., and Anderson, J. (2021). Simultaneous Multithreading in Mixed-Criticality Real-Time Systems. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 278–291.

Bakita, J. and Anderson, J. (2023). Hardware Compute Partitioning on NVIDIA GPUs. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 54–66.

Bakita, J. and Anderson, J. (2024). Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 294–305.

Baro, J., Boniol, F., Cordovilla, M., Noulard, E., and Pagetti, C. (2012). Off-line (Optimal) Multiprocessor Scheduling of Dependent Periodic Tasks. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1815–1820.

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 19th Design, Automation and Test in Europe Conference*, pages 1323–1328.

Baruah, S. (2015b). Federated Scheduling of Sporadic DAG Task Systems. In *Proceedings of the 2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 179–186.

Baruah, S. (2020). Scheduling DAGs When Processor Assignments Are Specified. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, page 111–116.

Baruah, S. (2021). Feasibility Analysis of Conditional DAG Tasks is co-NP^{NP}-Hard. In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, page 165–172.

Baruah, S., Bonifaci, V., and Marchetti-Spaccamela, A. (2015). The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 222–231.

Baruah, S. and Burns, A. (2006). Sustainable Scheduling Analysis. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 159–168.

Baruah, S., Cohen, N. K., Plaxton, C. G., and Varvel, D. A. (1996). Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15(6):600–625.

Baruah, S. K., 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. K., Gehrke, J., and Plaxton, C. G. (1995). Fast Scheduling of Periodic Tasks on Multiple Resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288.

Bedarkar, K., Vardishvili, M., Bozhko, S., Maida, M., and Brandenburg, B. (2022). From Intuition to Coq: A Case Study in Verified Response-Time Analysis of FIFO Scheduling. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 197–210.

Bhuiyan, A., Yang, K., Arefin, S., Saifullah, A., Guan, N., and Guo, Z. (2019). Mixed-Criticality Multicore Scheduling of Real-Time Gang Task Systems. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 469–480.

Biondi, A. and Buttazzo, G. C. (2017). Timing-Aware FPGA Partitioning for Real-Time Applications Under Dynamic Partial Reconfiguration. In *Proceedings of the 2017 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 172–179.

Blazewicz, J., Drabowski, M., and Weglarz, J. (1986). Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, C-35(5):389–393.

Bletsas, K. and Andersson, B. (2009). Preemption-Light Multiprocessor Scheduling of Sporadic Tasks with High Utilisation Bound. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 447–456.

Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A Flexible Real-Time Locking Protocol for Multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56.

Bonifaci, V. and Marchetti-Spaccamela, A. (2025). Feasibility Analysis of Recurrent DAG Tasks is PSPACE-Hard. *Theoretical Computer Science*, 1030(C):115062.

Bonifaci, V., Marchetti-Spaccamela, A., Megow, N., and Wiese, A. (2013a). Polynomial-Time Exact Schedulability Tests for Harmonic Real-Time Tasks. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 236–245.

Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S., and Wiese, A. (2013b). Feasibility Analysis in the Sporadic DAG Task Model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 225–233.

Brandenburg, B. (2011). *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill.

Brandenburg, B. (2013a). A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 292–302.

Brandenburg, B. (2013b). Improved Analysis and Evaluation of Real-Time Semaphore Protocols for P-FP Scheduling. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 141–152.

Brandenburg, B. (2014). The FMLP+: An Asymptotically Optimal Real-Time Locking Protocol for Suspension-Aware Analysis. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 61–71.

Brandenburg, B. and Anderson, J. (2009). On the Implementation of Global Real-Time Schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 214–224.

Brandenburg, B. and Anderson, J. (2010a). Optimality Results for Multiprocessor Real-Time Locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60.

Brandenburg, B. and Anderson, J. (2010b). Spin-Based Reader-Writer Synchronization for Multiprocessor Real-Time Systems. *Real-Time Systems*, 46(1):25–87.

Brandenburg, B. and Anderson, J. (2011). Real-Time Resource-Sharing under Clustered Scheduling: Mutex, Reader-Writer, and k-Exclusion Locks. In *Proceedings of the 11th International Conference on Embedded Software*, pages 69–78.

Brandenburg, B. and Anderson, J. (2014). The OMLP Family of Optimal Multiprocessor Real-Time Locking Protocols. *Design Automation for Embedded Systems*, 17(2):277–342.

Brandenburg, B., Calandrino, J. M., and Anderson, J. (2008). On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169.

Busquets-Mataix, J. V., Serrano, J. J., Ors, R., Gil, P. J., and Wellings, A. J. (1996). Using Harmonic Task-Sets to Increase the Schedulable Utilization of Cache-Based Preemptive Real-Time Systems. In *Proceedings of the 3rd International Workshop on Real-Time Computing Systems Application*, pages 195–202.

Buzzega, G. and Montangero, M. (2024). Characterizing Global Work-Conserving Scheduling Tardiness with Uniform Instances on Multiprocessors. *Real-Time Systems*, 60(4):537–569.

Buzzega, G., Nocetti, G., and Montangero, M. (2023). Characterizing G-EDF scheduling tardiness with uniform instances on multiprocessors. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, pages 45–55.

Calandrino, J. M., Leontyev, H., Block, A., Devi, U., and Anderson, J. (2006). LITMUS$^{\mathrm{RT}}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–126.

Caranddriver (2020). Electronics Account for 40 Percent of the Cost of a New Car, Online available=https://www.caranddriver.com/features/a32034437/computer-chips-in-cars/.

Cazorla, F., Kosmidis, L., Mezzetti, E., Hernandez, C., Abella, J., and Vardanega, T. (2019). Probabilistic Worst-Case Timing Analysis: Taxonomy and Comprehensive Survey. *ACM Computing Surveys*, 52(1):14:1–14:35.

Chang, S., Bi, R., Sun, J., Liu, W., Yu, Q., Deng, Q., and Gu, Z. (2022). Toward Minimum WCRT Bound for DAG Tasks Under Prioritized List Scheduling Algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):3874–3885.

Chang, S., Zhao, X., Liu, Z., and Deng, Q. (2020). Real-Time Scheduling and Analysis of Parallel Tasks on Heterogeneous Multi-cores. *Journal of Systems Architecture*, 105:101704.

Cho, H., Ravindran, B., and Jensen, E. D. (2006). An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 101–110.

Collette, S., Cucu, L., and Goossens, J. (2008). Integrating Job Parallelism in Real-Time Scheduling Theory. *Information Processing Letters*, 106(5):180–187.

Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J., and Wagner, F. (2010). Random Graph Generation for Scheduling Simulations. In *Proceedings of the 2nd EAI International Conference on Simulation Tools and Techniques*, page 60.

Cucu, L. and Goossens, J. (2006). Feasibility Intervals for Fixed-Priority Real-Time Scheduling on Uniform Multiprocessors. In *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 397–404.

Cucu-Grosjean, L. and Goossens, J. (2007). Feasibility Intervals for Multiprocessor Fixed-Priority Scheduling of Arbitrary Deadline Periodic Systems. In *Proceedings of the 11th Design, Automation and Test in Europe Conference*, pages 1635–1640.

Cucu-Grosjean, L. and Goossens, J. (2011). Exact Schedulability Tests for Real-Time Scheduling of Periodic Tasks on Unrelated Multiprocessor Platforms. *Journal of Systems Architecture*, 57(5):561–569.

Dalal, N. and Triggs, B. (2005). Histograms of Oriented Gradients for Human Detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 886–893.

Davis, R. and Cucu-Grosjean, L. (2019). A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems. *Leibniz Transactions on Embedded Systems*, 6(1):03:1–03:60.

Dertouzos, M. (1973). *Control Robotics: the Procedural Control of Physical Processes*.

Dertouzos, M. and Mok, A. K. (1989). Multiprocessor Online Scheduling of Hard-Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506.

Devi, U. and Anderson, J. (2005). Tardiness Bounds under Global EDF Scheduling on a Multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341.

Devi, U. and Anderson, J. (2008). Tardiness Bounds under Global EDF Scheduling on a Multiprocessor. *Real-Time Systems*, 38(2):133–189.

Dong, Z. and Liu, C. (2019). Analysis Techniques for Supporting Hard Real-Time Sporadic Gang Task Systems. *Real-Time Systems*, 55(3):641–666.

Dong, Z. and Liu, C. (2022). A Utilization-Based Test for Non-Preemptive Gang Tasks on Multiprocessors. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 105–117.

Dong, Z., Yang, K., Fisher, N., and Liu, C. (2021). Tardiness Bounds for Sporadic Gang Tasks Under Preemptive Global EDF Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 32(12):2867–2879.

Easwaran, A. and Andersson, B. (2009). Resource Sharing in Global Fixed-Priority Preemptive Multiprocessor Scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 377–386.

Eisenbrand, F. and Rothvoß, T. (2008). Static-Priority Real-Time Scheduling: Response Time Computation Is NP-Hard. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 397–406.

Eisenbrand, F. and Rothvoß, T. (2010). EDF-schedulability of Synchronous Periodic Task Systems is coNP-hard. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1029–1034.

Ekberg, P. (2020). Rate-Monotonic Schedulability of Implicit-Deadline Tasks is NP-hard Beyond Liu and Layland's Bound. In *Proceedings of the 41st Real-Time Systems Symposium*, pages 308–318.

Ekberg, P. and Yi, W. (2015). Uniprocessor Feasibility of Sporadic Tasks with Constrained Deadlines is Strongly coNP-Complete. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 281–286.

Ekberg, P. and Yi, W. (2017). Fixed-Priority Schedulability of Sporadic Tasks on Uniprocessors is NP-hard. In *Proceedings of the 38th Real-Time Systems Symposium*, pages 139–146.

Elliott, G. and Anderson, J. (2013). An Optimal k-Exclusion Real-Time Locking Protocol Motivated by Multi-GPU Systems. *Real-Time Systems*, 49(2):140–170.

Emberson, P., Stafford, R., and Davis, R. (2010). Techniques for the Synthesis of Multiprocessor Tasksets. In *Proceedings of the 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 6–11.

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. and Anderson, J. (2012). Fair Lateness Scheduling: Reducing Maximum Lateness in G-EDF-Like Scheduling. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 3–12.

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. (2010). Improved Tardiness Bounds for Global EDF. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 14–23.

Faggioli, D., Lipari, G., and Cucinotta, T. (2010). The Multiprocessor Bandwidth Inheritance Protocol. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 90–99.

Fisher, N. (2007). *The multiprocessor Real-Time Scheduling of General Task Systems*. PhD thesis, University of North Carolina at Chapel Hill.

Fonseca, J., Nelissen, G., Nelis, V., and Pinho, L. M. (2016). Response Time Analysis of Sporadic DAG Tasks under Partitioned Scheduling. In *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems*, pages 1–10.

Fonseca, J. C., Nélis, V., Raravi, G., and Pinho, L. M. (2015). A Multi-DAG Model for Real-Time Parallel Applications with Conditional Execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, page 1925–1932.

Fonseca, J. C., Nelissen, G., and Nélis, V. (2019). Schedulability Analysis of DAG Tasks with Arbitrary Deadlines under Global Fixed-Priority Scheduling. *Real-Time Systems*, 55(2):387–432.

Forget, J., Boniol, F., Grolleau, E., Lesens, D., and Pagetti, C. (2010). Scheduling Dependent Periodic Tasks without Synchronization Mechanisms. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 301–310.

Fu, Y., Kottenstette, N., Chen, Y., Lu, C., Koutsoukos, X. D., and Wang, H. (2010). Feedback Thermal Control for Real-time Systems. In *Proceedings of 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 111–120.

Funk, S. (2010). LRE-TL: An Optimal Multiprocessor Algorithm for Sporadic Task Sets with Unconstrained Deadlines. *Real-Time Systems*, 46(3):332–359.

Funk, S. and Nadadur, V. (2009). LRE-TL: An Optimal Multiprocessing Scheduling Algorithm for Sporadic Task Sets. In *Proceedings of the 17th International Conferenece of Real-Time and Network Systems*, pages 159–168.

Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

Geeraerts, G., Goossens, J., and Lindström, M. (2013). Multiprocessor Schedulability of Arbitrary-Deadline Sporadic Tasks: Complexity and Antichain Algorithm. *Real-Time Systems*, 49:171–218.

Gohari, P., Voeten, J., and Nasri, M. (2024). Reachability-Based Response-Time Analysis of Preemptive Tasks Under Global Scheduling. In *Proceedings of the 36th Euromicro Conference on Real-Time Systems*, pages 3:1–3:24.

Goossens, J. and Berten, V. (2010). Gang FTP Scheduling of Periodic and Parallel Rigid Real-Time Tasks. *CoRR*.

Goossens, J. and Devillers, R. (1997). The Non-Optimality of the Monotonic Priority Assignments for Hard Real-Time Offset Free Systems. *Real-Time Systems*, 13(2):107–126.

Goossens, J. and Devillers, R. (1999). Feasibility Intervals for the Deadline Driven Scheduler with Arbitrary Deadlines. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 54–61.

Goossens, J., Grolleau, E., and Cucu-Grosjean, L. (2016). Periodicity of Real-Time Schedules for Dependent Periodic Tasks on Identical Multiprocessor Platforms. *Real-Time Systems*, 52(6):808–832.

Goossens, J. and Masson, D. (2022). Simulation Intervals for Uniprocessor Real-Time Schedulers with Preemption Delay. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pages 36–45.

Goossens, J. and Masson, D. (2024). Robust Schedulability Tests for Fixed Job Priorities: Addressing Context Switch Costs with Non-Resumable Delays. In *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, pages 290–301.

Goossens, J. and Richard, P. (2016). Optimal Scheduling of Periodic Gang Tasks. *Leibniz Transactions on Embedded Systems*, 3(1):04:1–04:18.

Graham, R. L. (1969). Bounds on Multiprocessing Timing Anomalies. *SIAM Jounral of Applied Mathematics*, 17(2):416–429.

Grolleau, E. and Choquet-Geniet, A. (2002). Off-Line Computation of Real-Time Schedules Using Petri Nets. *Discrete Event Dynamic Systems*, 12(3):311–333.

Grolleau, E., Goossens, J., and Cucu-Grosjean, L. (2013). On the Periodic Behavior of Real-Time Schedulers on Identical Multiprocessor Platforms. *CoRR*, abs/1305.3849.

Guan, F., Peng, L., and Qiao, J. (2022). A Fluid Scheduling Algorithm for DAG Tasks with Constrained or Arbitrary Deadlines. *IEEE Transactions on Computers*, 71(8):1860–1873.

Guan, F., Peng, L., and Qiao, J. (2023). A New Federated Scheduling Algorithm for Arbitrary-Deadline DAG Tasks. *IEEE Transactions on Computers*, 72(8):2264–2277.

Guan, F., Qiao, J., and Han, Y. (2021). DAG-Fluid: A Real-Time Scheduling Algorithm for DAGs. *IEEE Transactions on Computers*, 70(3):471–482.

Guan, N., Gu, Z., Deng, Q., Gao, S., and Yu, G. (2007). Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. In *Proceedings of the Software Technologies for Embedded and Ubiquitous Systems*, pages 263–272.

Hamdaoui, M. and Ramanathan, P. (1995). A Dynamic Priority Assignment Technique for Streams with (m, k)-Firm Deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451.

Han, C.-C. and Tyan, H.-Y. (1997). A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 36–45.

Han, M., Guan, N., Sun, J., He, Q., Deng, Q., and Liu, W. (2019). Response Time Bounds for Typed DAG Parallel Tasks on Heterogeneous Multi-Cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2567–2581.

He, Q., Guan, N., Lv, M., Jiang, X., and Chang, W. (2022). Bounding the Response Time of DAG Tasks Using Long Paths. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 474–486.

He, Q., Jiang, X., Guan, N., and Guo, Z. (2019). Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295.

He, Q., Lv, M., and Guan, N. (2021). Response Time Bounds for DAG Tasks with Arbitrary Intra-Task Priority Assignment. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*, volume 196, pages 8:1–8:21.

He, Q., Sun, J., Guan, N., Lv, M., and Sun, Z. (2023a). Real-Time Scheduling of Conditional DAG Tasks With Intra-Task Priority Assignment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(10):3196–3209.

He, Q., Sun, Y., Lv, M., and Liu, W. (2023b). Efficient Response Time Bound for Typed DAG Tasks. In *Proceedings of the 29th IEEE International Conference on Embedded and Real-Time Computing Systems*, pages 226–231.

Hobbs, C., Tong, Z., and Anderson, J. (2019). Optimal Soft Real-Time Semi-Partitioned Scheduling Made Simple (and Dynamic). In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 112–122.

Hohmuth, M. and Härtig, H. (2001). Pragmatic Nonblocking Synchronization for Real-Time Systems. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, page 217–230.

Holman, P. and Anderson, J. (2006). Locking Under Pfair Scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174.

Hong, K. and Leung, J.-T. (1988). On-line Scheduling of Real-Time Tasks. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 244–250.

Jaffe, J. M. (1980). Bounds on the Scheduling of Typed Task Systems. *SIAM Journal of Computing*, 9(3):541–551.

Jain, R., Hughes, C. J., and Adve, S. V. (2002). Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 134–145.

Jiang, X., Guan, N., Liang, H., Tang, Y., Qiao, L., and Yi, W. (2021). Virtually-Federated Scheduling of Parallel Real-Time Tasks. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 482–494.

Jiang, X., Guan, N., Long, X., Tang, Y., and He, Q. (2020). Real-Time Scheduling of Parallel Tasks with Tight Deadlines. *Journal of Systems Architecture*, 108:101742.

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.

Jiang, X., Long, X., Yang, T., and Deng, Q. (2018). On the Soft Real-Time Scheduling of Parallel Tasks on Multiprocessors. In *Embedded Systems Technology*, pages 65–77.

Kato, S. and Ishikawa, Y. (2009). Gang EDF Scheduling of Parallel Task Systems. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 459–468.

Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y., and Azumi, T. (2018). Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 287–296.

Kenna, C. J., Herman, J. L., Brandenburg, B. B., Mills, A. F., and Anderson, J. (2011). Soft Real-Time on Multiprocessors: Are Analysis-Based Schedulers Really Worth It? In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 93–103.

Koren, G. and Shasha, D. (1995). Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 110–117.

Kramer, S., Ziegenbein, D., and Hamann, A. (2015). Real World Automotive Benchmarks for Free. In *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*.

Kubale, M. (1987). The Complexity of Scheduling Independent Two-Processor Tasks on Dedicated Processors. *Information Processing Letters*, 24(3):141–147.

Lee, S., Guan, N., and Lee, J. (2011). Gang Fixed Priority Scheduling of Periodic Moldable Real-Time Tasks. In *Proceedings of the 4th Junior Researcher Workshop on Real-Time Computing*, pages 132–144.

Lee, S., Guan, N., and Lee, J. (2022a). Design and Timing Guarantee for Non-Preemptive Gang Scheduling. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 132–144.

Lee, S., Lee, S., and Lee, J. (2022b). Response Time Analysis for Real-Time Global Gang Scheduling. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 92–104.

Leoncini, M., Montangero, M., and Valente, P. (2019). A Parallel Branch-and-Bound Algorithm to Compute a Tighter Tardiness Bound for Preemptive Global EDF. *Real-Time Systems*, 55(2):349–386.

Leontyev, H. and Anderson, J. (2007). 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. (2010). Generalized Tardiness Bounds for Global Multiprocessor Scheduling. *Real-Time Systems*, 44(1-3):26–71.

Leung, J. Y. and Merrill, M. L. (1980). A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118.

Leung, J. Y. and Whitehead, J. (1982). On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4):237–250.

Levin, G., Funk, S., Sadowski, C., Pye, I., and Brandt, S. (2010). DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 3–13.

Li, H., Sweeney, J., Ramamritham, K., Grupen, R. A., and Shenoy, P. J. (2003). Real-Time Support for Mobile Robotics. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 10–18.

Li, J., Agrawal, K., Gill, C., and Lu, C. (2014). Federated Scheduling for Stochastic Parallel Real-Time Tasks. In *Proceedings of the 31st IEEE International Conference on Embedded and Real-Time Computing Systems and Application*, pages 1–10.

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, X., Ma, Y., Chen, Y., Sun, J., Chang, W., Guan, N., Chen, L., and Deng, Q. (2024). Priority Optimization for Autonomous Driving Systems to Meet End-to-End Latency Constraints. In *Proceedings of the 45th IEEE Real-Time Systems Symposium*, pages 402–414.

Lin, C., Shi, J., Ueter, N., Günzel, M., Reineke, J., and Chen, J. (2023). Type-Aware Federated Scheduling for Typed DAG Tasks on Heterogeneous Multicore Platforms. *IEEE Transactions on Computers*, 72(5):1286–1300.

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 Anderson, J. (2011). Supporting Graph-Based Real-Time Applications in Distributed Systems. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 143–152.

Liu, C. L. and Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61.

Liu, S., Wagle, R., Ahmed, S., Tong, Z., and Anderson, J. (2025). ROS$^{\text{RT}}$: Enabling Flexible Scheduling in ROS 2. In *Proceedings of the 46th IEEE Real-Time Systems Symposium*. to appear.

Massey, D., Ahmed, S., and Anderson, J. (2024). On the Feasibility of Sporadic Tasks with Restricted Parallelism on Heterogeneous Multiprocessors. In *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, pages 105–116.

Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., and Buttazzo, G. C. (2015). Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 211–221.

Mollison, M., Erickson, J., Anderson, J., Baruah, S., and Scoredos, J. (2010). Mixed criticality real-time scheduling for multicore systems. In *ICESS*, pages 1864–1871.

Nasri, M. and Brandenburg, B. (2017). An Exact and Sustainable Analysis of Non-Preemptive Scheduling. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 12–23.

Nasri, M., Nelissen, G., and Brandenburg, B. (2018). A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, pages 9:1–9:23.

Nasri, M., Nelissen, G., and Brandenburg, B. B. (2019). Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 21:1–21:23.

Nélis, V., Yomsi, P. M., and Goossens, J. (2013). Feasibility Intervals for Homogeneous Multicores, Asynchronous Periodic Tasks, and FJP Schedulers. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 277–286.

Nelissen, G., Berten, V., Goossens, J., and Milojevic, D. (2011). Reducing Preemptions and Migrations in Real-Time Multiprocessor Scheduling Algorithms by Releasing the Fairness. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 15–24.

Nelissen, G., i Igual, J. M., and Nasri, M. (2022). Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks. In *Proceedings of the 34th Euromicro Conference on Real-Time Systems*, pages 12:1–12:22.

Nelissen, G., Su, H., Guo, Y., Zhu, D., Nelis, V., and Goossens, J. (2014). An Optimal Boundary Fair Scheduling. *Real-Time Systems*, 50:456–508.

Osborne, S. H., Ahmed, S., Nandi, S., and Anderson, J. (2020). Exploiting Simultaneous Multithreading in Priority-Driven Hard Real-Time Systems. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Ousterhout, J. K. (1982). Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30.

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 30th International Conference on Real-Time Networks and System*, pages 205–214.

Patel, P., Baek, I., Kim, H., and Rajkumar, R. (2018). Analytical Enhancements and Practical Insights for MPCP with Self-Suspensions. In *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 177–189.

Pathan, R., Voudouris, P., and Stenström, P. (2018). Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):915–928.

Pinedo, M. L. (2008). *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition.

Prisaznuk, P. J. (2008). Arinc 653 role in integrated modular avionics (ima). In *Proceedings of the 27th IEEE/AIAA Digital Avionics Systems Conference*, pages 1.E.5–1–1.E.5–10.

Purdom, P. (1970). A Transitive Closure Algorithm. *BIT*, 10:76–94.

Qamhieh, M., Fauberteau, F., George, L., and Midonnet, S. (2013). Global EDF Scheduling of Directed Acyclic Graphs on Multiprocessor Systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 287–296.

Qamhieh, M., George, L., and Midonnet, S. (2014). A Stretching Algorithm for Parallel Real-time DAG Tasks on Multiprocessor Systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, page 13–22.

Rajkumar, R. (1990). Real-Time Synchronization Protocols for Shared Memory Multiprocessors. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 116–123.

Rajkumar, R. (1991). *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers.

Rajkumar, R., Sha, L., and Lehoczky, J. (1988). Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 259–269.

Regnier, P., Lima, G., Massa, E., Levin, G., and Brandt, S. A. (2011). RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 104–115.

Richard, P., Goossens, J., and Kato, S. (2017). Comments on "Gang EDF Schedulability Analysis". *CoRR*, abs/1705.05798.

Rispo, V., Aromolo, F., Casini, D., and Biondi, A. (2024). Response-Time Analysis of Bundled Gang Tasks Under Partitioned FP Scheduling. *IEEE Transactions on Computers*, 73(11):2534–2547.

Saifullah, A., Agrawal, K., Lu, C., and Gill, C. (2011). Multi-core Real-Time Scheduling for Generalized Parallel Task Models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 217–226.

Saifullah, A., Ferry, D., Li, J., Agrawal, K., Lu, C., and Gill, C. D. (2014). Parallel Real-Time Scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252.

Serrano, M. A. and Quiñones, E. (2018). Response-Time Aanalysis of DAG Tasks Supporting Heterogeneous Computing. In *Proceedings of the 55th Design and Automation Conference*, pages 125:1–125:6.

Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority Inheritance Protocols: An Approach to Real-Time System Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185.

Shi, J., Günzel, M., Ueter, N., der Bruggen, G. v., and Chen, J.-J. (2024). DAG Scheduling with Execution Groups. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 149–160.

Shih, C., Gopalakrishnan, S., Ganti, P., Caccamo, M., and Sha, L. (2003). Scheduling Real-Time Dwells Using Tasks with Synthetic Periods. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 210–219.

Stoica, I., Abdel-Wahab, H. M., Jeffay, K., Baruah, S., Gehrke, J., and Plaxton, C. G. (1996). A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299.

Sun, B., Kloda, T., and Caccamo, M. (2024a). Strict Partitioning for Sporadic Rigid Gang Tasks. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 252–264.

Sun, B., Kloda, T., Wu, C., and Caccamo, M. (2024b). Partitioned Scheduling and Parallelism Assignment for Real-Time DNN Inference Tasks on Multi-TPU. In *Proceedings of the 61st Design and Automation Conference*, pages 333:1–333:6.

Sun, J., Duan, K., Li, X., Guan, N., Guo, Z., Deng, Q., and Tan, G. (2023). Real-Time Scheduling of Autonomous Driving System with Guaranteed Timing Correctness. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 185–197.

Sun, J., Guan, N., Sun, J., Zhang, X., Chi, Y., and Li, F. (2021). Algorithms for Computing the WCRT Bound of OpenMP Task Systems With Conditional Branches. *IEEE Transaction of Computers*, 70(1):57–71.

Sun, J., Li, F., Guan, N., Zhu, W., Xiang, M., Guo, Z., and Yi, W. (2020). On Computing Exact WCRT for DAG Tasks. In *In Proceedings of the 57th ACM/IEEE Design Automation Conference*, pages 1–6.

Sun, Y. and Lipari, G. (2016). A Pre-Order Relation for Exact Schedulability Test of Sporadic Tasks on Multiprocessor Global Fixed-Priority Scheduling. *Real-Time Systems*, 52:323–355.

Tang, S. (2024). *Extending Soft Real-Time Analysis for Heterogeneous Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, USA.

Tang, S. and Anderson, J. (2020). Towards Practical Multiprocessor EDF with Affinities. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, pages 89–101.

Tang, S., Voronov, S., and Anderson, J. (2019). GEDF Tardiness: Open Problems Involving Uniform Multiprocessors and Affinity Masks Resolved. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 13:1–13:21.

Tang, S., Voronov, S., and Anderson, J. (2021). Extending EDF for Soft Real-Time Scheduling on Unrelated Multiprocessors. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 253–265.

The Linux Kernel Organization (2024). *RT-mutex Subsystem with PI support*. The Linux Kernel Organization. `https://docs.kernel.org/locking/rt-mutex.html`.

Tong, Z., Ahmed, S., and Anderson, J. (2022). Overrun-Resilient Multiprocessor Real-Time Locking. In *Proceedings of the 34th Euromicro Conference on Real-Time Systems*, pages 10:1–10:25.

Tong, Z., Ahmed, S., and Anderson, J. (2023). Holistically Budgeting Processing Graphs. In *Proceedings of the 44th IEEE Real-Time Systems Symposium*, pages 27–39.

Tong, Z., Ali, S. W., and Anderson, J. (2025). Asymptotically Optimal Multiprocessor Real-Time Locking for non-JLFP Scheduling. In *31st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 1–12.

Ueter, N., Günzel, M., von der Brüggen, G., and Chen, J. (2021). Hard Real-Time Stationary Gang-Scheduling. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*, pages 10:1–10:19.

Ueter, N., Günzel, M., von der Brüggen, G., and Chen, J. (2023). Parallel Path Progression DAG Scheduling. *IEEE Transaction on Computers*, 72(10):3002–3016.

Ueter, N., von der Brüggen, G., Chen, J., Li, J., and Agrawal, K. (2018). Reservation-Based Federated Scheduling for Parallel Real-Time Tasks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 482–494.

Ullman, J. (1975). NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10(3):384–393.

Valente, P. (2016). Using a Lag-Balance Property to Tighten Tardiness Bounds for Global EDF. *Real-Time Systems*, 52(4):486–561.

Verucchi, M., Theile, M., Caccamo, M., and Bertogna, M. (2020). Latency-Aware Generation of Single-Rate DAGs from Multi-Rate Task Sets. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 226–238.

Voronov, S., Anderson, J., and Yang, K. (2021a). Tardiness Bounds for Fixed-Priority Global Scheduling without Intra-Task Precedence Constraints. *Real-Time Systems*, 57(1-2):4–54.

Voronov, S., Tang, S., Amert, T., and Anderson, J. (2021b). AI Meets Real-Time: Addressing Real-World Complexities in Graph Response-Time Analysis. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 82–96.

Wang, K., Jiang, X., Guan, N., Liu, D., Liu, W., and Deng, Q. (2019). Real-Time Scheduling of DAG Tasks with Arbitrary Deadlines. *ACM Transactions on Design Automation of Electronic Systems*, 24(6):66:1–66:22.

Wang, Y., Liu, C., Wong, D., and Kim, H. (2024). GCAPS: GPU Context-Aware Preemptive Priority-Based Scheduling for Real-Time Tasks. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 14:1–14:25.

Ward, B., Elliott, G., and Anderson, J. (2012). Replica-Request Priority Donation: A Real-Time Progress Mechanism for Global Locking Protocols. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 280–289.

Wasly, S. and Pellizzoni, R. (2019). Bundled Scheduling of Parallel Real-Time Tasks. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 130–142.

West, R. and Poellabauer, C. (2000). Analysis of a Window-Constrained Scheduler for Real-Time and Best-Effort Packet Streams. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 239–248.

Wilhelm, R. (2020). Real Time Spent on Real Time (invited talk). In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, pages 1–2.

Yang, K. and Anderson, J. (2015a). 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. (2015b). 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. (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., 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., Amert, T., Yang, K., Otterness, N., Anderson, J. H., Smith, F. D., and Wang, S. (2018). Making OpenVX Really "Real Time". In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 80–93.

Yang, M., Wieder, A., and Brandenburg, B. (2015). Global Real-Time Semaphore Protocols: A Survey, Unified Analysis, and Comparison. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*, pages 1–12.

Zhao, S., Dai, X., Bate, I., Burns, A., and Chang, W. (2020). DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*, pages 128–140.

Zhu, D., Mossé, D., and Melhem, R. G. (2003). Multiple-Resource Periodic Scheduling Problem: How Much Fairness is Necessary? In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 142–151.