EFFICIENT DESIGN, ANALYSIS, AND IMPLEMENTATION OF COMPLEX MULTIPROCESSOR REAL-TIME SYSTEMS

Cong Liu

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
2013

ABSTRACT

CONG LIU: Efficient Design, Analysis, and Implementation of Complex Multiprocessor Real-Time Systems
(Under the direction of Prof. James H. Anderson)

The advent of multicore technologies is a fundamental development that is impacting software design processes across a wide range of application domains, including an important category of such applications, namely, those that have real-time constraints. This development has led to much recent work on multicore-oriented resource management frameworks for real-time applications. Unfortunately, most of this work focuses on simple task models where complex but practical runtime behaviors among tasks do not arise. In practice, however, many factors such as programming methodologies, interactions with external devices, and resource sharing often result in complex runtime behaviors that can negatively impact timing correctness. The goal of this dissertation is to support such more realistic and complex applications in multicore-based real-time systems. The thesis of this dissertation is:

Capacity loss (i.e., over provisioning) can be significantly reduced on multiprocessors while providing soft and hard real-time guarantees for real-time applications that exhibit complex runtime behaviors such as self-suspensions, graph-based precedence constraints, non-preemptive sections, and parallel execution segments by designing new real-time scheduling algorithms and developing new schedulability tests.

The above thesis is established by developing new multiprocessor scheduling algorithms and schedulability tests that are sufficient to provide real-time guarantees for task systems containing each of the above mentioned complex runtime behaviors individually and in combination.

First, we present several efficient multiprocessor schedulability tests for both soft and hard real-time sporadic self-suspending task systems. For the past 20 years, the unsolved problem of supporting real-time systems with suspensions has impeded research progress on many related research topics such as analyzing and implementing I/O-intensive applications in multiprocessor systems. The impact of this work is demonstrated by the fact that it provides a first set of practically efficient solutions that can fundamentally solve this problem.

To better handle graph-based precedence constraints, we propose a scheduling algorithm that can achieve no capacity loss for scheduling general task graphs on a multiprocessor while providing soft real-time correctness guarantees. We also extend this result to support task graphs in a distributed system containing multiple multiprocessor-based clusters. The impact of this work is demonstrated by the fact that we closed a problem that stood open for 12 years. By achieving no capacity loss, our research results can provide a set of analytically correct and practically efficient methodologies to designers of real-time systems that execute task graphs, such as signal processing and multimedia application systems.

Furthermore, besides handling runtime behaviors such as self-suspensions and graph-based precedence constraints independently, we also investigate how to support sophisticated real-time task systems containing mixed types of complex runtime behaviors. Specifically, we derive a sufficient schedulability test for soft real-time sporadic task systems in which non-preemptive sections, self-suspensions, and graph-based precedence constraints co-exist. We present a method for transforming such a sophisticated task system into a simpler sporadic task system with only self-suspensions. The transformation allows maximum response-time bounds derived for the transformed sporadic self-suspending task system to be applied to the original task system.

Finally, we present schedulability analysis for sporadic parallel task systems. The proposed analysis shows that such systems can be efficiently supported on multiprocessors with bounded response times. In particular, on a two-processor platform, no capacity loss results for any parallel task system. Despite this special case, on a platform with more than two processors, capacity loss is fundamental. By observing that capacity loss can be reduced by restructuring tasks to reduce intra-task parallelism, we propose optimization techniques that can be applied to determine such a restructuring.

ACKNOWLEDGEMENTS

My dissertation and graduation as a Ph.D would not have been possible without the help of many people.

I am deeply grateful to my advisor, Jim Anderson, for educating, guiding, and helping me over the past five years. First of all, I am indebted to Jim for supporting me as an RA and working with me when I first came to UNC. Ever since, it has been a great pleasure working with Jim and learning from him. His thoroughness, sharpness of intellect, genuine care and concern for his students are admirable. I also greatly appreciate Jim for many of his impromptu discussions over half-baked ideas and fresh insights. I would like to thank Jim in particular for being patient with some of my sloppy writing, getting those fixed, and teaching me to write in the process. For every paper Jim writes with his students, he always spends a great amount of time providing careful and prompt feedback on drafts. That is perhaps the reason why papers submitted by his students to various conferences and journals consistently get high review scores in the category of presentation. Last but not the least, I want to thank Jim for pointing out and reminding me my weakness in doing research: always having a tendency to go too fast and publish more papers. Jim, I will always keep your counsel in mind. In my upcoming academic career as an independent researcher, I promise, I will not let you down!

I would also like to thank my dissertation committee: Sanjoy Baruah, Kevin Jeffay, Ketan Mayer-Patel, Jasleen Kaur, and Steve Goddard. I learned quite a lot from Sanjoy on how to be a good presenter, and I am indebted to Steve for his valuable comments on my research work, particularly my work on PGM graph scheduling. I want to thank Kevin and Don Smith for performing mock faculty interviews with me. They reposed a lot of confidence (which is very much needed) in me during that tough period of time.

I also owe much to many colleagues in the real-time systems group: Andrea Bastoni, Aaron Block, Bjorn Brandenburg, John Calandrino, Ben Casses, Bipasa Chattopadhyay, Glenn Elliott, Jeremy Erickson, Zhishan Guo, Jonathan Herman, Haohan Li, Chris Kenna, Alex Mills, Mac Mollison, and Bryan Ward. I especially want to thank Mac and Bryan for giving valuable suggestions on my job talk, and Hennadiy and Alex for sharing their insights on my research. I also want to thank Zhishan for driving me to the airport many times during the past few months.

I would like to take this opportunity to extend my thanks to the administrative and technical staff of the Computer Science Department. Special thanks go to Janet Jones, Dawn Andres, Jodie Turnbull, and Tim Quigg for their help and kindness to me over the past five years.

I would also like to thank people at places where I did my two summer internships in 2010 and 20011: IBM research T. J. Watson center and IBM research Austin Laboratory. My collaborators Jian Li, Wei Huang, and Even Speight at IBM research Austin lab deserve a large amount of credit for working with me on interesting projects over the summer, 2011. I hope I will have collaboration opportunities with you again in the near future.

Foremost, I thank my parents for their unwavering support and utmost care. Mom, although I have never had the courage to say this to you, you are the greatest mother in this universe, and I love you! I promise some day soon I will make you proud!

Finally, I want to thank QianWen, for all her love, for her patience, for her positive, optimistic, and happy attitudes, for her enormous support during the dark times of my life. Without you, I would not be able to finish this dissertation and find a job. Although I feel fortunate enough, I would be more pleased if I had met you much earlier in my life! I love you so much and I will be loving you all my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

DAG	Directed Acyclic Graph
DBF	Demand Bound Function
DM	Deadline Monotonic
EDF	Earliest-Deadline-First
FIFO	First-In-First-Out
GEDF	Global Earliest-Deadline-First
GEPPF	Global Earliest-Priority-Point-First
GFIFO	Global First-In-First-Out
GPU	Graphics Processing Unit
GSA	Generic Scheduling Algorithm
GTFP	Global Task-level Fixed-Priority
HRT	Hard Real-Time
I/O	Input/Output
ILP	Integer Linear Programming
JDP	Job-level Dynamic-Priority
JFP	Job-level Fixed-Priority
LASM	Logical Application Stream Model
LHS	Left Hand Side
LLF	Least-Laxity-First
LP	Linear Programming
MAD	Monotonic Absolute Deadline
PDM	Partitioned Deadline Monotonic
PGM	Processing Graph Method
PS	Processor Sharing (schedule)
RB	Rate-Based
RBE	Rate-Based Execution
RHS	Right Hand Side
RM	Rate Monotonic

RTA	Response-Time Analysis
SDF	Synchronous DataFlow
SMP	Symmetric (shared-memory) MultiProcessor
SRPTF	Shortest-Remaining-Processing-Time-First
SRT	Soft Real-Time
S.W.A.P.	Size, Weight, and Power
TFP	Task-level Fixed-Priority
WCET	Worst-Case Execution Time

CHAPTER 1

Introduction

Given the need to achieve higher performance without driving up power consumption and heat, most chip manufacturers have shifted to multicore architectures. Current in-market systems commonly contain chips with four, ten, and even 32 cores. Per-chip core counts are expected to further increase significantly in the near future. Indeed, Intel and Tilera have both recently released chips with 80 and 100 cores [33, 47].

With the growing prevalence of multicore platforms, software design processes across a wide range of application domains are being impacted. One important category of such applications is real-time applications that must satisfy *temporal constraints* in order to be deemed correct; examples include stock trading, avionics, real-time scoring of bank transactions, live video processing, real-time sensing, etc. Such applications range from desktop-level systems to systems that require significant computational capacity. As an example of the latter, systems processing time-sensitive business transactions have been implemented by Azul Systems [94] on top of the highly-parallel Vega3 platform, which consists of up to 864 cores.

These developments have spurred much recent research on multiprocessor real-time systems. Unfortunately, to a large extent, existing research has been limited to systems supporting simple tasks that do not exhibit complex runtime behaviors; this limits the practical applicability of such research. In practice, real-time applications often exhibit rather complex runtime behaviors besides performing computation on a CPU. For example, an application might be suspended by the operating system while waiting for some interaction to occur, such as interactions with humans or external devices. Also, many applications such as multimedia and signal processing applications are implemented using processing graph implementation methodologies [51], where data communications or logical dependencies exist among stages of such applications. Other complex runtime behaviors include those due to resource sharing and the use of parallel programming paradigms.

Recently, this limitation of existing research has become even more evident and significant, due to the emerging embedded system design trend towards building complex cyber-physical systems (CPSs). Such systems often integrate and coordinate multiple networked computational and physical elements. Examples include self-driving vehicles, high-confidence healthcare devices, smart power grids, and autonomous military combat systems. Different types of runtime behaviors often arise in CPSs. For instance, many CPSs such as self-driven vehicles have multiple dependent network-connected components, each of which implements a specific functionality. Moreover, CPSs are expected to actively interact with the environment, human beings, and other external devices. These properties often result in complex runtime behaviors. For example, consider two common tasks in driving an autonomous vehicle: environmental sensing and navigation control. Data/logical dependencies exist between these two tasks: the navigation control task can execute only after receiving needed data from the environmental sensing task.

Unfortunately, inherent limitations and assumptions existing in today's theoretical foundations for real-time systems (i.e., the inability to efficiently handle complex runtime behaviors) cause many such systems to be built and supported in an inefficient way. In practice, these runtime behaviors are currently dealt with by over-provisioning systems, which is an economically wasteful practice. The goal of this dissertation is thus aimed at bridging this gap between practice and theory. Specifically, the goal of the research discussed herein is to enable multiprocessor real-time systems that support real-world applications with common types of complex runtime behaviors to be efficiently designed and built. To further motivate this research, we provide an introduction to real-time systems below. This is followed by an overview of this dissertation.

1.1 What is a Real-Time System?

Different from non-real-time systems, the distinguishing characteristic of a *real-time system* is the inclusion of temporal constraints in its specification. That is, the correctness of a real-time system depends not only on the correctness of the logical results produced by the system, but also on the time at which such results are produced. In other words, a real-time system is required to execute applications and produce correct results within specific time frames. Real-time systems are pervasive often without being noticed. Such systems range from portable devices such as heart rate monitors, to large stationary installations like traffic lights, to systems controlling nuclear power plants.

Temporal constraints of activities in real-time systems are often specified as *deadlines*. The key requirement for building a real-time system is to guarantee that all temporal constraints are satisfied. In other words, programs must complete their execution by specified deadlines. For instance, for an anti-lock braking system, if the driver presses the brake pedal, the car must respond within a specific time frame (e.g., within ten milliseconds). Since many real-time systems are safety-critical, designers also need to guarantee that all required temporal constraints can be satisfied at runtime in a predictable manner. This notion of *predicatability* as it pertains to temporal correctness defines another important characteristic of real-time systems and is the subject of concentration of this dissertation. Predicability can be ensured by proving at design time that all temporal constraints are *always* met subject to any permitted runtime scenario.

Guaranteeing the temporal correctness of a real-time system is typically achieved by (*i*) formally modeling the system, (*ii*) selecting an appropriate real-time scheduling algorithm, and (*iii*) deriving schedulability tests to validate temporal correctness at design time. The next two sections explain these three steps in detail.

1.2 Classical Real-Time System Model

To predictably guarantee the temporal correctness of a real-time system, *a priori* knowledge of applications (i.e., tasks) running in the system and available resources is necessary. Real-time task models are designed to mathematically describe real-time applications and their associated temporal constraints, while a resource model is used to formally describe available hardware resources. In this section, we describe the widely studied sporadic task model as well as the categories of temporal constraints and the resource model considered in this dissertation.

1.2.1 Sporadic Task Model

The *sporadic task model* [88] is a well-studied and simple recurrent task model. Under the sporadic task model, each *task* is a sequential program that is repeatedly invoked in response to external

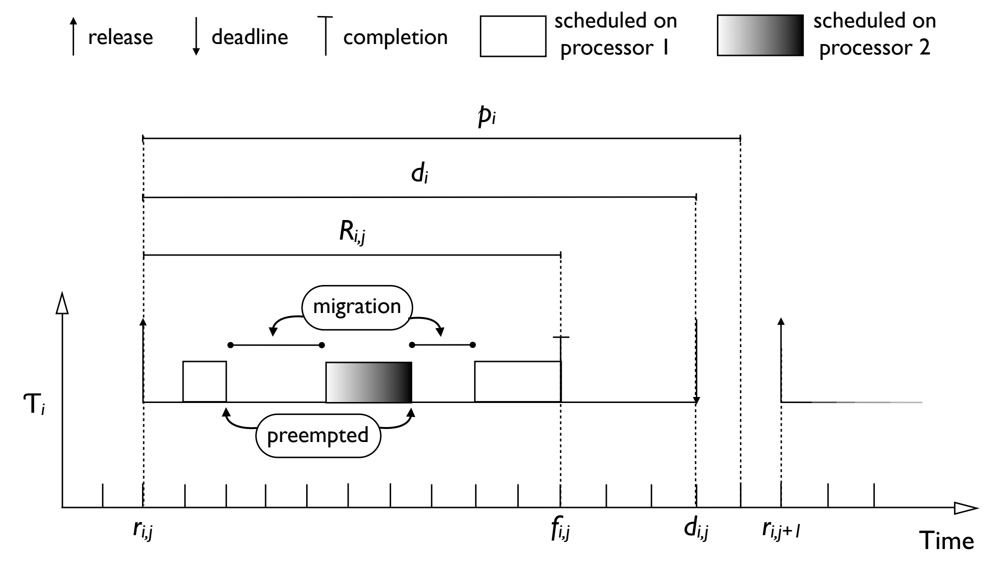


Figure 1.1: Illustration of a constrained-deadline sporadic task τ_i. The job $\tau_{i,j}$ is preempted twice by higher-priority jobs (which are not depicted). Therefore,$\tau_{i,j}$ migrates twice: it first migrates from processor 1 to processor 2, and then back to processor 1.

events such as device interrupts or expiring timers. When a task is invoked in response to an event, it releases a *job* to process the event. Note that a recurrent task can be invoked an infinite number of times, i.e., can generate jobs indefinitely. Next, we provide a detailed definition of the sporadic task model; later, in Chapter 2, more sophisticated and practical real-time task models are considered that are based upon the sporadic task model defined here. An example sporadic task is shown in Figure 1.1. Note that the symbols shown at the top of this figure will be used in subsequent figures throughout this dissertation to denote job releases, deadlines, and completions, respectively. We will omit some of these legends in later figures.

A real-time system can be sometimes modeled as a set of n sporadic tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each sporadic task τ_i is characterized by three parameters (e_i, d_i, p_i), where $e_i > 0$ denotes its per-job worst-case execution time (WCET), $d_i \geq e_i$ denotes its *relative deadline*, and $p_i \geq e_i$ (often called the task's *period*) denotes its *minimum* job inter-arrival time. That is, τ_i can release its first job at any time, but can release any two consecutive jobs only at least p_i time units apart. Each released job executes for at most e_i time units, and each job should complete no more than d_i time units after its release. Note that d_i and p_i are platform-independent since they are associated with the task.

In contrast, e_i depends upon characteristics of the underlying hardware platform such as processor speeds.

The sporadic task model was introduced by Mok in 1983 [88]. It generalizes the earlier *periodic task model* [84]. For a *periodic task*, its period parameter defines the *exact* separation between any two consecutive job releases. A periodic task thus releases jobs on a more regular basis. The motivation behind the sporadic task model is to allow tasks to experience inter-arrival delays so that scenarios where tasks might become inactive in the absence of triggering events can be accommodated.

The j^{th} released job of task τ_i is denoted $\tau_{i,j}$. Each job $\tau_{i,j}$ is eligible to execute at or after its *release time* $r_{i,j}$, where $r_{i,j} \geq 0$. As constrained by the task period, $r_{i,j+1} \geq r_{i,j} + p_i$ holds for any consecutive jobs $\tau_{i,j}$ and $\tau_{i,j+1}$. Each job $\tau_{i,j}$ executes for at most e_i time units and *finishes* at time $f_{i,j}$. Sporadic tasks are sequential, that is, job $\tau_{i,j+1}$ can start execution only after the finish time of $\tau_{i,j}$, even if $r_{i,j+1} \leq f_{i,j}$. Job $\tau_{i,j}$ has a *response time* $R_{i,j} = f_{i,j} - r_{i,j}$, which defines the duration from its release to its finish time. A task τ_i's response time R_i equals the maximum job response time among all of its released jobs. Note that R_i might not be well-defined in cases where response times of τ_i's jobs grow unboundedly.

The *relative deadline* parameter d_i specifies the temporal constraint of task τ_i. Specifically, it defines the range of acceptable response times for a task. Each job $\tau_{i,j}$ has an *absolute deadline* $d_{i,j} = r_{i,j} + d_i$. If a job $\tau_{i,j}$ finishes later than time $d_{i,j}$, then it is *tardy*. A job $\tau_{i,j}$'s *tardiness* is the extent of the deadline miss, which is given by $max(0, f_{i,j} - d_{i,j})$. Depending on the relationship between d_i and p_i, there are three categorizes of sporadic tasks. A task τ_i is said to be an *implicit-deadline* sporadic task if $d_i = p_i$, a *constrained-deadline* sporadic task if $d_i \leq p_i$, or an *arbitrary-deadline* sporadic task if d_i may be of any value (e.g., different from p_i). In this dissertation, we assume implicit-deadline tasks unless noted otherwise.

Utilizations. Another important parameter pertaining to a sporadic task is its *utilization*, which defines the task's required long-term processor demand. Specifically, a task τ_i's utilization is given by $u_i = \dfrac{e_i}{p_i}$, which describes the fraction of one processor τ_i requires. The utilization parameter is meaningful because it quantitatively defines the maximum rate of execution required for a task to receive enough processor capacity to execute its released jobs in the long-term. It also provides a way to determine whether a system is *overloaded*. The *total utilization* of a task set τ is defined as

$U_{sum} = \sum_{\tau_i \in \tau} u_i$. A system is said to be overloaded if the total utilization of all tasks exceeds the number of available processors m. For a constrained-deadline sporadic task τ_i, a useful parameter is its *density*, which is given by $\delta_i = \frac{e_i}{d_i}$. The *total density* of a task set τ is thus defined as $\delta_{sum} = \sum_{\tau_i \in \tau} \delta_i$. As a notational convenience, we let u_{max} and δ_{max} denote the maximum utilization and density of tasks in τ, respectively.

1.2.2 Hard and Soft Temporal Constraints

There are mainly two categories of real-time temporal constraints: *hard real-time* (HRT) and *soft real-time* (SRT). A task is referred to as a ***HRT task*** if none of its released jobs may ever miss its deadline, that is, $f_{i,j} \leq d_{i,j}$ must hold for any job $\tau_{i,j}$. A ***HRT system*** solely consists of HRT tasks. On the other hand, for ***SRT tasks***, deadline misses are allowed. However, any such deadline misses must be bounded by a *constant*, that is, $f_{i,j} - d_{i,j} \leq C$ must hold for any job $\tau_{i,j}$, for some constant C. Similarly, a ***SRT system*** solely consists of SRT tasks. This notion of a SRT constraint is well-studied [25, 39, 76] and the one considered in this dissertation. Note that other notions of SRT constraints exist, including: a specified percentage of deadlines must be met [57], and x out of every y consecutive jobs of each task must meet their deadlines [54, 67, 116].

In practice, both HRT and SRT systems widely exist in many application-specific domains. In general, a HRT system is one whose violation may lead to catastrophic consequences such as loss of life. Military combat, industrial process-control, automotive control, and air-traffic control systems are some examples of HRT systems. In contrast, a SRT system is less critical, as deadline misses may lead to degraded quality of service, but not system failure. Multimedia systems, computer vision and image processing systems, and mobile computing systems are some examples of systems with SRT constraints. Moreover, in many systems, HRT and SRT constraints co-exist in different components. For instance, in an avionics system, the flight management system has HRT constraints while the on-board entertainment system only requires SRT constraints.

Temporal correctness. Constrained by either HRT or SRT requirements, a real-time task's temporal correctness essentially depends upon its maximum response time. Since a job's response time necessarily depends on the underlying scheduling algorithm, temporal correctness is defined with respect to a given scheduling algorithm. The following definitions formally define HRT and SRT correctness.

Definition 1.1. A task set τ is *HRT schedulable* under a scheduling algorithm $\mathcal{A}$ iff $R_i \leq d_i$ holds for each $\tau_i \in \tau$.

Definition 1.2. A task set τ is *SRT schedulable* under a scheduling algorithm $\mathcal{A}$ iff there exists a constant C such that $R_i \leq d_i + C$ holds for each $\tau_i \in \tau$.

1.2.3 Resource Model

In this dissertation, we consider real-time task systems running on an *identical* multiprocessor platform comprised of a set of $m \geq 1$ identical unit-speed processors. As its name suggests, all processors of an identical multiprocessor have the same characteristics. For example, in the absence of contention, they have uniform access times to memory. To realize uniform memory access times, a multiprocessor architecture with a centralized memory that is shared among processors is often implemented. This architecture is commonly referred to as a *symmetric (shared-memory) multiprocessor* (SMP).

Figure 1.2 illustrates example SMP architectures. As shown in Figure 1.2 (a), each processor can have one or more private caches to reduce memory access times. In this dissertation, we assume that each job is allowed to execute on any processor except that it can occupy at most one processor at any point of time (except for the parallel task model given in Chapter 7). We say that a job (task) *migrates* if it executes on different processors. A job migration may incur migration overheads due to the reloading of job-related instructions and data into a local cache. Techniques exist that can lower or eliminate migration overheads. For example, one technique is to restrict the execution of a task or a job to one or a subset of processors. Moreover, for current multicore architectures, different cores on the same die often share a cache at some level, as illustrated in Figure 1.2 (b). Such shared caches are able to reduce migration overheads in many cases, e.g., when task-related instructions are not loaded from memory after a migration.

Accounting for overheads. Besides migration overheads, task preemptions and context switches, and the act of scheduling may also incur system overheads that take processor time from tasks. Thus, to guarantee temporal correctness, any temporal validation approaches must take into account

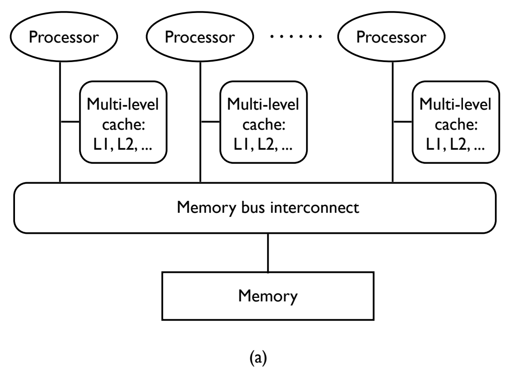
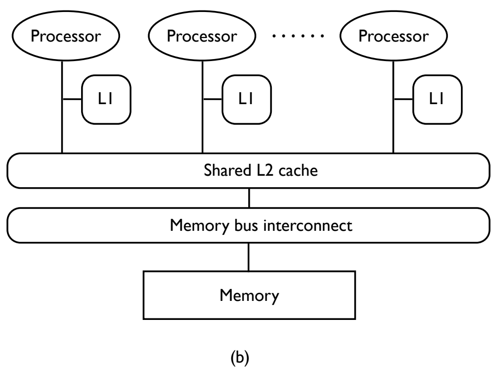

Processor
Processor
Processor
Multi-level cache: L1, L2, ...
Multi-level cache: L1, L2, ...
Multi-level cache: L1, L2, ...
Memory bus interconnect
Memory
(a)


Figure 1.2: Illustration of symmetric multiprocessor architecture **(a)** without and **(b)** with a shared cache.

processor time lost due to overheads. To account for overheads, a common approach is to charge each extrinsic activity (e.g., preemptions, migrations, or scheduler invocations) to each job. This is achieved by inflating each task's WCET by the maximum total time required for all such extrinsic activities charged to any of its released jobs. Throughout this dissertation, we will assume that all system overheads are included in the WCETs of tasks using efficient charging methods [25, 39]. Therefore, the WCET of a task depends on the implementation platform, application characteristics, and the scheduling algorithm.

1.3 Real-Time Scheduling Algorithms and Schedulability Tests

As mentioned in Section 1.1, after formally modeling a real-time system as described in the previous section, we need to utilize real-time scheduling algorithms and derive schedulability tests to validate temporal correctness. (A detailed review of existing real-time scheduling algorithms and schedulability tests designed for the real-time sporadic task model is presented in Chapter 2.) A scheduling algorithm determines when to execute which task on which processor. It determines the specific execution-time intervals and processors for each job while considering any restrictions, such as precedence constraints among jobs and tasks. Different from other non-real-time systems where performance optimization is a major concern, in a real-time system, algorithmic design pertaining to processor-allocation strategies is mainly driven by the need to guarantee temporal correctness. Before discussing in detail existing scheduling algorithms and schedulability tests, we first introduce a few key concepts and metrics that are commonly used in describing properties of real-time scheduling algorithms and in comparing different algorithms.

1.3.1 Concepts and Metrics

As defined in Definitions 1.1 and 1.2, a real-time task system is schedulable iff its temporal constraints will always be satisfied (either in a HRT or SRT sense). Two concepts of importance are *feasibility* and *optimality*, which are both defined in terms of schedulability. Intuitively, a task set is feasible if there is a possibility to schedule it and a scheduling algorithm is optimal if all feasible task sets are schedulable under it. These concepts are formally defined as follows.

Definition 1.3. A task set τ is HRT (respectively, SRT) *feasible* iff there exists at least one scheduling algorithm $\mathcal{A}$ such that τ is HRT (respectively, SRT) schedulable under $\mathcal{A}$.

Definition 1.4. A scheduling algorithm $\mathcal{A}$ is *optimal* iff every task set τ that is HRT (respectively, SRT) feasible is HRT (respectively, SRT) schedulable under $\mathcal{A}$.

To check whether a given real-time sporadic task system is feasible, a straightforward feasibility test is given by the system's total utilization. As defined in Section 1.2, a task τ_i's utilization u_i specifies the fraction of a processor the task requires. Therefore, this implies that the total utilization of a task set cannot exceed the total number of processors, for otherwise the total processor demand could exceed the available supply, which causes tasks to violate their temporal constraints.

Lemma 1.1. *A real-time sporadic task set is feasible on m processors only if $U_{sum} \leq m$.*

The above lemma formally gives a necessary utilization-based feasibility condition for sporadic task sets. In the context of a set of non-recurrent tasks, a necessary feasibility condition that implies Lemma 1.1 was proven by Horn [55].[1] Leung and Merrill [79] showed that $U_{sum} \leq m$ is a necessary feasibility condition for implicit-deadline periodic task sets. For sporadic tasks, a formal proof of Lemma 1.1 for the case $m = 1$ was given by Baruah *et al.* [14], which can easily be generalized to $m > 1$. Note that Lemma 1.1 can be applied equally to HRT and SRT feasibility. For HRT implicit-deadline sporadic task sets, Lemma 1.1 can be strengthened to an equivalence, as given by the following lemma.

[1]Note that in this case, a task's utilization is defined to be its WCET divided by its deadline.

Lemma 1.2. *[11, 79, 84] A real-time implicit-deadline sporadic task set is HRT feasible on m processors iff $U_{sum} \leq m$.*

Capacity loss. The above constraint on total utilization (given our focus on implicit deadlines) implies that any task set with total utilization not exceeding the total processor capacity m should be schedulable. In practice, fully allocating all processor capacity to real-time tasks is not always possible, i.e., task systems with total utilization less than m may not be schedulable on m processors. One primary cause of such capacity loss is due to the choice of scheduling algorithm. If a non-optimal scheduling algorithm is used, then a feasible task set may not be schedulable due to algorithmic capacity loss. In this dissertation, our goal is to design new scheduling algorithms and derive better schedulability tests (as described below) to minimize capacity loss for scheduling complex real-time task systems on multiprocessors.

Schedulability tests. A schedulability test serves the purpose of determining *a priori* whether a real-time task set will be schedulable under a scheduling algorithm $\mathcal{A}$, either in a HRT or SRT sense. Many existing schedulability tests are only *sufficient* but not *necessary* (i.e., are not *exact* tests). This implies that such sufficient tests can be pessimistic, that is, they may incorrectly consider a task set to be unschedulable while the task set is in fact schedulable.

1.4 The Divergence of Theory and Practice

As will be discussed in Chapter 2, real-time scheduling under the sporadic task model on both uniprocessors and multiprocessors has received much attention over the past 40 years. Although significant research progress on this topic has been made, the practical applicability of such research is quite limited due to limitations of the sporadic task model, which we discuss next.

1.4.1 Limitations of the Sporadic Task Model

As defined in Section 1.2, an application that is modeled as a sporadic task triggers a piece of computation recurrently. Unfortunately, several inherent limitations make it inappropriate for modeling many real-world applications. We identify four major limitations of the sporadic task model as listed below.

1. Tasks are computation-only and do not self-suspend: released jobs of a task only request processor (CPU) capacity and are always ready to execute when allocated a processor by the scheduler; no interaction between the task and external devices such as disks could exist.

2. Tasks are totally independent: no precedence constraint exists among tasks.

3. Tasks are purely sequential: a job is completely sequential and thus can only be executed on one processor at any point of time.

4. Tasks are fully preemptive: released jobs do not access any critical sections that must be accessed non-preemptively.

However, in practice, tasks may incur complex runtime behaviours that violate these assumptions. For example, a task may need to access external devices during its execution, such as to read/write data, wait for keyboard input, or access graphics processing units (GPUs). Such interactions with external devices cause the execution of tasks to be suspended (i.e., blocked) by the operating system on CPUs, as illustrated in Figure 1.3. A task is said to *self-suspend* when it incurs such interactions. Task execution on a CPU may be resumed only after the suspending interval ends. The duration of a suspension interval of a task is often called *suspension delay*. Such suspension delays could range from a few nanoseconds to several seconds depending on task characteristics. For instance, delays introduced by disk I/O range from 15ns (for NAND flash) to 15ms (for magnetic disks) per read [63, 74]. Even longer delays are possible when accessing special-purpose devices such as digital signal processors or GPUs [44]. Tasks that may self-suspend cannot be accurately modeled as sporadic tasks.

Besides self-suspensions, tasks may exhibit other types of runtime behaviours, including inter-task data communications, critical section accesses, and parallel code execution segments. Consider a simple real-time video processing system that decodes incoming video streams and then displays them.[2] This system thus naturally consists of two real-time tasks: video decoder and video display, as illustrated in Figure 1.4. Different types of runtime behaviors may be incurred in this system. For instance, graph-based precedence constraints exist between these two tasks because the video display

[2]Such a system could conceivably be applied in a wide range of application domains. For example, it could be applied in an airline entertainment system, or in a surveillance system that must process multiple video streams.

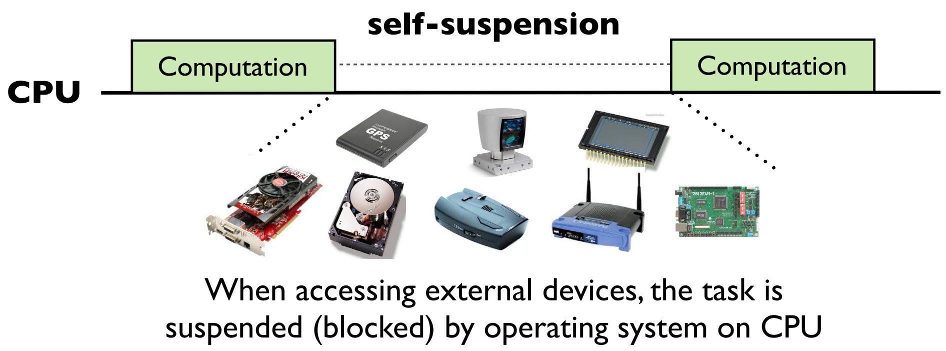

self-suspension
CPU
Computation
Computation
When accessing external devices, the task is suspended (blocked) by operating system on CPU

Figure 1.3: Pervasive self-suspension behaviours.

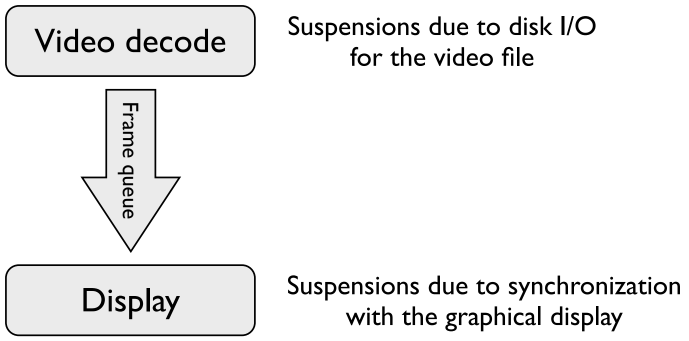

Video decode
Suspensions due to disk I/O for the video file
Frame queue
Display
Suspensions due to synchronization with the graphical display

Figure 1.4: Example real-time video processing system.

task cannot start execution until receiving decoded video frames from the video decoder task. Also, both tasks may self-suspend at runtime due to disk I/O for the video file and synchronization with the graphical display, respectively. Obviously, the sporadic task model, which assumes that such complex runtime behaviors do not arise, is incapable of modeling such tasks.

Due to the limitations of the sporadic task model as discussed above, current research on the real-time scheduling of sporadic tasks on multiprocessors cannot be applied to efficiently support many real-world applications in real-time systems. In the rest of this section, we discuss how various complex runtime behaviors are currently supported by applying state-of-the-art scheduling techniques, and the resulting pessimism.

1.4.2 Limitations of the State-of-the-Art

The general problem of supporting real-time task systems containing complex runtime behaviors (e.g., self-suspensions, precedence constraints, non-preemptive sections, and parallel execution segments) on multiprocessors is known to be hard and remains largely open. Some prior work has been done on scheduling and analyzing task systems containing different complex runtime behaviors. However, such analysis [22, 65, 73, 96, 97, 101, 107, 108, 114, 119] is rather pessimistic and only applies to uniprocessor platforms. We will review such work in detail in Section 2. In the context of multiprocessor scheduling, not much work has been done on supporting real-time task systems with complex runtime behaviors. In practice, such behaviors are currently dealt with by over-provisioning systems, which may cause significant capacity loss. Over-provisioning implies that in order to guarantee temporal correctness, system designers have to utilize more hardware components (e.g., CPUs) than necessary, largely due to the inefficiency and pessimism of current real-time scheduling algorithms and schedulability tests. System over-provisioning is also an economically wasteful practice. With each added component, overall system complexity as well as size, weight, and power consumption (S.W.a.P.) of the system can grow significantly. Moreover, in many safety-critical real-time domains such as avionics, systems have stringent space and energy constraints and thus over-provisioning is problematic. With the rising complexity of networked real-time embedded systems, over-provisioning has led to an increasingly unmanageable proliferation of such systems, to the effect that some modern cars contain over one hundred processors. Therefore, reducing the number of hardware components is highly desirable. Instead of embedding hundreds of networked uniprocessors inside a car, it would be much more desirable to use fewer, but more powerful multicore processors that are highly utilized.

1.5 Research Overview

This dissertation is thus aimed at bridging this gap between practice and theory in the design of multiprocessor real-time systems. Specifically, our goal is to enable practical real-time applications with common types of complex runtime behaviors to be efficiently supported on multiprocessors. By designing new real-time scheduling algorithms and schedulability tests for such applications, this dissertation seeks to avoid over-provisioning systems and to reduce the number of needed hardware components to the extent possible while providing temporal correctness guarantees. Motivated by limitations seen in current research, this dissertation specifically addresses the problem of efficiently supporting the following four common types of complex runtime behaviors:

1. *Self-suspensions*: Self-suspensions usually occur when a task is blocked, waiting for some interaction to occur, such as interactions with humans or external devices, resources sharing, etc.

2. *Graph-based precedence constraints*: Many tasks such as multimedia and signal processing tasks are implemented using processing graph implementation methodologies, where data communications or logical dependencies exist between stages of such tasks.

3. *Mixed types of complex runtime behaviors*: In practice, many applications may have several mixed types of complex runtime behaviors, including self-suspensions, graph-based precedence constraints, and non-preemptivity. For example, consider a pipelined real-time computation where some tasks may require disk accesses and non-preemptivity arises due to system calls or critical sections

4. *Parallel execution segments*: Parallel programming techniques such as OpenMP [28] and MapReduce [36] are commonly used to explicitly express parallelism for certain program segments (i.e., allowing multiple threads to exist within the context of a single job).

1.6 Thesis Statement

The main thesis to be supported by this dissertation is the following.

> *Capacity loss can be significantly reduced on multiprocessors while providing SRT and HRT guarantees for real-time applications that exhibit complex runtime behaviors such as self-suspensions, graph-based precedence constraints, non-preemptive sections, and parallel execution segments by designing new real-time scheduling algorithms and developing new schedulability tests.*

1.7 Contributions

In the following, we briefly summarize the contributions presented in the subsequent chapters.

1.7.1 Multiprocessor SRT Schedulability Test for Globally-Scheduled Self-Suspending Task Systems

The first set of contributions we discuss is two multiprocessor global earliest-deadline-first (GEDF) (defined in Section 2.3.1.2 of Chapter 2) schedulability tests proposed in [81, 83] for SRT self-suspending task systems. The approach presented in [83] was the first to be proposed for dealing with self-suspensions on globally-scheduled (defined in Section 2.3.1.2 of Chapter 2) SRT multiprocessors.

1.7.1.1 The First SRT Suspension-Aware Global Schedulability Test

Prior work showed that suspension delays quite negatively impact schedulability in real-time systems if deadline misses cannot be tolerated [105]. Thus, we focus on investigating whether, on multiprocessor platforms, such negative impacts can be ameliorated if task deadlines are soft.

Perhaps the most commonly used approach for dealing with suspensions is the *suspension-oblivious* analysis [86], which simply integrates suspensions into per-task WCET requirements. However, unless the number of tasks is small and suspension delays are short, this approach may sacrifice significant system capacity. The alternative is to explicitly consider suspensions in the task model and corresponding schedulability analysis; this is known as *suspension-aware* analysis. To improve upon the suspension-oblivious approach, we present the first multiprocessor suspension-aware analysis for globally scheduled SRT sporadic self-suspending task systems.

Specifically, we show in Section 3.2 that GEDF's ability to guarantee bounded tardiness in such systems hinges upon a task parameter that we call the "maximum suspension ratio," denoted ξ_{max}, with range [0,1]. (A task's suspension ratio represents the ratio of its suspension length divided by the sum of its suspension length and execution length.) We present a general tardiness bound, which is applicable to GEDF, that expresses tardiness as a function of ξ_{max} and other task parameters. This bound shows that task systems consisting of both self-suspending tasks and ordinary computational tasks that do not suspend can be supported with bounded tardiness if

$$\xi_{max} < 1 - \frac{U^s_{sum} + U^c_L}{m}, \tag{1.1}$$

where U^s_{sum} is the total utilization of all self-suspending tasks in the system, and U^c_L is the total utilization of the $m-1$ computational tasks of highest utilization.

1.7.1.2 An Improved Utilization Constraint

Using the above approach [83], significant capacity loss may occur when ξ_{max} is large. Unfortunately, it is unavoidable that many self-suspending task systems will have large ξ_{max} values. Thus, the utilization bound can be improved if the value of ξ_{max} can be decreased. Motivated by this, we show in Section 3.3 that ξ_{max} can be effectively decreased by treating *partial* suspensions as computation. That is, we consider intermediate choices between the two extremes of treating *all* (as is done in suspension-oblivious analysis) or *no* (using the analysis in [83]) suspensions as computation. Our technique can find the amount of the suspension time of each task that should be treated as computation in order for the task system to satisfy the utilization constraint and thus become schedulable.

1.7.1.3 An O(m) Analysis Technique

Although the above suspension-aware schedulability test (as presented in Section 3.2) can improve upon the suspension-oblivious approach for many task systems, it may still cause significant capacity loss. A main reason is because this analysis does not fully address the root cause of pessimism due to suspensions. A key step in this suspension-aware analysis involves bounding the number of tasks that have ready jobs (i.e., eligible for executing or suspending) at a specifically defined non-busy time instant t (i.e., at least one processor is idle at t). For ordinary task systems without suspensions, this number of tasks can be safely upper-bounded by $m-1$, where m is the number of processors, for otherwise, t would be busy. For self-suspending task systems, however, non-busy instants can exist due to suspensions even if m or more tasks have ready jobs. The worst-case scenario that serves as the root source of pessimism in prior analysis is the following: *all n self-suspending tasks have jobs that suspend at some time t simultaneously, thus causing t to be non-busy.*

By exploiting this observation, we derive a much improved schedulability test in Section 3.4 that shows that any given sporadic self-suspending task system is schedulable under GEDF scheduling with bounded tardiness if $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ holds, where v^j is the j^{th} maximum ratio of a task's suspension length over its period among all tasks (formal defintions of these terms can be found in

Section 2.1). We show that our derived schedulability test theoretically dominates the suspension-oblivious approach [86], and our previously proposed suspension-aware analysis [83] if every task in the system is a self-suspending task. Moreover, we show via a counterexample that task systems that violate our utilization constraint may have unbounded tardiness. As demonstrated by experiments, our proposed test significantly improves upon prior methods with respect to schedulability, and is often able to guarantee schedulability with little or no capacity loss while providing low predicted tardiness.

1.7.2 Multiprocessor HRT Schedulability Tests for Self-Suspending Task Systems

Although the analysis presented in Chapter 3 can handle the SRT case, how to support HRT self-suspending task systems on multiprocessors (other than using the suspension-oblivious approach) remains as an open issue. As the first attempt at solving this problem, we present in Chapter 4 global suspension-aware multiprocessor schedulability analysis techniques for HRT arbitrary-deadline sporadic self-suspending task models under both GEDF and global task-level fixed-priority (GTFP) (defined in Section 2.3.1.2 of Chapter 2) scheduling (note that this analysis can also be applied to uniprocessors). Our analysis is based upon identifying sufficient conditions for ensuring that each task in any given task system cannot miss any deadlines. These conditions must be checked for each task in the task system. Our analysis shows that schedulability is much less impacted by suspensions than computation on multiprocessors (which is seen in prior uniprocessor analysis [86]). For any job, suspensions of jobs with higher priorities do *not* contribute to the competing work that may prevent the job from executing (while computation does). Indeed, as shown by experiments, schedulability tests based on our new analysis proved to be superior to the suspension-oblivious method of treating all suspensions as computation.

1.7.3 Multiprocessor Scheduling of SRT Task Graphs

Another major contribution of this dissertation is a new (also the first) global scheduling algorithm and corresponding schedulability test for supporting SRT acyclic processing graph method (PGM) graphs on multiprocessors. PGM [69] is a particularly expressive and widely used graph-based formalism that captures graph-based precedence constraints in a general way. One unique property of PGM is that it allows data communications among graph nodes to be specifically modeled. A producing node produces and transports a certain amount of data to a consuming node. In 1998, Goddard and Jeffay showed how to support real-time PGM graphs on an uniprocessor (we will review such prior work in Chapter 2 in detail). In the 12 years that followed, the problem of supporting PGM graphs in multiprocessor systems remained unsolved, due to the additional complexities that arise in that setting.

1.7.3.1 Supporting PGM Task Systems on Multiprocessors

Goddard and Jeffay showed that on a single processor, PGM graphs can be supported by EDF with no capacity loss. Motivated by this, in Section 5.1, we propose a variant of EDF that can be used on multiprocessor platforms as the underlying scheduling algorithm. The associated analysis shows that the main complicating factor in supporting graph-based dependencies in a multiprocessor setting is workload burstiness, which may cause deadline burstiness. (Deadline burstiness also serves as a complicating factor in the uniprocessor case, as shown in [51].) Thus, we propose a technique that effectively postpones deadlines of certain tasks to avoid such burstiness without affecting SRT correctness. We show that the proposed solution is able to achieve no capacity loss for executing PGM task graphs in multiprocessor systems while providing SRT guarantees. That is, any PGM task system τ is SRT schedulable if $U_{sum} \leq m$ holds.

1.7.3.2 Supporting PGM Task Systems in a Distributed System under Clustered Scheduling

We also show in Section 5.2 how to extend the above PGM approach for application in distributed systems comprised of clusters of processors, where scheduling within each cluster is global. Our main contribution in that work is to develop a method for assigning tasks to clusters so as to minimize the amount of data movement across clusters. Once tasks are so assigned, those in each cluster can be scheduled globally as described above (with some slight adjustments due to potential dependencies across clusters). Note that this same task-assignment method can be applied in a fully partitioned system (where each cluster is just one processor), in which case Goddard and Jeffay's original work (with some slight modifications) can be applied on each processor. Although our focus in this work is distributed systems, the same techniques can be applied to schedule tasks on a (large) multiprocessor platform in a clustered fashion.

1.7.4 Multiprocessor SRT Scheduling of Complex Task Graphs Containing Non-Preemptive Sections and Self-Suspensions

Besides handling runtime behaviors such as self-suspensions and graph-based precedence constraints independently, in this dissertation, we also investigate how to support sophisticated real-time task systems containing mixed types of complex runtime behaviors. Specifically, we consider this issue in the context of SRT sporadic task systems in which non-preemptive sections, self-suspensions, and graph-based precedence constraints co-exist.

Any one of these kinds of runtime behaviors can cause a task system to be difficult to analyze from a schedulability perspective. For instance, non-preemptive sections may cause scheduling anomalies (e.g., shortening a job's execution time may actually increase some job's response time [86]) and suspensions may cause unbounded job response times even in lightly-loaded systems [83]. Still, situations may exist in which all three kinds of dependencies are present. Consider, for example, a pipelined real-time computation where some tasks may require disk accesses and non-preemptivity arises due to system calls or critical sections. The timing correctness of such a system may be quite difficult to analyze, particularly if deadline misses cannot be tolerated. However, we show in Chapter 6 that the situation is not nearly so bleak, if bounded deadline tardiness is acceptable.

Specifically, we address the problem of deriving conditions under which bounded tardiness can be ensured when *all* of the above-mentioned behaviors—non-preemptive sections, graph-based precedence constraints, and self-suspensions—are allowed. In considering this problem, we focus specifically on GEDF. Our main result is a transformation process that converts any implicit-deadline sporadic task system with self-suspensions, graph-based precedence constraints, and non-preemptive sections into a simpler system with only suspensions. In the simpler system, each task's maximum job response time is at least that of the original system. This result allows tardiness bounds to be established by focusing only on the impacts of suspensions. It thus enables our prior results on systems with suspensions [82, 83] to be applied to derive tardiness bounds for more complex systems, as scheduled by GEDF.

1.7.5 A Tardiness Bound for Multiprocessor Real-Time Parallel Tasks

The final contribution of this dissertation is a tardiness bound established for GEDF-scheduled sporadic parallel task systems on multiprocessors. Parallel task models pose new challenges to scheduling since intra-task parallelism has to be specifically considered. Recent papers [70, 109] on scheduling real-time periodic and sporadic parallel tasks have focused on providing HRT guarantees under GEDF or partitioned deadline-monotonic (PDM) scheduling (defined in Section 2.3.1.1 of Chapter 2). However, viewing parallel tasks as HRT may be overkill in many settings and furthermore may result in significant schedulability-related capacity loss. Thus, our focus is to instead ensure bounded response times in supporting parallel task systems by applying SRT scheduling analysis techniques. Specifically, we schedule parallel tasks using GEDF, but in contrast to previous work [70, 109], we allow deadlines to be missed provided such misses are bounded (hence response times are bounded as well).

As presented in Chapter 7, our analysis shows that on a two-processor platform, no capacity loss results for any parallel task system. Despite this special case, on a platform with more than two processors, utilization constraints are needed. To discern how severe such constraints must fundamentally be, we present a parallel task set with minimum utilization that is unschedulable on any number of processors. This task set violates our derived constraint and has unbounded response times. The impact of utilization constraints can be lessened by restructuring tasks to reduce intra-task parallelism. We propose optimization techniques that can be applied to determine such a restructuring.

1.8 Organization

The rest of this dissertation is organized as follows. Chapter 2 presents the real-time task models considered in this dissertation and reviews prior work on the scheduling of real-time task systems containing common types of complex runtime behaviors. Multiprocessor schedulability tests for SRT and HRT self-suspending task systems are derived in Chapters 3 and 4, respectively. Techniques for supporting SRT PGM task graphs on multiprocessors and distributed systems are then presented in Chapter 5. The issue of handling mixed types of runtime behaviors including non-preemptive sections, self-suspensions, and graph-based precedence constraints is addressed in Chapter 6. A tardiness bound for GEDF-scheduled sporadic parallel task systems on multiprocessors is derived in Chapter 7. Chapter 8 concludes with a summary of the work presented in this dissertation and a discussion of interesting future work motivated by this dissertation.

CHAPTER 2

Background and Prior Work

In this chapter, we first define the real-time task models that are considered in this dissertation (Section 2.1). Then we review relevant prior work on the scheduling of real-time task systems containing complex runtime behaviors. We specifically review prior work on handling self-suspensions (Section 2.3.2), graph-based precedence constraints (Section 2.3.3), non-preemptive sections (Section 2.3.4), and parallel execution segments (Section 2.3.5).

2.1 Real-Time Task Models

In this section, we present a number of real-time task models that are considered in this dissertation. For readability, we progress from simpler models to more sophisticated ones. A summary of our considered task models and the relationships among them is illustrated in Figure 2.1. We assume that time is integral in this dissertation (as will be discussed in more detail in Section 2.2). Note that there is much flexibility in defining what constitutes a "time unit"—e.g., it could be based on "time" as defined by a hardware clock or "time" as maintained (in software) by the operating system as it responds to periodic clock interrupts.

2.1.1 The Non-Recurrent Task Model

Under the non-recurrent task model, a task releases exactly one job, which has a specified *release time*, *deadline*, and *worst-case execution time* (WCET). A job's release time specifies the earliest time it is allowed to execute, and its WCET defines the maximum time that it will execute on a dedicated processor. For conciseness, we let τ_i denote both a non-recurrent task and its released job, and let $r_{i,1}$ ($d_{i,1}$) denote the job's release time (deadline). An example non-recurrent task is illustrated in Figure 2.2. In practice, this simple task model is insufficient due to the fact that activities

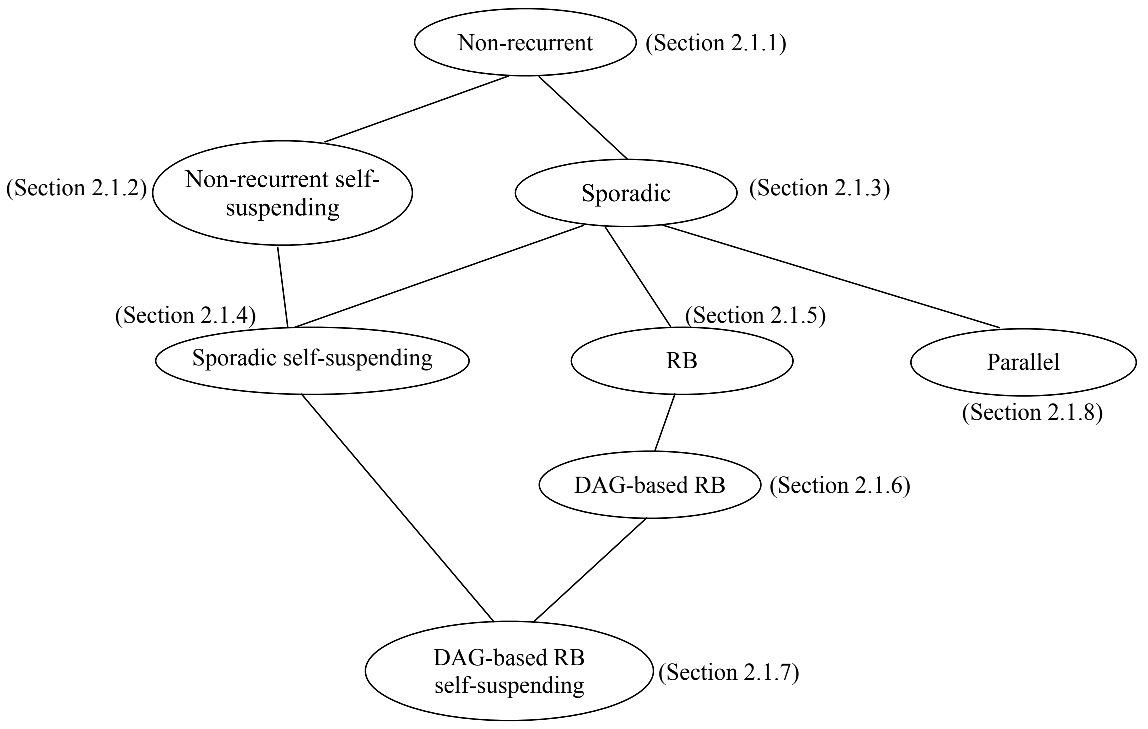


Figure 2.1: Illustration of real-time task models considered in this dissertation. For each pair of connected task models, the successor (lower) generalizes its predecessor (upper). For example, the sporadic task model generalizes the non-recurrent task model.

in many real-time systems are recurrent in nature, that is, such activities may not terminate during normal operation of the system. For example, in the autonomous driving system described earlier in Chapter 1, the environmental sensing task will be recurrently invoked (e.g., every 10 milliseconds) in order to gather real-time environmental data for making safe driving instructions autonomously. In Section 2.1.3, we present a well-studied recurrent task model, namely, the sporadic task model.

2.1.2 The Non-Recurrent Self-Suspending Task Model

As discussed in Chapter 1, one complex runtime behaviour that often arise in practice is self-suspensions, which are caused by interactions with external devices such as disks. Based upon the non-recurrent task model, the non-recurrent self-suspending task model has been proposed and studied in prior work (which will be reviewed in Section 2.3.2). Under this model, each self-suspending task is defined to have a deterministic suspending pattern. That is, each task only releases one job, which has exactly two computation phases separated by one suspension phase. An example

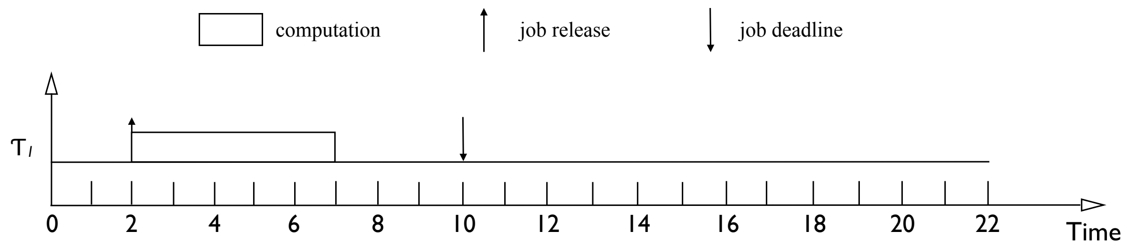


Figure 2.2: An example non-recurrent task τ_1 with a release time at time 2, a deadline at time 10, and a WCET of five time units.

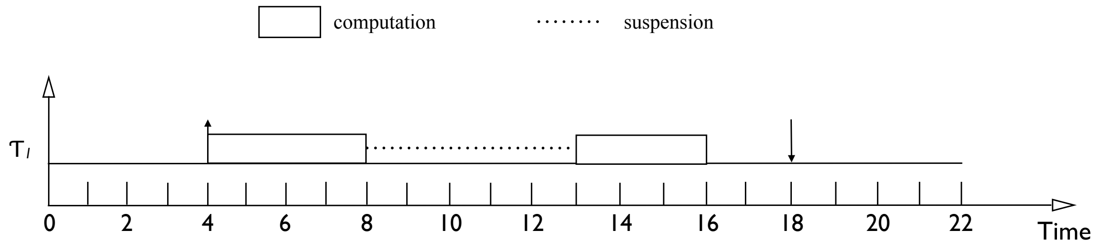


Figure 2.3: Illustration of the non-recurrent self-suspending task model.

non-recurrent self-suspending task is illustrated in Figure 2.3. This model is not sufficient to model suspensions in practice, largely due to the unpredictability of I/O operations. In other words, it is often unrealistic to predict when and how frequently I/O operations may occur. As defined in Section 2.1.4, our proposed sporadic self-suspending task model is more general and takes the unpredictability of I/O into consideration.

Independent suspension assumption. In all of the self-suspending task models we consider in this dissertation, it is assumed that each task's suspensions are simply upper-bounded and will not be interfered with by other tasks' suspensions. As discussed in Section 8.2.3, an interesting avenue for future work is to consider a more sophisticated self-suspending task model where tasks need to compete with each other for accessing shared external devices. Under this model, a task's suspension length is dependent upon other tasks' suspensions. Note that solutions on dealing with independent suspensions derived in this dissertation can still be applied to handle the dependent suspension model assuming *upper bounds* in suspension lengths that factor in contention have been determined.

Notation	Interpretation	definition/constraint
τ	A task set	$\tau = \{\tau_1, \tau_2, ..., \tau_n\}$
τ_i	i^{th} sporadic task	$1 \leq i \leq n$
$\tau_{i,j}$	j^{th} job of τ_i	$j \geq 1$
e_i	WCET of τ_i	$e_i > 0$
d_i	Relative deadline of τ_i	$d_i \geq e_i$
p_i	Period of τ_i; minimum separation between job releases	$p_i \geq e_i$
u_i	Utilization of τ_i	$u_i = e_i / p_i$
δ_i	Density of τ_i	$\delta_i = e_i / min(d_i, p_i)$
$r_{i,j}$	Release time of $\tau_{i,j}$	$r_{i,j} \geq r_{i,j-1} + p_i$
$d_{i,j}$	Absolute deadline of $\tau_{i,j}$	$d_{i,j} = r_{i,j} + d_i$
$f_{i,j}$	Completion time of $\tau_{i,j}$	$f_{i,j} > r_{i,j}$
$R_{i,j}$	Response time of $\tau_{i,j}$	$r_{i,j} = f_{i,j-1} - r_{i,j}$
R_i	Response time of τ_i	$R_i = max_j\{r_{i,j}\}$
U_{sum}	Total utilization of τ	$U_{sum} = \sum_{i=1}^{n} u_i$
δ_{sum}	Total density of τ	$\delta_{sum} = \sum_{i=1}^{n} \delta_i$

Table 2.1: Summary of our notation and the sporadic task models constraints.

2.1.3 The Sporadic Task Model

The *sporadic task model* [88] is a well-studied and simple recurrent task model. Detailed descriptions of this model and the related *periodic task model* were given in Section 1.2.1 (and will not be repeated here). Table 2.1 summarizes our notation for sporadic tasks. Due to the fact that the more sophisticated task models presented in later sections are all based upon the sporadic task model, much of the notation shown in Table 2.1 is reused to denote task parameters under other task models.

2.1.4 The Sporadic Self-Suspending Task Model

The sporadic self-suspending task model extends the sporadic task model by allowing tasks to self-suspend. Similar to sporadic tasks, a sporadic self-suspending task releases jobs sporadically, with each invocation called a *job*. Jobs alternate between computation and suspension phases. We assume that each job of τ_l executes for at most e_l time units (across all of its execution phases) and

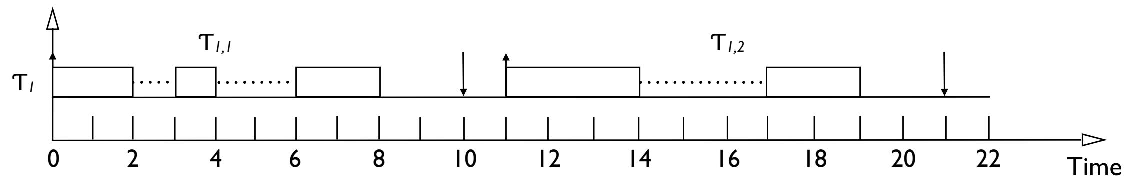


Figure 2.4: Illustration of the sporadic self-suspending task model.

suspends for at most s_l time units (across all of its suspension phases). We place no restrictions on how these phases interleave (a job can even begin or end with a suspension phase). We also do not restrict any task's suspension length, other than upper-bounding it. Note, in particular, that zero-length computation phases are allowed in our model. The circumstances under which a zero-length computation phase can commence execution are just like with any other computation phase. However, a 0-length computation phase that commences execution at time t finishes execution at time t.

The j^{th} job of τ_l, denoted $\tau_{l,j}$, is released at time $r_{l,j}$ and has a deadline at time $d_{l,j}$. Associated with each task τ_l is a period p_l, which specifies both the minimum time between two consecutive job releases of τ_l and the relative deadline of each such job, i.e., $d_{l,j} = r_{l,j} + p_l$. The utilization of a task τ_l is defined as $u_l = e_l/p_l$, and the utilization of the task system τ as $U_{sum} = \sum_{\tau_i \in \tau} u_i$. We require $e_l + s_l \leq p_l$, $u_i \leq 1$, and $U_{sum} \leq m$; otherwise, tardiness can grow unboundedly. Given that a task may have multiple computation and suspension phases, we sometimes use $\tau_l(ea, sb, ec, sd, ..., p)$ to denote the exact sequence of τ_l's phases. For example, $\tau_1(e2, s1, e3, 10)$ denotes that task τ_1 with a period of ten time units first executes for up to two time units, then suspends for up to one time unit, and finally executes for up to three time units. Note that this notation is only used in subsequent examples for conciseness purposes. According to our model, different jobs of the same task can have different computation and suspension behaviors. An example sporadic self-suspending task is shown in Figure 2.4.

2.1.5 The RB Task Model

The *rate-based* (RB) task model is a general task model in which each task τ_i is specified by four parameters: $\tau_i(x_i, y_i, d_i, e_i)$. The pair (x_i, y_i) represents the maximum execution rate of an RB task:

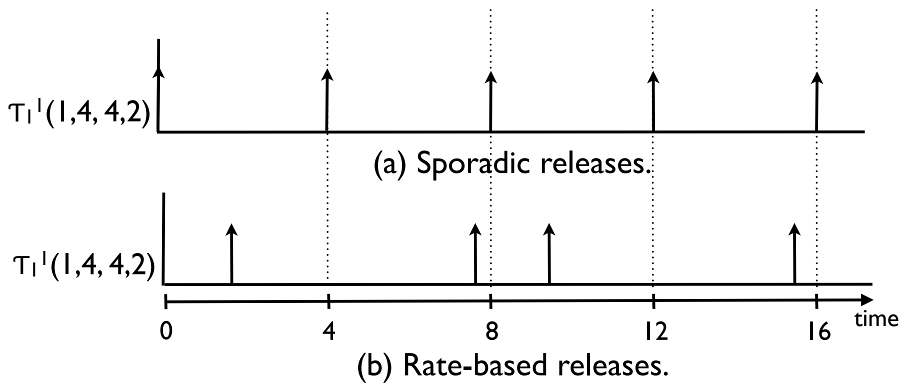


Figure 2.5: **(a)** Sporadic and **(b)** RB releases.

x_i is the maximum number of invocations of the task in any interval $[j \cdot y, (j+1) \cdot y_i)$ $(j \geq 0)$ of length y_i; such an invocation is called a *job* of the task. x_i and y_i are assumed to be non-negative integers. Additionally, d is the task's relative deadline, and e_i is its WCET. The *utilization* of an RB task is $e \cdot \dfrac{x_i}{y_i}$. It is required that $e_i \cdot \dfrac{x_i}{y_i} \leq 1$, for otherwise, the system may become overloaded and tasks may have unbounded response times. The sporadic task model is a special case of the RB task model. In the sporadic task model, a task is released no sooner than every p_i time units, where p_i is the task's period. In the RB task model, the notion of a "rate" is much more general.

Note that the RB task model proposed in this dissertation is similar to the RBE task model proposed by Jeffay and Goddard [61], except that **(i)** the RBE task model assumes that x_i is the number of executions *expected* to be requested in any interval of length y_i, and **(ii)** the RBE task model specifies a minimum separation between consecutive job deadlines of the same task (our RB model does not require such a minimum separation). We elaborate on this point further in Chapter 5.

Example. Figure 2.5 shows job release times and deadlines for a task $\tau_1(1, 4, 4, 2)$. In inset (a), jobs are released sporadically, once every four time units in this case. Inset (b) shows a possible job-release pattern that is not sporadic. As seen, the second job is released at time 7 while the third job is released at time 9. The separation time between these jobs is less than seen in the sporadic schedule.

2.1.6 The DAG-based RB Task Model

The DAG-based RB task model extends the RB task model by allowing precedence constraints to exist among tasks/jobs. The exact manner in which such constraints arise is motivated by those seen in acyclic PGM specifications,[1] as we shall see in Section 2.3.3.

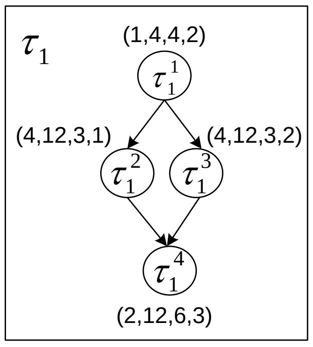
A *DAG-based RB task system* is comprised of a set $\tau^{RB} = \{\tau_1, ..., \tau_n\}$ of n independent DAG-based RB tasks, which we assume are to be scheduled on $m \geq 2$ identical processors. A z-node *DAG-based RB task*, τ_l, consists of z connected RB tasks (or nodes), $\tau_l^1, ...\tau_l^z$, which may have different execution rates. (If $z = 1$, then τ_l is an ordinary RB task.) Each DAG τ_l has a *source node* τ_l^1. Between any two connected nodes is an edge. A node can have outgoing or incoming edges. A source node, however, can only have outgoing edges. We assume that any DAG τ_l is fully-connected, i.e., any node τ_l^h ($h > 1$) is reachable from the source node τ_l^1. As before, an RB task τ_l^h is released at most x_l^h times in any interval $[j \cdot y_l^h, (j+1) \cdot y_l^h)$ ($j \geq 0$), with each such invocation called a *job*. The j^{th} job of τ_l^h in DAG τ_l, denoted $\tau_{l,j}^h$, is released at time $r^{RB}(\tau_{l,j}^h)$ and has a deadline at time $d^{RB}(\tau_{l,j}^h)$. (Note that the notation for job release times and deadlines used here is different from that used in other task models with no graph-based precedence constraints, including the sporadic task model, the RB task model, and the sporadic self-suspending task model.) The *relative deadline* of τ_l^h, denoted d_l^h, is y_l^h / x_l^h. Thus, for any job $\tau_{l,j}^h$,

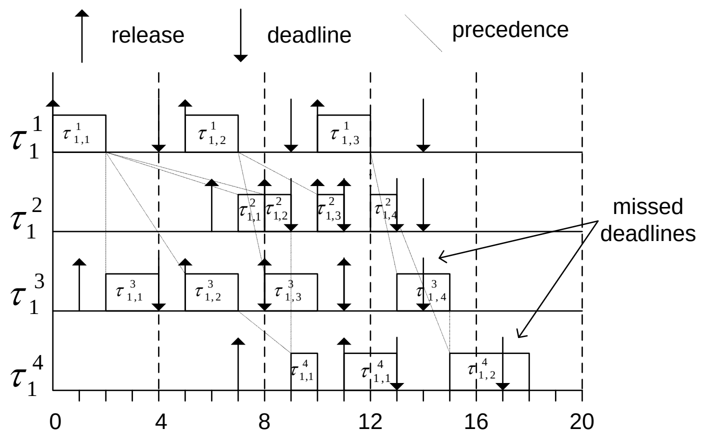
$$d^{RB}(\tau_{l,j}^h) = r^{RB}(\tau_{l,j}^h) + d_l^h. \tag{2.1}$$

The *utilization* of each RB task τ_l^h in τ_l, denoted u_l^h, is $e_l^h \cdot \dfrac{x_l^h}{y_l^h}$. The *utilization of the task system* τ^{RB} is $U_{sum} = \sum_{\tau_i \in \tau^{RB}} \sum_{\tau_i^j \in \tau_i} u_i^j$. We define the *depth* of a task to be the number of edges on the longest path between this task and the source task of the corresponding DAG.

example Figure 2.6(a) shows an example DAG-based RB task system with one DAG τ_1 containing four tasks. τ_1^1 is the source node. τ_1^2 and τ_1^3 are depth-1 tasks while τ_1^4 is a depth-2 task. τ_1^1 has two outgoing edges to τ_1^2 and τ_1^3, and τ_1^4 has two incoming edges from τ_1^2 and τ_1^3. For this system, the total utilization is $u_1^1 + u_1^2 + u_1^3 + u_1^4 = 1/2 + 1/3 + 2/3 + 1/2 = 2$.

[1]The graph-based precedence constraints considered in this dissertation are acyclic and expressed using PGM. Since PGM graph can be naturally represented by a DAG-based RB task set [51], we first present the DAG-based RB model. An overview of PGM is presented in Section 2.3.3





release
deadline
precedence
τ_1^1
τ_1^2
τ_1^3
τ_1^4
$\tau^1_{1,1}$ $\tau^1_{1,2}$ $\tau^1_{1,3}$
$\tau^2_{1,1}$ $\tau^2_{1,2}$ $\tau^2_{1,3}$ $\tau^2_{1,4}$
$\tau^3_{1,1}$ $\tau^3_{1,2}$ $\tau^3_{1,3}$ $\tau^3_{1,4}$
$\tau^4_{1,1}$ $\tau^4_{1,1}$ $\tau^4_{1,2}$
missed deadlines
0
4
8
12
16
20
(b)

Figure 2.6: **(a)** Example DAG-based RB task system and **(b)** GEDF schedule of this example system.

Consecutively-released jobs of the same task must execute in sequence (this is the precedence constraint enforced by the RB task model). Also, precedence constraints exist among tasks/jobs within a DAG. Edges represent potential precedence constraints among connected tasks. If there is an edge from task τ_l^k to task τ_l^h in the DAG τ_l, then τ_l^k is called a *predecessor task* of τ_l^h. We let $pred(\tau_l^h)$ denote the set of all predecessor tasks of τ_l^h.

A job $\tau_{l,j}^h$ may be restricted from beginning execution until certain jobs of tasks in $pred(\tau_l^h)$ have completed. We denote the set of such *predecessor jobs* as $pred(\tau_{l,j}^h)$. Defining $pred(\tau_{l,j}^h)$ precisely requires that job precedence constraints be specified. However, for a general DAG-based RB task system, this is not so straightforward. This is because RB tasks may execute at different rates, and their job release times are not specifically defined (only maximum rates are specified). However, the release time of a job can clearly be no earlier than the release time of any of its predecessor jobs. That is, for any job $\tau_{l,j}^h$ and one of its predecessor jobs, $\tau_{l,v}^w$, we have

$$r^{RB}(\tau_{l,j}^h) \geq r^{RB}(\tau_{l,v}^w). \tag{2.2}$$

As we shall see later in Chapter 5, under PGM, job precedence constraints can be explicitly determine.

If a job $\tau_{l,j}^h$ completes at time t, then its *tardiness* is defined as $max(0, t - d^{RB}(\tau_{l,j}^h))$. A DAG's tardiness is the maximum of the tardiness of any job of any of its tasks. We require $u_l^h \leq 1$ and the total utilization of τ^{RB} not to exceed m, i.e., $U_{sum} \leq m$; otherwise, tardiness can grow unboundedly.

Note that, when a job of a task misses its deadline, the release time of the next job of that task is not altered. Despite this, it is still required that a job cannot execute in parallel with any of its predecessor jobs or the prior job of the same task.

Under GEDF, released jobs are prioritized by their deadlines. Throughout this dissertation, we let $\prec$ denote the priority relationship among jobs, regardless of the scheduling algorithm: $\tau_{i,v}^{w} \prec \tau_{a,b}^{c}$ denotes that job $\tau_{i,v}^{w}$ has higher priority than $\tau_{a,b}^{c}$. We assume that jobs released by DAG-based RB tasks are ordered by deadline as follows: $\tau_{i,v}^{w} \prec \tau_{a,b}^{c}$ iff $d^{RB}(\tau_{i,v}^{w}) < d^{RB}(\tau_{a,b}^{c})$ or $d^{RB}(\tau_{i,v}^{w}) = d^{RB}(\tau_{a,b}^{c}) \wedge (i = a) \wedge (w < c)$ or $d^{RB}(\tau_{i,v}^{w}) = d^{RB}(\tau_{a,b}^{c}) \wedge (i < a)$.

Example. Figure 2.6(b) shows a two-processor GEDF schedule of the example DAG-based RB task system shown in Figure 2.6(a). Regarding the release pattern, note that each task τ_1^k releases at most x_1^k jobs within any time interval $[j \cdot y_1^k, (j+1) \cdot y_1^k)$ $(j \geq 0)$. Regarding job precedence constraints, it is assumed in this example that $\tau_{1,1}^{1} = pred(\tau_{1,1}^{2}) = pred(\tau_{1,2}^{2}) = pred(\tau_{1,1}^{3}) = pred(\tau_{1,2}^{3})$, $\tau_{1,2}^{1} = pred(\tau_{1,3}^{2}) = pred(\tau_{1,3}^{3})$, $\tau_{1,3}^{1} = pred(\tau_{1,4}^{2}) = pred(\tau_{1,4}^{3})$, $pred(\tau_{1,1}^{4}) = \{\tau_{1,2}^{2}, \tau_{1,2}^{3}\}$, and $pred(\tau_{1,2}^{4}) = \{\tau_{1,4}^{2}, \tau_{1,4}^{3}\}$. As seen in the schedule, jobs $\tau_{1,4}^{3}$ and $\tau_{1,2}^{4}$ miss their deadlines by one time unit. Moreover, $\tau_{1,2}^{4}$ can only start execution at time 15 although it is released at time 11. This is because one of its predecessor jobs, $\tau_{1,4}^{3}$, completes at time 15. This causes $\tau_{1,2}^{4}$ to miss its deadline.

2.1.6.1 The Processing Graph Method

Like a DAG-based RB task system, an acyclic PGM graph [69] consists of a set of connected nodes. Each directed edge in a PGM graph is a typed first-in-first-out (FIFO) queue, and (as before) all nodes in a DAG are assumed to be reachable from the DAG's source node. A producing node transports a certain number of tokens (i.e., some amount of data) to a consuming node, as indicated by the data type of the queue. Tokens are appended to the tail of the queue by the producing node and read from the head by the consuming node. A queue is specified by three attributes: a produce amount, threshold, and consume amount. The *produce amount* specifies the number of tokens appended to the queue when the producing node completes execution. The *threshold* specifies the minimum number of tokens required to be present in the queue in order for the consuming node to process any received data. The *consume amount* is the number of tokens dequeued when processing data. We assume in this dissertation that the queue that stores data is associated with the consuming node. That is,

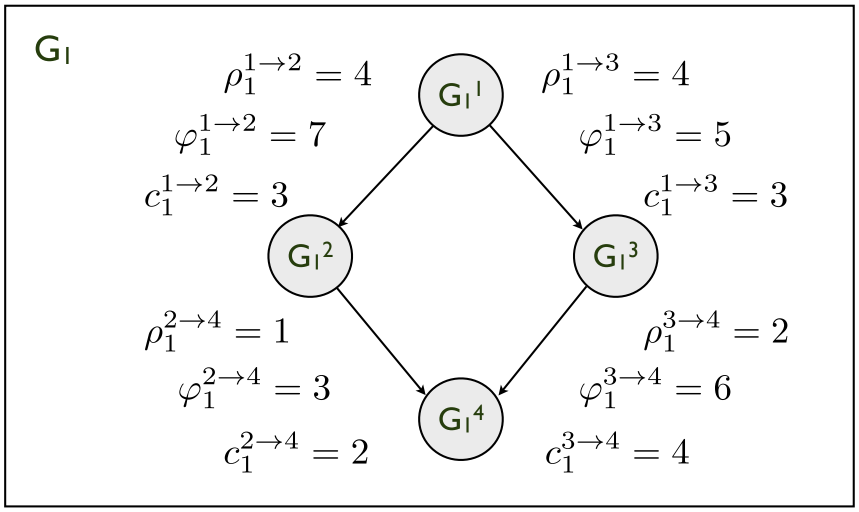


Figure 2.7: Example PGM graph.

data is stored in memory local to the corresponding consuming node.[2] The only restriction on queue attributes is that they must be non-negative integral values and the consume amount must be at most the threshold. In PGM, a node is *eligible* for execution when the number of tokens on each of its input queues is over that queue's threshold. Overlapping executions of the same node are disallowed. For any edge e_i^{jk} connecting nodes G_i^j and G_i^k in a PGM graph G_i, we let $\rho_i^{k\leftarrow j}$ denote its produce amount, $\varphi_i^{k\leftarrow j}$ denote its threshold, and $c_i^{k\leftarrow j}$ denote the consume amount. In the PGM framework, it is often assumed that each source node executes according to a rate-based pattern. Note that, even if all source nodes execute according to a periodic/sporadic pattern, non-source nodes may still execute following a rate-based pattern. We show in Chapter 5 that an acyclic PGM graph can be naturally represented by a DAG-based RB task.

Example. Figure 2.7 shows an example PGM graph G_1 containing four nodes. As an example of the notation, each invocation of G_1^1 appends four tokens to the queue shared with G_1^2. G_1^2 may execute, consuming three tokens, when at least seven tokens are in this queue.

[2]It is more reasonable to associate a queue with the corresponding consuming node than with the producer node because this enables the consumer to locally detect when the queue is over threshold and react accordingly.

2.1.7 The DAG-based RB Self-Suspending Task Model

The DAG-based RB self-suspending task model extends the DAG-based RB task model by allowing tasks/jobs to self-suspend and have non-preemptive sections. The only difference between these two models is that a job released by a DAG-based RB task has only a single computation phase, which is fully-preemptive, while a job released by a DAG-based RB self-suspending task can have interleaving computation and suspension phases (as defined by the sporadic self-suspending task model shown in Section 2.1.4) and computation phases are also allowed to contain non-preemptive code segments. Thus, one notational difference between these two models is that under the DAG-based RB self-suspending task model, each job of a task τ_l^h of the DAG τ_l executes for at most e_l^h time units (across all of its execution phases) and suspends for at most s_l^h time units (across all of its suspension phases). Moreover, the maximum duration of any non-preemptive segment of τ_l^h is denoted b_l^h, and the maximum such value taken over all tasks is denoted b_{max}.

2.1.8 The Parallel Task Model

Under the parallel task model, each parallel task τ_i is a sequence of s_i segments, where the j^{th} segment τ_i^j contains a set of v_i^j threads ($v_i^j > m$ is allowed). The k^{th} ($1 \leq k \leq v_i^j$) thread $\tau_i^{j,k}$ in segment τ_i^j has a WCET of $e_i^{j,k}$. For notational convenience, we order the threads of each segment τ_i^j of each parallel task τ_i in largest WCET order. Thus, thread $\tau_i^{j,1}$ has the largest WCET among all threads in any segment τ_i^j. For any segment τ_i^j, if $v_i^j > 1$, then the threads in this segment can be executed in parallel on different processors. The threads in the j^{th} segment can execute only after all threads of $(j-1)^{th}$ segment (if any) have completed. We let v_i^{max} denote the maximum number of threads in any segment of task τ_i.

The WCET of any segment τ_i^j is defined as $e_i^j = \sum_{k=1}^{v_i^j} e_i^{j,k}$ (when all threads execute sequentially). The WCET of any *parallel task* τ_i is defined as $e_i = \sum_{j=1}^{s_i} e_i^j$ (when all threads in each segment of the task execute sequentially). In our analysis, we also make use of the *best-case execution time* of τ_i on m processors (when τ_i is the only task executing on m processors), denoted e_i^{min}. In general, for any parallel task τ_i, if we allow $v_i^{max} \geq m$ and threads in each segment have different execution costs, then the problem of calculating e_i^{min} is equivalent to the problem of *minimum makespan scheduling* [56], where we treat each thread in a segment as an independent job

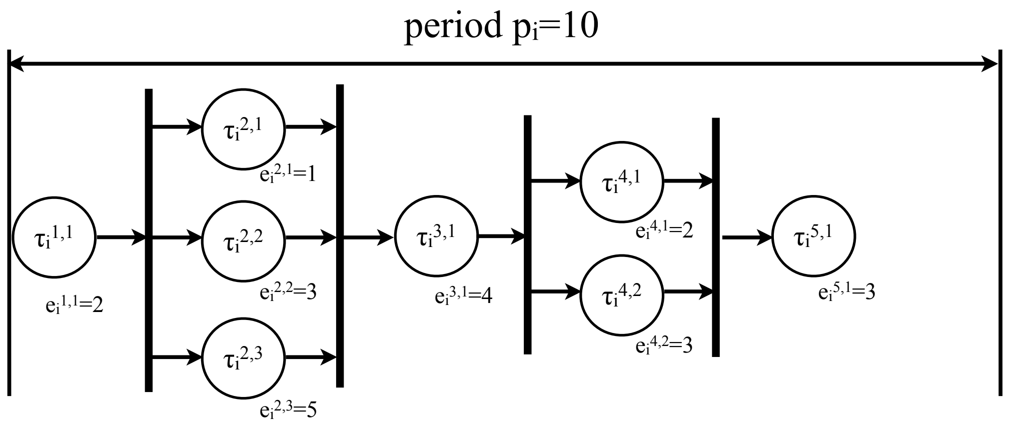


Figure 2.8: Example parallel task τ_i. It has five segments where the second and fourth segments are parallel segments and contain three and two threads, respectively. This task has a WCET of 23 time units, a period of 10 time units, and thus a utilization of 2.3.

and seek to obtain the minimum completion time for executing all such jobs on m processors. This gives us per-segment best-case execution times, which can be summed to yield e_i^{min}. Unfortunately, this problem has been proven to be NP-hard [56]. This problem can be solved using a classical dynamic programming-based algorithm [56], which has exponential time complexity with respect to the per-segment thread count. However, for some special cases where certain restrictions on the task model apply, we can easily calculate e_i^{min} in linear time. For example, when $v_i^{max} \leq m$ holds, $e_i^{min} = \sum_{j=1}^{s_i} e_i^{j,1}$ since in this case all threads of each segment of τ_i can be executed in parallel on m processors and thread $\tau_i^{j,1}$ has the largest execution cost in each segment τ_i^j. Moreover, when all threads in each segment have equal execution costs, $e_i^{min} = \sum_{j=1}^{s_i} \sum_{k=1}^{\lceil v_i^j/m \rceil} e_i^{j,1}$, because the execution of each segment τ_i^j can be viewed as the executions of $\lceil v_i^j/m \rceil$ sequential sub-segments, each with an equal execution cost of $e_i^{j,1}$.

Each parallel task is released repeatedly, with each such invocation called a *job*. The k^{th} job of τ_i, denoted $\tau_{i,k}$, is released at time $r_{i,k}$. Associated with each task τ_i is a period p_i, which specifies the minimum time between two consecutive job releases of τ_i. We require $e_i^{min} \leq p_i$ for any task τ_i; otherwise, response times can grow unboundedly. The utilization of a task τ_i is defined as $u_i = e_i/p_i$, and the utilization of the task system τ as $U_{sum} = \sum_{\tau_i \in \tau} u_i$. We require $U_{sum} \leq m$; otherwise, response times can grow unboundedly. For any job $\tau_{i,k}$ of task τ_i, its u^{th} segment is denoted $\tau_{i,k}^u$, and the v^{th} thread of this segment is denoted $\tau_{i,k}^{u,v}$. An example parallel task is shown in Figure 2.8.

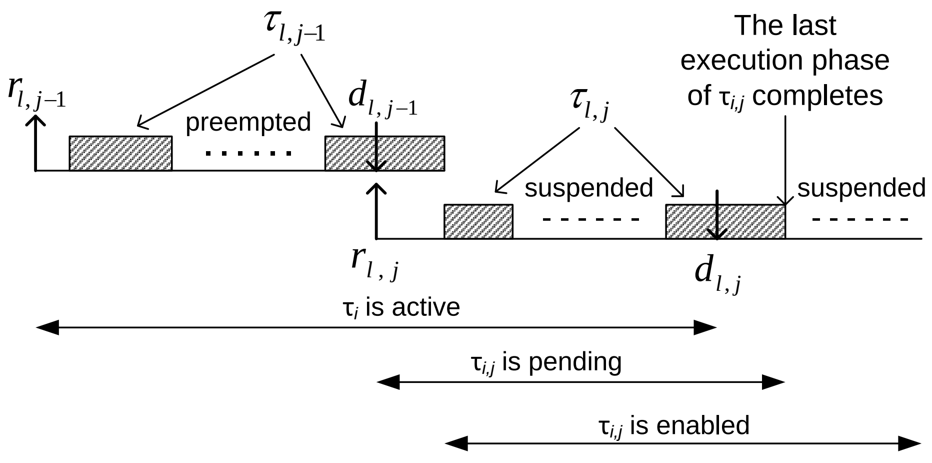


Figure 2.9: Illustration of Definitions 2.1-2.3.

2.2 Common Definitions

The following definitions and concepts are used throughout this dissertation.

We assume that time is integral. Thus, a job that executes or suspends at time instant t executes or suspends during the entire time interval $[t, t+1)$. The time interval $[t_1, t_2)$, where $t_2 > t_1$, consists of all time instances t, where $t_1 \leq t < t_2$, and is of length $t_2 - t_1$. For any time $t > 0$, the notation t^- is used to denote the time $t - \varepsilon$ in the limit $\varepsilon \to 0+$, and the notation t^+ is used to denote the time $t + \varepsilon$ in the limit $\varepsilon \to 0+$.

Definition 2.1. A task τ_i is *active* at time t if there exists a job $\tau_{i,v}$ of τ_i such that $r_{i,v} \leq t < d_{i,v}$.

Definition 2.2. A job is considered to be *completed* if it has finished its last phase (be it suspension or computation). Job $\tau_{i,v}$ is *enabled* at t if $t \geq r_{i,v}$, $\tau_{i,v}$ has not completed by t, and its predecessor $\tau_{i,v-1}$ (if any) has completed by t. Job $\tau_{i,v}$ is *ready* at t if $t \geq r_{i,v}$, $\tau_{i,v}$ has not completed by t, its predecessor $\tau_{i,v-1}$ (if any) has completed by t, and $\tau_{i,v}$ is not suspending at t.

Definition 2.3. Job $\tau_{i,v}$ is *pending* at time t if $t \geq r_{i,v}$ and $\tau_{i,v}$ has not completed its last computation phase by t. Note that $\tau_{i,v}$ is not pending at t if it has completed all its execution phases by t but not all of its suspension phases.

The above three definitions are illustrated in Figure 2.9.

Definition 2.4. A time instant t is *busy* for a job set J if all m processors execute jobs in J at t. A time interval is busy for J if each instant within it is busy for J.

LAG-based Analysis The schedulability tests derived in Sections 3.2, 3.4, and 7.2 are based upon a lag-based analysis framework. To avoid repetition in later sections, we present the concepts of lag and LAG below.

Let $A(\tau_{i,j}, t_1, t_2, S)$ denote the total allocation to the job $\tau_{i,j}$ in an arbitrary schedule S in $[t_1, t_2)$. Then, the total time allocated to all jobs of τ_i in $[t_1, t_2)$ in S is given by

$$A(\tau_i, t_1, t_2, S) = \sum_{j \geq 1} A(\tau_{i,j}, t_1, t_2, S).$$

Definition 2.5. For any given task system τ, a *processor share* (PS) schedule is an ideal schedule where each task τ_i executes with a rate equal to u_i when it is active (which ensures that each of its jobs completes exactly at its deadline). *Note that complex runtime behaviors including self-suspensions, precedence constraints, non-preemptive sections, and parallel execution segments are not considered in the PS schedule.* A valid PS schedule exists for τ if $U_{sum} \leq m$ holds.

Consider a PS schedule *PS*. By Definition 2.5, in such a schedule, if τ_i is active throughout $[t_1, t_2)$, then

$$A(\tau_{i,j}, t_1, t_2, PS) = (t_2 - t_1)u_i. \tag{2.3}$$

The difference between the allocation to a job $\tau_{i,j}$ up to time t in a PS schedule and an arbitrary schedule S, denoted *the lag of job $\tau_{i,j}$ at time t in schedule S*, is defined by

$$\begin{aligned} lag(\tau_i, t, S) &= \textstyle\sum_{j \geq 1} lag(\tau_{i,j}, t, S) \\ &= A(\tau_{i,j}, 0, t, PS) - A(\tau_{i,j}, 0, t, S). \end{aligned} \tag{2.4}$$

The concept of lag is important because, if lags remain bounded, then tardiness is bounded as well. The *LAG* for a finite job set J at time t in the schedule S is defined by

$$\begin{aligned} LAG(J, t, S) &= \textstyle\sum_{\tau_{i,j} \in J} lag(\tau_{i,j}, t, S) \\ &= \textstyle\sum_{\tau_{i,j} \in J} (A(\tau_{i,j}, 0, t, PS) - A(\tau_{i,j}, 0, t, S)). \end{aligned} \tag{2.5}$$

2.3 Prior Work

In this section, we first review a number of existing scheduling algorithms and associated schedulability tests designed for sporadic task systems. Then we review relevant prior work on the scheduling of real-time task systems containing complex runtime behaviors including self-suspensions (Section 2.3.2), graph-based precedence constraints (Section 2.3.3), non-preemptive sections (Section 2.3.4), and parallel execution segments (Section 2.3.5).

2.3.1 Real-Time Scheduling of Sporadic Task Systems

We first review a number of existing scheduling algorithms and associated schedulability tests designed for the sporadic task model. Since job releases are not known *a priori*, pre-computing schedules off-line is impossible for sporadic task systems. Therefore, existing scheduling algorithms make online scheduling decisions, which is typically achieved by assigning different priorities to released jobs dynamically. We will consider uniprocessor scheduling first and then multiprocessor scheduling.

2.3.1.1 Uniprocessor Scheduling

To schedule a set of real-time sporadic tasks on an uniprocessor, existing scheduling algorithms can be divided into two major categories depending on how priorities are assigned: (*i*) job-level fixed-priority (JFP) scheduling, (*ii*) job-level dynamic-priority (JDP) scheduling, and (*ii*) task-level fixed-priority (TFP) scheduling.

JFP scheduling. Under JFP scheduling, each job has a unique priority that is fixed throughout its execution. The most important JFP scheduling algorithm is the *earliest-deadline-first* (EDF) scheduler, which assigns higher priorities to jobs with earlier absolute deadlines. If two jobs have the same absolute deadlines, then we assume that the job with a smaller corresponding task ID has higher priority. EDF is significant due to its optimality on a uniprocessor [38, 68, 80, 84]. That is, every feasible sporadic task set can be scheduled by EDF.[3] For an implicit-deadline sporadic task set τ, all deadlines can be met by EDF iff $U_{sum} \leq 1$ [80, 84]. In contrast, if $U_{sum} > 1$, then some tasks

[3]Note that EDF is also optimal for scheduling non-recurrent tasks and periodic tasks on a uniprocessor.

in τ will have unbounded tardiness (and hence response times as well). This implies that the notions of HRT and SRT schedulability are the same for implicit-deadline task sets scheduled under EDF on a uniprocessor.

JDP scheduling. Unlike JFP schedulers, a JDP scheduling algorithm assigns a priority to each job that can dynamically change over time. It is easy to see from their denitions that JDP is a generalization of JFP scheduling. *Least-laxity-first* (LLF) [88] is a well-known JDP scheduling algorithm, which gives higher priorities to jobs with smaller slack times. The *slack* time of a job at any given time is the difference betewen the amount of time remaining until the job's deadline and its remaining execution requirement. Like EDF, LLF is also optimal for sporadic task systems on a uniprocessor. Note that most JFP and TFP schedulers such as EDF and *rate-monotonic* (RM) are event-driven, that is, such schedulers react to events such as job releases and completions, and reschedule immediately when required. In contrast, LLF is a quantum-driven scheduler, which only makes scheduling decisions at times that are integer multiples of a scheduling quantum Q. In other words, the scheduler is invoked strictly every Q time units.

TFP scheduling. Under TFP scheduling, a fixed priority is assigned to all the jobs of each task. From the definitions, we can see that JFP and JDP are generalizations of TFP scheduling. The most well-known and widely used TFP scheduling algorithm is the RM scheduler, introduced by Liu and Layland in their seminal work on uniprocessor real-time scheduling [84]. Under RM scheduling, tasks are prioritized in the order of increasing periods, that is, tasks with smaller periods are assigned higher priorities. We assume that any ties in period are broken by comparing task IDs (a job with a smaller task ID has higher priority). As proved by Liu and Layland [84], RM is in fact not optimal on a uniprocessor even for implicit-deadline periodic tasks. They also derived a HRT schedulability test showing that an implicit-deadline periodic task set τ is HRT schedulable under RM if $U_{sum} \leq n \cdot (2^{1/n} - 1)$ holds, where n is the number of tasks. When $n \rightarrow \infty$, the bound $n \cdot (2^{1/n} - 1) \rightarrow ln2 \approx 0.69$. Moreover, Liu and Layland further showed that RM scheduling is optimal on a uniprocessor for periodic task systems with respect to TFP schedulers. That is, any task set that is schedulable under some TFP scheduler is also schedulable under RM scheduling. Therefore, any periodic task set with total utilization greater than 0.69 (approximately) is deemed unschedulable under RM or any TFP scheduler. Note that this schedulability test is only a sufficient test, which implies that higher-utilization task sets exist that are in fact schedulable under RM scheduling. For example, two implicit-deadline sporadic tasks with parameters $e_i = 1$ and $p_i = 2$ are schedulable under RM scheduling, yet their total utilization is $1 > 2 \cdot (2^{1/2} - 1) \approx 0.83$.

Another widely studied TFP scheduling algorithm is the *deadline-monotonic* (DM) scheduler. Leung and Whitehead [37] (1982) showed that DM is optimal for constrained-deadline sporadic task sets on a uniprocessor. Since $p_i = d_i$ holds for implicit-deadline tasks, DM scheduling generalizes RM scheduling.

Joseph and Pandya later derived an exact schedulability test for constrained-deadline tasks under TFP scheduling [62] by computing an upper bound on the maximum response time for each task explicitly. The resulting HRT schedulability test due to such response time bounds is given in the following theorem.

Theorem 2.1. *[62] A constrained-deadline sporadic task set τ is HRT schedulable under TFP scheduling (i.e., under any priority assignment) if $R_i \leq d_i$ holds for each task $\tau_i \in \tau$, where $R_i \geq e_i$ is the smallest value satisfying the following equation:*

$$R_i = e_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{p_h} \right\rceil \cdot e_h.$$

Each R_i can be iteratively computed by using e_i as an initial value and by repeatedly re-evaluating the above equation until the left-hand side and the right-hand side converge. It has also been proved that convergence is guaranteed if $U_{sum} \leq 1$ [62]. According to our definition of SRT schedulability defined in Definition 1.2, Theorem 2.1 also implies that TFP scheduling is optimal on a uniprocessor with respect to SRT schedulability since any task set with utilization exceeding one is infeasible on a uniprocessor.

In general, establishing HRT schedulability based upon deriving response time bounds is called *response time analysis* (RTA). The result presented in Theorem 2.1 was later extended to be applicable to arbitrary-deadline task sets, where response time bounds are derived for arbitrary-deadline tasks [75].

Summary on uniprocessor real-time scheduling. Under uniprocessor real-time scheduling, EDF and LLF are optimal for sporadic task systems (non-recurrent and periodic task systems as well)

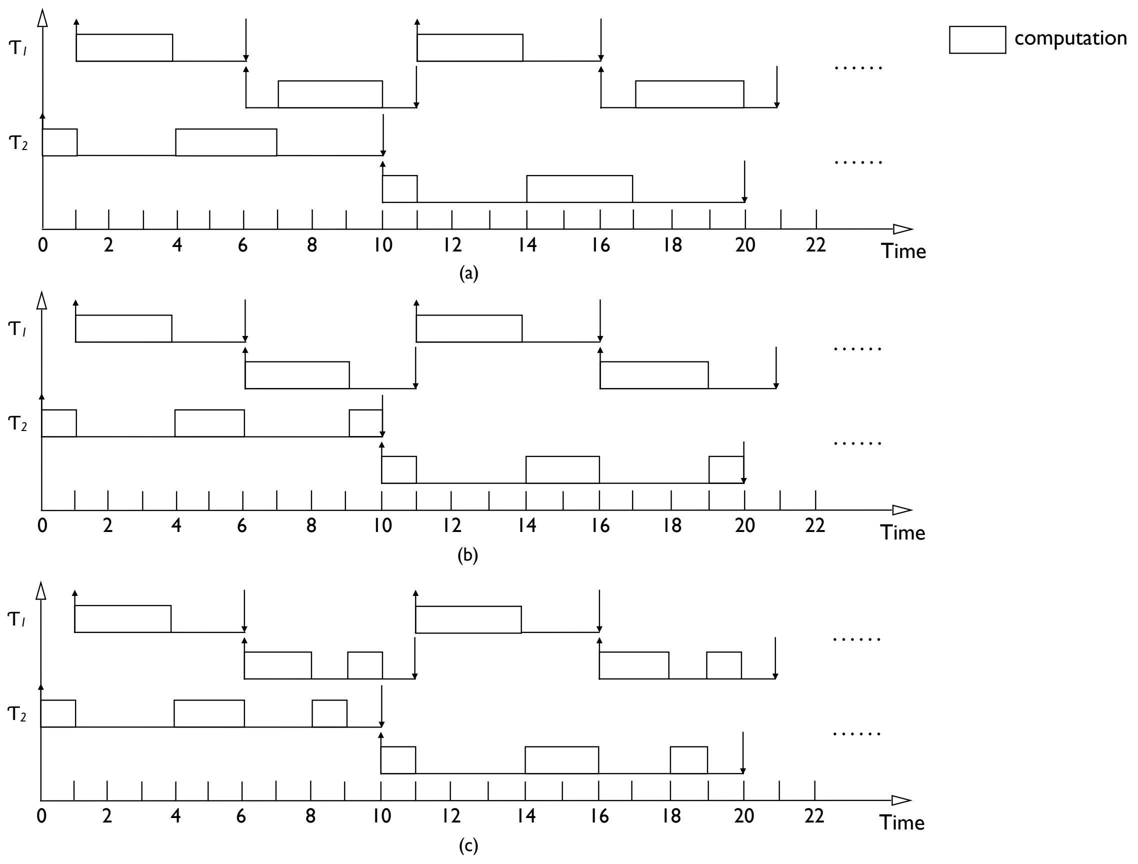


Figure 2.10: Uniprocessor schedules under **(a)** EDF, **(b)** RM, and **(c)** LLF for an example task system with two sporadic tasks $\tau_i(e_1 = 3, p_1 = 5)$ and $\tau_2(e_2 = 4, p_2 = 10)$.

with respect to both HRT and SRT schedulability, while TFP scheduling is optimal for sporadic task systems only with respect to SRT schedulability. Partial schedules of an example task set scheduled under EDF, RM, and LLF are shown in Figure 2.10.

2.3.1.2 Multiprocessor Scheduling

In this section, we discuss multiprocessor real-time scheduling in some detail. Multiprocessor scheduling algorithms can be divided into three major categories: (*i*) partitioning, (*ii*) global scheduling, and (*iii*) clustered scheduling, as illustrated in Figure 2.11 (a), (b), and (c), respectively.[4] In the following sections, we will discuss each category in some detail. Note that similar to the categoriza-

[4]Another category of multiprocessor scheduling algorithms is called *semi-partitioned scheduling* that serves as a compromise between pure partitioned and global scheduling, first proposed by Anderson *et al.* [1]. In a semi-partitioned scheduling algorithm, most tasks are statically assigned to a processor and only a few tasks migrate among processors (as under global scheduling).

tion under uniprocessor scheduling, global scheduling algorithms can be divided into three major categories: GTFP scheduling, global JFP (GJFP) scheduling, and global JDP (GJDP) scheduling.

Partitioning. Under partitioning, tasks are statically partitioned among processors, that is, each task is bound to execute on a specific processor and will never migrate to another processor. Different processors can apply different scheduling algorithms. The algorithm that partitions tasks among processors should ensure that the total utilization of tasks assigned to a processor satisfies the schedulability condition of the corresponding scheduler used on that processor.

Several well-known bin-packing heuristics [32] such as first-fit, best-fit, and worst-fit can be applied to partition tasks among processors. It has been shown in [87] that an implicit-deadline task set τ is HRT schedulable on m processors if $U_{sum} \leq \frac{m+1}{2}$ when using either the first-fit, best-fit, or worst-fit decreasing heuristic for partitioning and EDF scheduling on each processor. For TFP scheduling, it has been shown in [3] that an implicit-deadline task set τ is HRT schedulable on m processors if $U_{sum} \leq \frac{m}{2}$ when using common bin-packing heuristics and TFP scheduling on each processor.

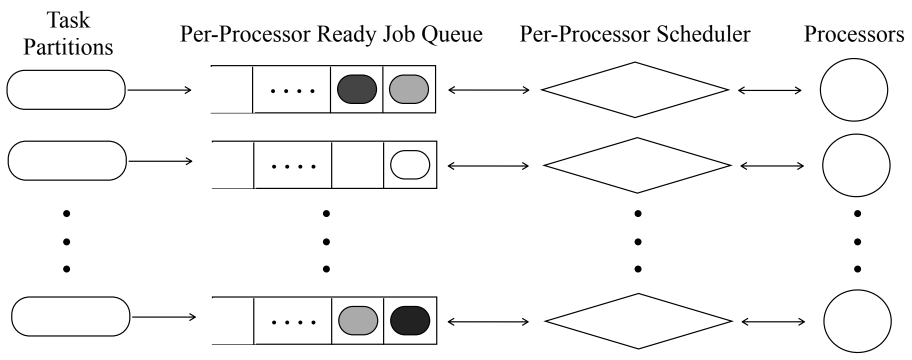
Global TFP scheduling. In contrast to partitioning, under global scheduling, a single global ready queue is used for storing ready jobs. Jobs are allowed to migrate from one processor to another at any point of time. At any time instant, at most m ready jobs with the highest priority among all jobs in the global ready job queue execute on the m processors. Under GTFP scheduling, tasks are assigned fixed priorities and jobs inherit the priority of the task to which they belong.

Global TFP scheduling was first considered by Dhall and Liu [42] who showed that RM is not optimal for scheduling HRT periodic tasks on a multiprocessor. They specifically presented HRT task sets of total utilization approaching one that are unschedulable under RM on $m > 1$ processors. This is known as the *Dhall effect*. This result is rather pessimistic since it implies that RM scheduling can cause algorithmic capacity loss approaching $m - 1$.[5]

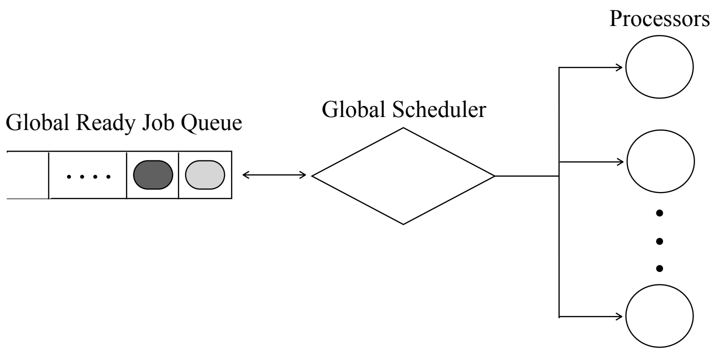
In the context of SRT schedulability with bounded tardiness, Devi [39] showed by means of a counterexample that global RM scheduling does not necessarily ensure bounded tardiness. A counterexample task set has also been constructed in [39] to prove this negative claim. Moreover, no provably optimal TFP scheduler or exact feasibility test for GTFP scheduling is known [35].

[5]Note that this negative result also holds for GEDF scheduler (defined in the following section).

(a)



(b)



(c)

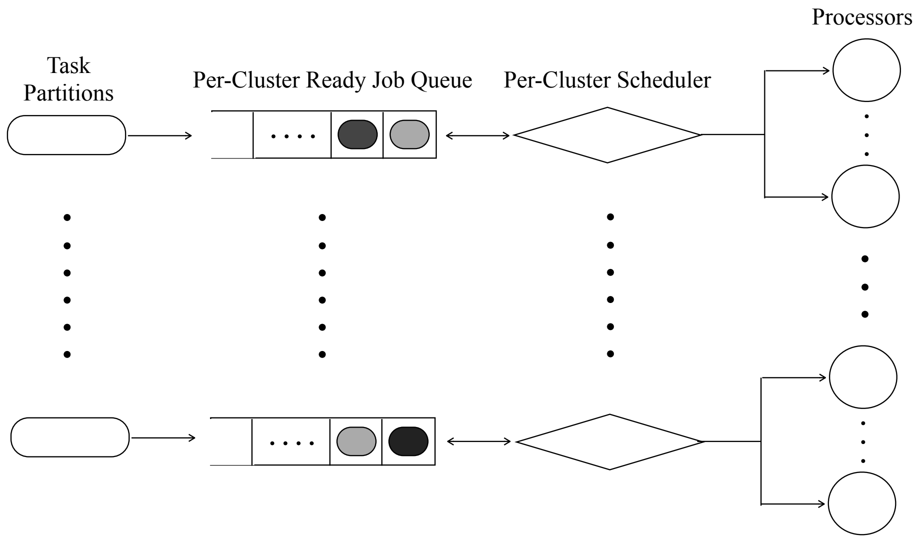


Figure 2.11: **(a)** Partitioning, **(b)** global scheduling, and **(c)** clustered scheduling

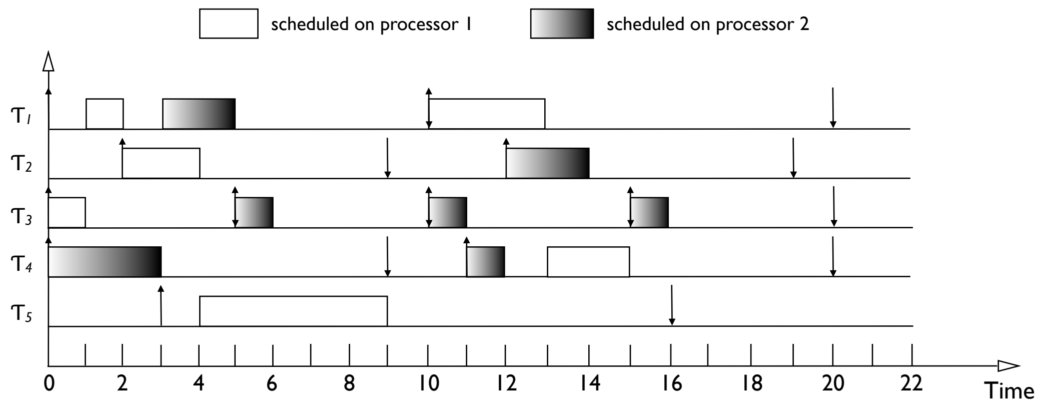


Figure 2.12: Example two-processor G-EDF schedule of the task set that consists of five sporadic tasks: $\tau_1(3,10), \tau_2(2,7), \tau_1(1,5), \tau_1(3,9)$ and $\tau_1(5,13)$, where the notation $\tau_i(e_i,p_i)$ is used. As we can observe in the schedule, jobs are allowed to be preempted and migrate among processors at any time due to GEDF.

Nevertheless, multiprocessor response-time analysis can be applied to derive an upper bound on response times for GTFP scheduling with a given priority assignment [35].

In this dissertation, we are more interested in GJFP scheduling since global scheduling is most compelling in the context of SRT constraints. In particular, in an SRT context, GTFP scheduling does not ensure bounded tardiness in all cases [39] while GEDF scheduling, which is GJFP, does [39, 41]. We thus focus on GJFP, particularly GEDF scheduling, which we describe next.

GJFP scheduling. Similar to GTFP scheduling, currently there does not exist an *exact* HRT feasibility test for GJFP scheduling. Among GJFP scheduling algorithms, GEDF is the most widely-studied one and also the one that we focus on in this dissertation. Under GEDF scheduling, jobs are prioritized by EDF and the $min(m,\alpha)$ jobs with the highest priorities are selected to execute, where α is the number of ready jobs. An example GEDF schedule is shown in Figure 2.12. Several well-known SRT and HRT schedulability tests have been derived based on GEDF. We review them next.

SRT schedulability tests for GEDF. In 2005, Devi and Anderson first proved in their seminal work [41] that any sporadic task set with total utilization up to m can be scheduled on m processors under both GEDF and non-preemptive GEDF[6] with bounded tardiness. Therefore, GEDF is SRT optimal because any feasible sporadic task set that does not over-utilize the system has bounded

[6]Under non-preemptive GEDF, jobs are scheduled according to GEDF but execute non-preemptively.

tardiness under GEDF. Later, this result was extended to prove that the same SRT schedulability can be ensured under a large class of schedulers, namely *window-constrained schedulers* [76, 78]. Under a window-constrained scheduler, each job is prioritized by a time point that can vary at runtime but has to remain within a constant-sized interval in which the job's release time resides. A wide range of scheduling algorithms such as GEDF and global first-in-first-out (GFIFO) [7] belongs to the category of window-constrained schedulers.

HRT schedulability tests for GEDF. Several well-known HRT schedulability tests have been derived for GEDF, including the density test [21, 52], Baker's test [7], Bertogna and Cirinei's multiprocessor response-time analysis [18], and Baruah's test [10].

Density test. In 2003, Goossens *et al.* first derived a HRT GEDF schedulability test showing that an implicit-deadline periodic task set τ is HRT schedulable under GEDF if $U_{sum} \leq m-(m-1) \cdot u_{max}$ holds [52]. Later this result was extended in [21] to be applicable to arbitrary-deadline sporadic task sets by substituting utilization with density. The resulting schedulability test is often referred to as the *density test* [21, 52]: an aribitrary-deadline sporadic task set τ is HRT schedulable under GEDF on m processors if $\delta_{sum} \leq m-(m-1) \cdot \delta_{max}$ holds.

Baker's test. In 2003, Baker developed a seminal analysis framework that has since enabled several influential results, including multiprocessor response-time analysis and Baruah's test discussed in the reminder of this section. Baker's proposed analysis strategy can be summarized from a high-level point of view as follows. For a given GEDF schedule of a task set τ to be analyzed, suppose that job $\tau_{i,k}$ is the first job that misses its deadline. First we compute an upper bound on the amount of work that GEDF can be required to execute over a specifically defined interval $[t_o, d_{i,k})$, where t_o is a specifically defined point of time and $d_{i,k}$ is the deadline of $\tau_{i,k}$. A straightforward choice of t_o could be $r_{i,k}$, the release time of $\tau_{i,k}$. Then, by setting this bound to be large enough to force $\tau_{i,k}$ to miss its deadline, that is, to deny $\tau_{i,k}$'s e_i units of execution over its scheduling window $[r_{i,k}, d_{i,k})$, we can derive a schedulability condition. It turns out that in the case of implicit-deadline sporadic task sets, Baker's test is equivalent to the density test [17]. But in the case of constrained- and arbitrary-deadline task sets, these two tests are incomparable.

[7]GFIFO prioritize jobs according to their releases.

Multiprocessor response time analysis. In 2005, Baker's analysis framework was adopted by Bertogna, Cirinei, and Lipari to develop a series of less pessimistic schedulability tests for constrained-deadline sporadic tasks [18, 20, 21]. Among these tests, Bertogna and Cirinei's multiprocessor response-time analysis [18] dominates the other two tests [17]. Although response-time analysis can be less pessimistic than either the density test or Baker's test [17], it is incomparable to both tests. In other words, there exist task sets that pass the density test or Baker's test that are deemed unschedulable under multiprocessor response-time analysis.

Baruah's test. In 2007, Baruah proposed an improved schedulability test based upon Baker's proof technique for constrained-deadline task sets. The key observation made by him is that by smartly defining the analyzed interval $[t_o, d_{i,k})$, scheduability can be significantly improved. Due to the way t_o is defined in Baker's test, Baker's test is based upon examining a "worst-case" scenario in which each of the n tasks in the system carry work into $[t_o, d_{i,k})$ (i.e., all n tasks have pending jobs at t_o) that must be considered. This results in over-estimation of the cumulation of such "carry-in" work, which further causes pessimism in upper-bounding the amount of work that GEDF can be required to execute over $[t_o, d_{i,k})$. Therefore, to reduce such over-estimation, Baruah extended the analyzed interval to an earlier time instant such that the number of tasks with carry-in work can be bounded by $min(n, m-1)$. This significantly reduces the number of such tasks and thus reduces the pessimism in the resulting schedulability test.

GJDP scheduling. In 1996, Baruah *et al.* [11] proposed the first global multiprocessor real-time scheduling algorithm, namely *proportionate fair* (*pfair*), that is HRT optimal for implicit-deadline periodic tasks. Pfair scheduling is a quantum-driven scheduler, which significantly differs from most conventional event-driven real-time scheduling algorithms such as EDF and RM. Under Pfair scheduling, tasks are broken into quantum-length[8] subtasks and these subtasks are scheduled only at quantum boundaries. Later, Pfair scheduling was extended to more general task models including the sporadic task model [112, 113] and to implement a more efficient variant called the PD^2 scheduling algorithm [2].

[8]The time between integer multiples of scheduling quantum Q is called a *quantum*; the points in time separating quanta are called *quantum boundaries*.

2.3.1.3 Clustered Scheduling

Under global scheduling, tasks are scheduled from a single ready queue and may migrate across processors; in contrast, under partitioning, tasks are statically bound to processors and per-processor schedulers are used. Partitioning is susceptible to bin-packing-related schedulability limitations, which global approaches can avoid. Indeed, as discussed above, if bounded tardiness is the temporal constraint of interest, then global approaches can be preferable on multiprocessor platforms. However, the virtues of global scheduling come at the expense of higher runtime overheads because jobs are allowed to migrate among different processors. Clustered scheduling, which combines the advantages of both global and partitioned scheduling, has been suggested as a compromise [15, 27]. Under clustered scheduling, tasks are first partitioned onto clusters of processors, and intra-cluster scheduling is global. Specifically, m processors are split into $\frac{m}{c}$ clusters of c processors each, where c divides m. Tasks are first partitioned to clusters during an offline partitioning phase. At runtime, a global scheduler such as GEDF is used for each cluster. Note that different clusters can use different global scheduling algorithms. As a result, jobs do not migrate across clusters but can migrate within their assigned clusters. In this dissertation, we design a new clustered scheduling algorithm to schedule task graphs in a distributed real-time system, as will be discussed in Section 5.2.

2.3.2 Real-Time Self-Suspending Task Scheduling

The general problem of dealing with self-suspensions in real-time systems is known to be hard. In the following sections, we first review several well-known negative results showing the hardness of this problem. Then we describe existing related work in the context of uniprocessor and multiprocessor scheduling.

2.3.2.1 Negative Results

Ridouard *at el.* [106] showed that well-known scheduling policies such as EDF and RM are not efficient for scheduling HRT self-suspending task systems on an uniprocessor, even under the simple non-recurrent self-suspending task model (presented in Section 2.1.2). Negative results on the complexity and the competitive analysis (defined later in this section) in this context were also shown by them. Note that all the negative results on dealing with self-suspensions discussed in this section (Section 2.3.2.1) were presented in [106].

Complexity. The feasibility problem of scheduling non-recurrent real-time tasks is shown to be NP-hard in the strong sense. The proof is by transforming the problem from 3-Partition, which is known to be strongly NP-Complete. Moreover, it is shown that there is no *universal* scheduling algorithm[9] that can successfully schedule real-time self-suspending tasks, unless *P* = *NP*.

Negative results for scheduling self-suspending tasks under well-known algorithms. It is also shown in [106] that several well-known scheduling algorithms, which are competitive (defined below) for non-recurrent task systems (defined in Section 2.1.1), are not competitive for non-recurrent self-suspending task systems (defined in Section 2.1.4), as discussed below.

A common way to compare the schedulability of different scheduling algorithms is by conducting *competitive analysis*, which can determine the performance guarantee of an on-line algorithm. The strategy is to compare a given algorithm to an optimal *clairvoyant* algorithm, often called the *adversary*. An algorithm that minimizes an objective function is said to be c-competitive if the value of the targeted objective function obtained by the algorithm is no greater than c times the optimal value obtained by the adversary. An algorithm is *competitive* if there exists a constant c so that the algorithm is c-competitive.

For non-recurrent task systems without suspensions, there does not exist any competitive algorithms, but competitive algorithms exist for special cases. Baruah *et al.* [13] showed that no competitive on-line scheduling algorithm exists that maximizes job completions for uniprocessor systems. But for special cases, the following positive result given in Theorem 2.2 exist for ordinary task systems without suspensions.

Definition 2.6. A non-recurrent task system is said to be monotonic absolute deadline (MAD) if $d_{i,1} < d_{j,1}$ holds for any two tasks τ_i and τ_j in the system where $r_{i,1} < r_{j,1}$.

Definition 2.7. *Shortest remaining processing time first* (SRPTF) is an online algorithm that allocates the processor at any time to the task that has the shortest remaining processing time.

[9]A scheduling algorithm is said to be universal if it makes each scheduling decision in polynomial time (with respect to the length of the input).

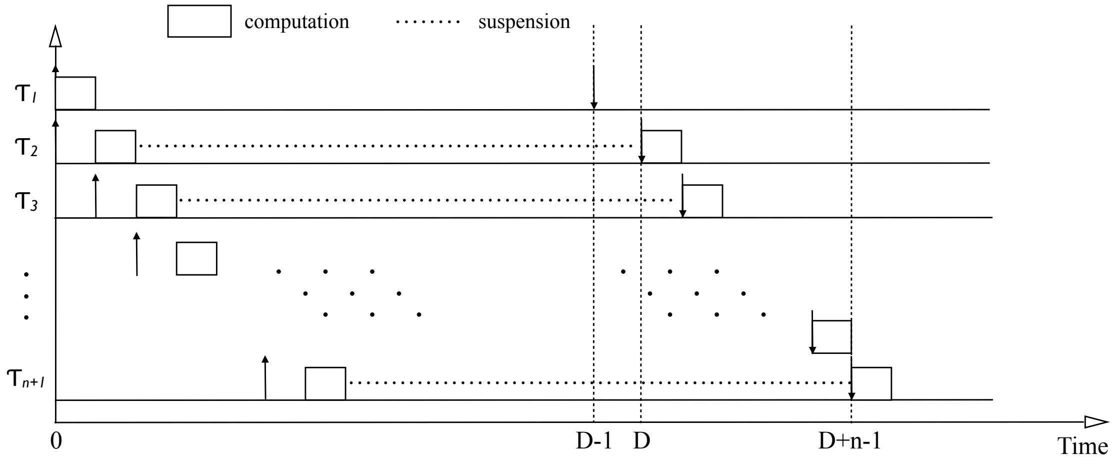
Theorem 2.2. *[13] For non-recurrent task systems with the MAD property, SRPTF is 2-competitive to minimize the number of tardy tasks. Moreover, SRPTF is the best possible on-line algorithm.*

Unfortunately, it is shown in [106] that if a task is allowed to self-suspend just once, then this result no longer holds. That is, SRPTF is not competitive to minimize the number of tardy tasks if each task is allowed to self-suspend at most once. This is proved by showing a counterexample, as illustrated in Figure 2.13. Using the same counterexample, this negative result for SRPTF can be extended for EDF, RM, LLF, and DM. That is, the scheduling algorithms EDF, RM, DM, and LLF are not competitive to minimize the number of tardy tasks if each non-recurrent task is allowed to self-suspend at most once.

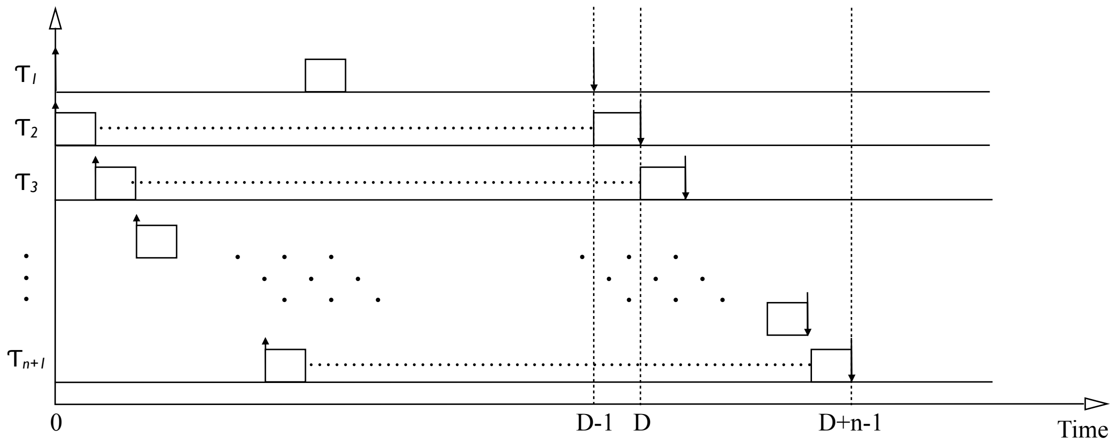
Negative results with respect to resource augmentation. The technique of *resource augmentation* is often used to improve the competitive ratio by giving a faster processor to the online algorithm while the adversary still runs on a unit speed processor. In [100], it is proven that for non-recurrent task systems, EDF is still optimal even under overloaded conditions if it is run on a two-speed processor while the optimal algorithm runs on a unit speed processor. In other words, EDF has a resource augmentation bound of 2 under overloaded conditions. However, if tasks are allowed to self-suspend at most once, then EDF cannot define a feasible schedule even under a k-speed processor, where k can be an arbitrarily large constant, while there exists a feasible schedule under a unit speed processor. This is proved in [106] by showing another counterexample, as illustrated in Figure 2.14. Therefore, allocating faster processors does not help to define a simple on-line scheduling policy for the simple self-suspending task model.

2.3.2.2 Self-Suspending Task Scheduling on Uniprocessors

In work pertaining to supporting self-suspending tasks on uniprocessors (and by extension multiprocessors scheduled via partitioning), the common suspension-oblivious analysis technique treats all suspensions as computation [86]. That is, when a task self-suspends, the processor is deemed to be busy as if the task executed. Thus, by treating all suspensions as computation, a sporadic self-suspending task system can be transformed into a sporadic task system with no suspensions. The downside of this approach is that it may cause severe capacity loss.



(a) SRPTF



(b) Optimal scheduler

Figure 2.13: A counterexample showing that SRPTF is not competitive for the non-recurrent self-suspending task model. The example task set contains $n+1$ task. The first task τ_1 releases a job that arrives at time 0 and has a deadline at time $D-1$, where D is an arbitrary large number. It contains only one computation phase of one time unit. For each other task τ_i $(2 \leq i \leq n)$, it releases a job that has a release time at $i-1$, an absolute deadline at $D+i-2$. Each job τ_i first executes for one time unit, then self-suspends for $D-2$ time units, and finally executes for another time unit. As seen in inset **(a)**, n out of $n+1$ deadlines are missed under SRPTF, while all deadlines are met under an optimal scheduling algorithm. If D becomes large enough, this task set has an arbitrarily small utilization. Therefore, we have an instance with an arbitrarily small utilization such that SRPTF is not competitive to minimize the number of tardy tasks if they are allowed to self-suspend at most once. Nevertheless, an optimal schedule exists for this tast set, as shown in inset **(b)**.

Several HRT schedulability tests have been presented for analyzing periodic tasks that may self-suspend at most once on uniprocessors. Tests presented in [65, 96, 97, 101, 114] involve straightforward execution control mechanisms, which modify task deadlines. For instance, a self-suspending task that suspends once can be divided into two subtasks with appropriately shorted deadlines and modified release times. Such a method is able to transform a sporadic self-suspending task into two independent periodic subtasks without any suspension. As long as we can meet the modified deadlines of these subtasks, we guarantee meeting the deadline of the original self-suspending task. This approach is often known as the *end-to-end* approach [86].[10] This approach unfortunately suffers from significant capacity loss due to the artificial shortening of deadlines.

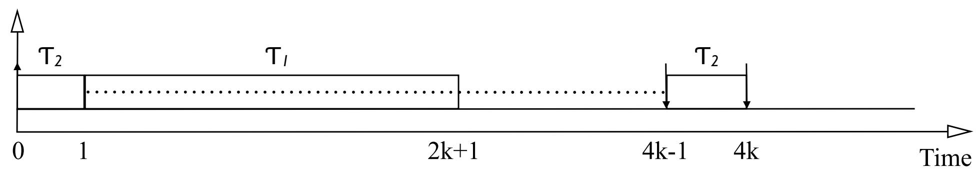
For TFP task systems, several tests have been derived based upon the computation of worst-case response times by Kim *et al.* [65], Liu [86], and Lakshmanan *et al.* [71]. Note that same techniques presented in [86] can also be applied for EDF scheduling [40].

In [65], the end-to-end approach as discussed above is used. Periodic tasks that may self-suspend at most once are considered in this work. Under this approach, each self-suspending task τ_i is divided into two independent subtasks τ_i^1 and τ_i^2 with modified deadlines and release offsets. Then a worst-case response time is derived for each subtask, denoted R_i^1 and R_i^2, using uniprocessor RTA under either EDF or TFP scheduling as described in Section 2.3.1.1, respectively. The given self-suspending task system τ is HRT schedulable if for each task $\tau_i \in \tau$, $R_i^1 + s_i + R_i^2 \leq d_i$ holds, where s_i and d_i denote the suspension length and the relative deadline of τ_i, respectively. Similar techniques of viewing suspensions as release offsets have been applied in by Palencia and Harbour in [97, 96].

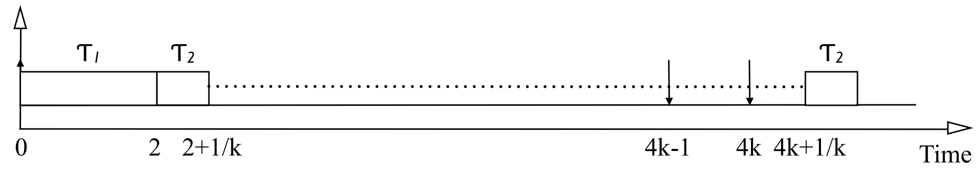
In [86], a schedulability test is derived by calculating the blocking time due to self-suspension and higher-priority tasks.[11] This approach first calculates the extra delay suffered by a self-suspending task τ_i due to its own self-suspension and the suspension of higher-priority tasks. This delay is called the *blocking time* of τ_i. The first component of the blocking time of τ_i is due to its own suspension, which is clearly no more than s_i. Moreover, the blocking time due to the suspension

[10]The name of this scheduling approach originates from the classical end-to-end principle of designing computer networks, which states that if application-specific functions can be implemented completely and correctly in the end hosts of a network, then they should reside in the end hosts rather than in intermediate nodes. Similarly, the end-to-end scheduling approach ensures the deadline of a task to be met by meeting deadlines of all intermediate subtasks.

[11]The simple self-suspending task model is also assumed here where each task may self-suspend at most once.



(a) A feasible schedule under a 1-speed processor



(b) An EDF schedule under a k-speed processor

Figure 2.14: A counterexample showing that EDF is not optimal for scheduling non-recurrent suspending task systems even with a k-speed processor, for any positive integer k. The non-recurrent task set contains two tasks that release jobs both arriving at time 0. τ_1 executes for $2k$ time units, and has an absolute deadline at $4k-1$. τ_2 first executes for one time unit, then self-suspend for $4k-2$ time units, and finally executes for another time unit. τ_2 has an absolute deadline at $4k$. Inset **(a)** shows a feasible schedule of this task set under a 1-speed processor. Inset **(b)** shows the corresponding EDF schedule, but under a k-speed processor, where k can be arbitrarily large. As seen, the deadline of τ_2 is missed under EDF even with such a k-speed processor due to self-suspensions.

of a higher-priority task τ_k depends upon two cases: (*i*) τ_i cannot be delayed by τ_k for more than e_k because τ_i can be scheduled (or partially scheduled) during the suspension of τ_k, and (*ii*) the blocking time cannot be more than s_k if $s_k < e_k$. Therefore, the blocking time of τ_i due to each higher-priority task τ_k is at most $min(e_k, s_k)$. Finally, the total blocking time of τ_i can be obtained by $e_i + \sum_{k=1}^{i-1} min(e_k, s_k)$. After obtaining this extra blocking time due to tasks' self-suspensions, standard RTA can be then applied with the consideration of this blocking term. The same technique of obtaining the blocking time due to self-suspensions has been applied in [40] for EDF scheduling.

In [71], Lakshmanan and Rajkumar consider the general sporadic self-suspending task model (defined in Section 2.1.4). It is shown by them that the critical scheduling instant characterization is easier in the context of sporadic real-time tasks than periodic ones. A *critical scheduling instant* of self-suspending tasks is an instance of job releases that results in the worst-case interference from higher-priority non-suspending tasks. Note that this critical scheduling instant is with respect to a system where there is only one self-suspending task.[12] With this assumption, a pseudo-polynomial exact-case response-time test is then derived based upon the identified critical scheduling instant. Then, the authors prove that the interference from self-suspending tasks on lower-priority tasks is no worse than their non-suspending counterparts (with all suspension phases removed). Therefore, the obtained pseudo-polynomial response-time test can also be applied to general task systems that may contain multiple self-suspending tasks.

Our contributions. To the best of our knowledge, globally-scheduled multiprocessor systems that allow suspensions to be expressed have not been considered before. Thus, how to support real-time self-suspending task systems on multiprocessors under global scheduling was an open problem. As mentioned Section 2.3.1.2, for SRT sporadic multiprocessor task systems, global scheduling can ensure bounded tardiness with no ulitzation loss. Thus, global scheduling algorithms have the potential of also efficiently scheduling real-time self-suspending task systems on multiprocessors. Motivated by this, our contributions made in this dissertation with respect to the suspension problem is the following: we derive a set of multiprocessor schedulability tests for globally-scheduled HRT and SRT sporadic self-suspending task systems.

2.3.3 Real-Time Task Graph Scheduling

In this dissertation, we consider a particularly expressive directed-acyclic-graph-based (DAG-based) formalism, the processing graph method (PGM) [69]. To our knowledge, PGM has not been considered before in the context of global real-time scheduling algorithms. However, the issue of scheduling PGM graphs on a uniprocessor was extensively considered by Goddard in his dissertation [51]. Since his work inspired the research done on multiprocessor PGM graph scheduling in this dissertation, we provide a detailed review on Goddard's work in the following.

2.3.3.1 Uniprocessor PGM Scheduling

Among models that allow the DAG-based precedence constraints to be expressed, the processing graph method (PGM) [69] is of great interest and importance. First developed by the U.S. Navy, PGM has been widely used in many signal processing applications such as radar sensing [51, 69].

[12]The task system considered in [71] is assumed to have only one self-suspending task, with the rest being ordinary sporadic tasks with no suspensions.

PGM generalizes common forms of DAG-based dependencies and considers data communications among nodes (as discussed in Section 2.3.3).

HRT PGM scheduling on uniprocessors. The issue of scheduling HRT PGM task systems on a uniprocessor was extensively considered by Goddard in his dissertation [51]. A major emphasis of his dissertation is schedulability analysis pertaining to uniprocessor PGM task systems. To derive such analysis, Goddard applied a two-phase approach. First, according to the PGM specification [69], non-source nodes in a PGM graph execute following a *rate-based* pattern (as defined in Section 2.1.5), regardless of whether the source node executes according to a rate-based pattern (i.e., the source node could execute according to periodic or sporadic pattern). Goddard presented techniques for computing the execution rates (as defined below) for nodes in a PGM graph. Second, he presented conditions for verifying the schedulability of the resulting task set under a rate-based, EDF scheduler.

More specifically, Goddard first showed that according to PGM, the execution relationship between connected nodes in a PGM graph can be specified by *execution rates* that represent a rate-based execution pattern (defined in Section 2.1.5). Moreover, in order to avoid data loss, an execution of a node in a PGM graph must be *valid*. An execution of a node is valid iff (*i*) the corresponding task executes only when it is eligible for execution and no two executions of the same node overlap, (*ii*) each input queue has its tokens atomically consumed after each output queue has its tokens atomically produced, and (*iii*) tokens are produced at most once on an output queue during each node execution. An execution of a PGM graph is *valid* iff all of its nodes have valid execution rates and no data loss occurs.

After computing the execution rates for PGM nodes, Goddard further showed that a PGM graph can be implemented as a rate-based DAG (defined in Section 2.1.6). After implementing PGM graphs in a given PGM task system as RBE tasks, Goddard then presented conditions for verifying the schedulability of the resulting task set under a rate-based, EDF scheduler. Specifically, prior schedulability analysis pertaining to the RBE task model [61] was directly applied to derive schedulability conditions. This schedulability analysis established a schedulability condition for an RBE task set that can be evaluated in pseudo-polynomial time if the total utilization of the task set does not exceed one. Moreover, for the special case where $d_i = y_i$ holds for every RBE task τ_i, the schedulability condition evaluation can be reduced to the polynomial-time schedulability condition $\sum_{i=1}^{n} \frac{x_i}{y_i} \cdot e_i \leq 1$.

2.3.3.2 Other Prior Work on Real-Time Task Graph Scheduling

There are two processing graph models that are similar to PGM: the *synchronous dataflow* (SDF) *graph model* [73] and the *logical application stream model* (LASM) [29, 30]. Although we focus on PGM graphs in this dissertation, our results can also be applied to these graph models.

SDF. The SDF graph model was first proposed by Lee and Messerschmitt [73] to develop signal processing applications. The SDF graph model is in fact a subset of the PGM graph model. The main difference between PGM graphs and SDF graphs is that a queue's threshold value must equal its consume value in an SDF graph, whereas PGM only requires that the consume amount must be at most the threshold. For uniprocessor platforms, several efforts [22, 73, 107, 108, 119] have been made to minimize memory usage by creating various online scheduling algorithms. The online algorithms create static node execution schedules that are executed periodically by the uniprocessor. For TFP scheduling on a uniprocessor, Parks and Lee [98] studied the applicability of non-preemptive RM scheduling to SDF graphs. For multiprocessor platforms, Bekooij *et al.* presented a dataflow analysis based upon time division multiplexing (TDM) scheduling on applications modeled as SDF graphs [16]. To improve upon [16], Moreira *et al.* developed a resource allocation heuristic [90] and a TDM scheduler combined with static allocation policies [89, 91].

LASM. The LASM was first proposed by Chatterjee and Strosnider [29, 30] to develop multimedia applications. It is remarkably similar to PGM, although it was developed independently ten years after PGM was developed. The main difference between LASM and PGM is due to different node execution models. LASM requires a node to execute periodically, which is a reasonable requirement for multimedia applications but adds latency (compared to the RBE model) to a signal in a signal processing application. The work on PGM graphs [51, 69] improves on the analysis of LASM graphs by not requiring periodic execution of the nodes in the graph. The adopted rate-based graph execution model can more accurately predict processor demand without imposing the additional latency created by periodic node execution.

Graph scheduling in distributed systems. The scheduling of real-time graphs in distributed systems (which must be scheduled by partitioning approaches) has also been considered. For example, Jayachandran and Abdelzaher have presented delay composition rules that provide a bound on the end-to-end delay of jobs in partitioned distributed systems that include DAGs [59] or graphs with cyclic precedence constraints [58]. These rules permit a graph system to be transformed so that uniprocessor schedulability analysis can be applied. Offline schedulability tests have also been proposed by Palencia and Pellizzoni for scheduling tasks with precedence constraints in distributed real-time systems comprised of periodic tasks [95, 99]. Several schedulability tests have also been presented by by Kao and Zhang to divide the end-to-end deadline of tasks into per-stage deadlines such that uniprocessor schedulability tests can then be applied to determine if each stage is schedulable [64, 118].

Our contributions. To the best of our knowledge, globally-scheduled multiprocessor systems that allow DAG-based precedence constraints to be expressed have not been considered before. Thus, how to support real-time DAG-based task systems on multiprocessors under global scheduling was an open problem. Our contributions in dissertation is to support SRT PGM task graphs on multiprocessors using global scheduling algorithms and distributed systems using clustered schedulers.

2.3.4 Dealing with Non-Preemptive Sections

In this dissertation, we consider a general form of non-preemptive section where a job that is executing its non-preemptive section cannot be preempted by any other task (even with higher priority). Thus, jobs with higher priorities can be blocked by lower-priority jobs due to non-preemptive sections. A common approach [86] of dealing with such non-preemptive sections is to first compute the worst-case blocking time that each task can experience. Then, such blocking times can be accommodate into tasks' WCETs as well as the corresponding schedulability tests.

A special form of non-preemptive section, called non-preemptive critical section that is often due to resource sharing, has received much attention. When a job $\tau_{i,k}$ is executing within its non-preemptive critical section due to accessing a resource A, it blocks jobs with higher priorities that require accesses to the same resource A. However, $\tau_{i,k}$ can be preempted by jobs with higher priorities that do not require accessing A. This property may cause "unbounded" blocking time as noted in [86, 110]. Various strategies [6, 23, 25, 26, 60, 72, 86, 110] have been proposed to control the length of the blocking time due to non-preemptive critical sections. The fundamental idea behind all these strategies is to change the priority of the lower-priority job, which holds the resource blocking other higher-priority jobs. The difference between them lies in when the priority is changed and to which level. Under all strategies the changed priority is switched back at the end of the critical section.

Regarding the general form of non-preemptive section we consider in this dissertation, as discussed in Section 2.3.1.2, bounded tardiness can be achieved under non-preemptive GEDF for any sporadic task system with total utilization up to m scheduled on m processors [41].

Our contributions. For real-time task systems with non-preemptive sections and other types of complex runtime behaviors such as self-suspensions and precedence constraints, no obvious solution exists. As the first attempt, we consider the issue of scheduling SRT task systems in which mixed types of complex runtime behaviors may exist due to non-preemptive sections, self-suspensions, and graph-based precedence constraints. We present a transformation process that converts such a sophisticated task system into a simpler system with only self-suspensions. In the simpler system, each task's maximum job response time is at least that of the original system. This result allows tardiness bounds to be established by focusing only on the impacts of suspensions.

2.3.5 Real-Time Parallel Task Scheduling

Scheduling non-real-time parallel applications is a deeply explored topic [31, 36, 48, 49, 85, 115, 117]. However, in most (if not all) prior work on this topic, including all of the just-cited work, scheduling decisions are made on a best-effort basis, so none of these results can provide performance guarantees such as response time bounds.

Regarding scheduling HRT parallel task systems, Lakshmanan *et al.* proposed a scheduling technique for the *fork-join* model, where a parallel task is a sequence of segments, alternating between sequential and parallel phases [70]. A sequential phase contains only one thread while a parallel phase contains multiple threads that can be executed concurrently on different processors. In their model, all parallel phases are assumed to have the same number of parallel threads, which must be no greater than the number of processors. Also, all threads in any parallel segment must have the same execution cost. The authors derived a resource augmentation bound of 3.42 under DM scheduling.

In [109], Saifullah et al. extended the fork-join model so that each parallel phase can have a different number of threads and threads can have different execution costs. The authors proposed an approach that transforms each periodic parallel task into a number of ordinary constrained-deadline periodic tasks by creating per-segment intermediate deadlines. They also showed that resource augmentation bounds of 2.62 and 3.42 can be achieved under GEDF and PDM scheduling, respectively. In [92], Nelissen et al. proposed techniques that optimize the number of processors needed to schedule sporadic parallel tasks. The authors also proved that the proposed techniques achieve a resource augmentation bound of 2.0 under scheduling algorithms such as U-EDF [93] and PD^2 [111].

Our contributions. Although the problem of scheduling HRT parallel tasks has received much more attention, achieving HRT correctness for many parallel applications in practice is overkill. For applications such as real-time parallel video and image processing applications [5, 43] and computer vision applications such as collision detection and feature tracking [66], fast and bounded response times for individual video frames are important, to ensure smooth video output. Motivated by this observation, in this dissertation, we focus on supporting SRT parallel task systems on multiprocessors with bounded response times.

2.4 Summary

In this chapter, we reviewed some prior work on the real-time scheduling of task systems containing complex runtime behaviors. Specifically, we reviewed prior work on handling each of our considered runtime behaviors: self-suspensions in Section 2.3.2, graph-based precedence constraints in Section 2.3.3, non-preemptive sections in Section 2.3.4, and parallel execution segments in Section 2.3.5. From the review presented in this chapter, it is quite evident that the problem of dealing with complex runtime behaviors in multiprocessor real-time systems has received limited attention. In the following chapters, we present our results that contribute to fill this research vacancy.

CHAPTER 3

Scheduling SRT Self-Suspending Tasks[1]

In this chapter, we present multiprocessor schedulability tests for SRT self-suspending task systems. We present a suspension-aware schedulability test, a technique that can further improve this test, and another improved schedulability test for SRT self-suspending task systems, in Section 3.2, 3.3, and 3.4, respectively. First, we provided needed definitions.

3.1 System Model

We consider the problem of scheduling a set $\tau = \{\tau_1, ..., \tau_n\}$ of n SRT (defined in Section 1.2.2) sporadic self-suspending tasks (defined in Section 2.1.4) on $m \geq 1$ identical processors $M_1, M_2, ..., M_m$.

A common case for real-time workloads is that both self-suspending tasks and computational tasks (which do not suspend) co-exist. To reflect this, we let U^s_{sum} denote the total utilization of all self-suspending tasks, and U^c_{sum} denote the total utilization of all computational tasks.

So that our analysis can be more accurately applied in settings where a task's total suspension time varies from job to job, we assume that a fixed parameter H ($H \geq 1$) is specified and that S_i^H denotes the maximum total self-suspension length for any H ($H \geq 1$) consecutive jobs of task τ_i. Note that if $H = 1$, then a maximum per-job total suspension length is being assumed.

[1]Contents of this chapter previously appeared in preliminary form in the following papers:
Cong Liu and James Anderson. An $O(m)$ Analysis Technique for Supporting Real-Time Self-Suspending Task Systems, Proceedings of the 33th IEEE Real-Time Systems Symposium, pages 373-382, 2012.
Cong Liu and James Anderson. A New Technique for Analyzing Soft Real-Time Self-Suspending Task Systems, pages 29-32, ACM SIGBED Review, 2012.
Cong Liu and James Anderson. Improving the Schedulability of Sporadic Self-Suspending Soft Real-Time Multiprocessor Task Systems, Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pages 13-22, 2010.
Cong Liu and James Anderson. Task Scheduling with Self-Suspensions in Soft Real-Time Multiprocessor Systems, Proceedings of the 30th IEEE Real-Time Systems Symposium, pages 425-436, 2009.

As discussed in Chapter 2, under GEDF (GFIFO), released jobs are prioritized by their deadlines (release times). So that our results can be applied to both algorithms, we consider a generic scheduling algorithm (GSA) where each job is prioritized by some time point between its release time and deadline. Specifically, for any job $\tau_{i,j}$, we define a priority value with lower values denoting higher priority: $\rho_{i,j} = r_{i,j} + \kappa \cdot p_i$, where $0 \leq \kappa \leq 1$. We assume that ties are broken by task ID. Note that GEDF and GFIFO are special cases of GSA where κ is set to 1 and 0, respectively.

For simplicity, we henceforth assume that each job of any task τ_l executes for *exactly* e_l time units. By Claim 1 provided below, any tardiness bound derived for such systems applies to other systems as well. (Claim 1 also considers GTFP scheduling because GTFP scheduling is considered in Chapter 4 for HRT sporadic self-suspending task systems.)

We say that a sporadic task system τ is *concrete* if the release time (and hence deadline) and actual execution cost and suspension time of every job of each task is fixed. Two concrete task systems are *compatible* if they have the same jobs with the same release times (they can have different actual execution and suspension times). A concrete task system τ is *maximal* if the actual execution time of any job equals the corresponding task's WCET.

Claim 1. For any concrete task system τ, there exists a compatible maximal concrete task system τ' such that, for any job $\tau_{i,k}$, its response time in the GSA (or GTFP) schedule for τ' is at least its response time in the GSA (or GTFP) schedule for τ.

Proof. The existence of the desired maximal concrete system is demonstrated via a construction method in which computation phases are ranked as follows: **(i)** if $\tau_{i,k} \prec \tau_{x,y}$ (i.e., $\tau_{i,k}$ has higher priority than $\tau_{x,y}$, as defined in Chapter 2), then all computation phases of $\tau_{i,k}$ are ranked before all computation phases of $\tau_{x,y}$; **(ii)** earlier computation phases of $\tau_{i,k}$ are ranked before later computation phases of $\tau_{i,k}$.

Consider a computation phase C of a job $\tau_{i,k}$ that is not maximal. We show that the length of C can be increased by one time unit by adding to the end of C a piece of computation ρ of length one time unit (recall from Chapter 2 that we assume that time is integral). In doing so, it may be necessary to reduce the length of a lower-ranked computation phase by one time unit and to reduce the length of a subsequent suspension phase (if any) of $\tau_{i,k}$. By inducting over all computation phases in rank order, and by iteratively increasing any non-maximal execution time by one time unit, we can obtain a compatible concrete task system that is maximal. The construction method will ensure that no job's response time is reduced.

Let $[t, t+1)$ denote the time interval where ρ should be added to the schedule (according to GSA or GTFP). If, before adding ρ, task τ_i is scheduled within $[t, t+1)$, then the computation phase of τ_i executing at that time, call it C', is ranked lower than C. In this case, we can accommodate ρ by reducing C' in length by one time unit.[2] If C and C' are separated by a suspension phase, then the length of that suspension phase must be defined to be zero.

In the rest of the proof, we consider the other possibility: before adding ρ, τ_i is *not* scheduled within $[t, t+1)$ (and hence, it is not scheduled in $[t', t+1)$, where t' is the completion time of C). In this case, if there is an idle processor in $[t, t+1)$, then ρ can be scheduled there without modifying the length of any lower-ranked computation phase. On the other hand, if there is no idle processor, then, as ρ should be scheduled in $[t, t+1)$, there must be a computation phase ranked lower than C scheduled then. We can accommodate ρ and allow it to be scheduled in $[t, t+1)$ by reducing the length of that lower-ranked computation phase by one time unit. If C is followed by a suspension phase, then, once ρ has been added to the schedule, it may be necessary to reduce the length of that suspension phase. In particular, if, before adding ρ, $\tau_{i,k}$ was suspended in $[t, t+1)$, then the length of that suspension phase must be reduced so that it starts as $t+1$.

Note that the construction method used in this proof strongly exploits the fact that, in our task model, suspension phases are upper-bounded, and hence, can be reduced. $\square$

3.2 First SRT Schedulability Test

We now derive a SRT schedulability test and thus a tardiness bound for GSA by comparing the allocations to a task system τ in a processor sharing (PS) schedule (defined in Definition 2.5) and an actual GSA schedule of interest for τ, both on m processors, and quantifying the difference between the two. We analyze task allocations on a per-task basis. Our analysis draws inspiration from the seminal work of Devi and Anderson [41], and follows the same general framework.

[2]If C' is of length one time unit and is followed by a suspension phase, then we can avoid altering the length of that suspension phase by assuming that C' executes for zero time at time $t+1$. Note that C''s execution time will be increased in a subsequent induction step. A similar comment applies to the argument in the next paragraph.

Our tardiness-bound derivation focuses on a given task system τ. We order the jobs in τ based on their priorities: $\tau_{i,v} \prec \tau_{a,b}$ iff $\rho_{i,v} < \rho_{a,b}$ or $(\rho_{i,v} = \rho_{a,b}) \wedge (i < a)$. Let $\tau_{l,j}$ be a job of a task τ_l in τ, $t_d = d_{l,j}$, and S be a GSA schedule for τ with the following property.

(P) The tardiness of every job $\tau_{i,k}$ such that $\tau_{i,k} \prec \tau_{l,j}$ is at most $x + e_i + s_i$ in S, where $x \geq 0$.

Our objective is to determine the smallest x such that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$. This would by induction imply a tardiness of at most $x + e_i + s_i$ for all jobs of every task τ_i, where $\tau_i \in \tau$. We assume that $\tau_{l,j}$ finishes after t_d, for otherwise, its tardiness is trivially zero. The steps for determining the value for x are as follows.

1. Determine an upper bound on the work pending for tasks in τ that can compete with $\tau_{l,j}$ after t_d. This is dealt with in Lemmas 3.1–3.3 in Section 3.2.1.

2. Determine a lower bound on the amount of work pending for tasks in τ that can compete with $\tau_{l,j}$ after t_d, required for the tardiness of $\tau_{l,j}$ to exceed $x + e_l + s_l$. This is dealt with in Lemma 3.4 in Section 3.2.2.

3. Determine the smallest x such that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$, using the above upper and lower bounds. This is dealt with in Theorem 3.1 in Section 3.2.3.

Definition 3.1. We categorize jobs based on the relationship between their priorities and deadlines and those of $\tau_{l,j}$:

$$\mathbf{d} = \{\tau_{i,v} : (\tau_{i,v} \preceq \tau_{l,j}) \wedge (d_{i,v} \leq t_d)\};$$

$$\mathbf{D} = \{\tau_{i,v} : (\tau_{i,v} \prec \tau_{l,j}) \wedge (d_{i,v} > t_d)\}.$$

d is the set of jobs with deadlines at most t_d with priority at least that of $\tau_{l,j}$. These jobs do not execute beyond t_d in the PS schedule. Note that $\tau_{l,j}$ is in **d**. **D** is the set of jobs that have higher priorities than $\tau_{l,j}$ and deadlines greater than t_d. Note that jobs not in $\mathbf{d} \cup \mathbf{D}$ have lower priority than those in $\mathbf{d} \cup \mathbf{D}$ and thus do not affect the scheduling of jobs in $\mathbf{d} \cup \mathbf{D}$. For simplicity, we will henceforth assume that no job not in $\mathbf{d} \cup \mathbf{D}$ executes in either the PS or GSA schedule. Let D_H be the set of tasks with jobs in **D**. **D** consists of *carry-in jobs*, which have a release time before t_d and a deadline after t_d. Exactly one such job exists for each task in D_H. (Note that **D** is empty under GEDF because jobs with later deadlines have lower priorities.)

Definition 3.2. A time instant t is *busy* for a job set J if all m processors execute a job in J at t. A time interval is busy for J if each instant within it is busy for J.

The following claim follows from the definition of LAG (the concepts of lag and LAG are presented in Section 2.2).

Claim 2. If $LAG(\mathbf{d}, t_2, S) > LAG(\mathbf{d}, t_1, S)$, where $t_2 > t_1$, then $[t_1, t_2)$ is non-busy for **d**. In other words, *LAG* for **d** can increase only throughout a non-busy interval.

An interval could be non-busy for **d** for two reasons:

1. There are not enough enabled non-suspended jobs in **d** to occupy all available processors. Such an interval is called *non-busy non-displacing*.

2. Jobs in **D** occupy one or more processors and there are enabled non-suspended jobs in **d** that are not scheduled. Such an interval is called *non-busy displacing*.

Definition 3.3. Let δ_k be the amount of execution time consumed by a carry-in job $\tau_{k,v}$ by time t_d.

Definition 3.4. Let $B(\mathbf{D}, t_d, S)$ be the amount of work due to jobs in **D** that can compete with $\tau_{l,j}$ after t_d.

Since $\mathbf{d} \cup \mathbf{D}$ includes all jobs of higher priority than $\tau_{l,j}$, the competing work for $\tau_{l,j}$ is given by the sum of **(i)** the amount of work pending at t_d for jobs in **d**, and **(ii)** the amount of work $B(\mathbf{D}, t_d, S)$ demanded by jobs in **D** that competes with $\tau_{l,j}$ after t_d. Since jobs from **d** have deadlines at most t_d, they do not execute in the PS schedule beyond t_d. Thus, the work pending for them is given by $LAG(\mathbf{d}, t_d, S)$. Therefore, the competing work for $\tau_{l,j}$ after t_d is given by $LAG(\mathbf{d}, t_d, S) + B(\mathbf{D}, t_d, S)$. Let

$$Z = LAG(\mathbf{d}, t_d, S) + B(\mathbf{D}, t_d, S). \tag{3.1}$$

3.2.1 Upper Bound

In this section, we determine an upper bound on Z.

Definition 3.5. Let t_n be the end of the latest non-busy non-displacing interval for **d** before t_d, if any; otherwise, $t_n = 0$.

The following two lemmas have been proved previously for both GEDF [41] and GFIFO [78] for ordinary sporadic task systems without self-suspensions. Note that the value of $LAG(\mathbf{d}, t_d, S) + B(\mathbf{D}, t_d, S)$ depends only on allocations in the PS schedule PS and allocations to jobs in $\mathbf{d} \cup \mathbf{D}$ in the actual schedule S by time t_d. The PS schedule is not impacted by self-suspensions. Also, Property (P) alone is sufficient for determining how much work any job in $\mathbf{d} \cup \mathbf{D}$ other than $\tau_{l,j}$ completes before t_d. For these reasons, Lemmas 3.1 and 3.2 continue to hold for task systems with self-suspensions.

Lemma 3.1. *$LAG(\boldsymbol{d}, t_d, S) \leq LAG(\boldsymbol{d}, t_n, S) + \sum_{\tau_k \in D_H} \delta_k(1 - u_k)$, where $t \in [0, t_d]$.*

Proof. By (2.5), we have

$$\begin{aligned} LAG\left(\mathbf{d}, t_d, S\right) \quad \leq \quad & LAG(\mathbf{d}, t_n, S) + A\left(\mathbf{d}, t_n, t_d, PS\right) \\ & -A\left(\mathbf{d}, t_n, t_d, S\right). \qquad (3.2) \end{aligned}$$

We split $[t_n, t_d)$ into z non-overlapping intervals $[t_{p_i}, t_{q_i}), 1 \leq i \leq z$, such that $t_n = t_{p_1}, t_{q_{i-1}} = t_{p_i}$, and $t_{q_z} = t_d$. Each interval $[t_{p_i}, t_{q_i})$ is either busy or non-busy displacing for $\mathbf{d}$, by the selection of t_n. We assume that the intervals are defined so that for each non-busy displacing interval $[t_{p_i}, t_{q_i})$, if a task in D_H executes in $[t_{p_i}, t_{q_i})$ then it executes continuously throughout $[t_{p_i}, t_{q_i})$; we let α_i denote the set of such tasks.

We now bound the difference between the work performed in the PS schedule and the GSA schedule S across each of these intervals $[t_{p_i}, t_{q_i})$. The sum of these bounds will give us a bound on the total allocation difference throughout $[t_n, t_d)$. Depending on the nature of the interval $[t_{p_i}, t_{q_i})$, two cases are possible.

Case 1. $[t_{p_i}, t_{q_i})$ is busy. Since in S all processors are occupied by jobs in $\mathbf{d}$, we have $A(\mathbf{d}, t_{p_i}, t_{q_i}, PS) - A(\mathbf{d}, t_{p_i}, t_{q_i}, S) \leq U_{sum}(t_{q_i}, t_{p_i}) - m(t_{q_i} - t_{p_i}) \leq 0$.

Case 2. $[t_{p_i}, t_{q_i})$ is non-busy displacing. The cumulative utilization of all tasks $\tau_k \in \alpha_i$ is $\sum_{\tau_k \in \alpha_i} u_k$. The carry-in jobs of these tasks do not belong to $\mathbf{d}$, by the definition of $\mathbf{d}$. Therefore, the allocation of jobs in $\mathbf{d}$ during $[t_{p_i}, t_{q_i})$ in PS is $A(\mathbf{d}, t_{p_i}, t_{q_i}, PS) \leq (t_{q_i} - t_{p_i})(m - \sum_{\tau_k \in \alpha_i} u_k)$.

All processors are occupied at every time instant in the interval $[t_{p_i}, t_{q_i})$, because it is displacing. Thus, $A(\mathbf{d}, t_{p_i}, t_{q_i}, S) = (t_{q_i} - t_{p_i})(m - |\alpha_i|)$. Therefore, the allocation difference for jobs in **d** throughout the interval is

$$\begin{aligned}
& A(\mathbf{d}, t_{p_i}, t_{q_i}, PS) - A(\mathbf{d}, t_{p_i}, t_{q_i}, S) \\
\leq\ & (t_{q_i} - t_{p_i})\left((m - \sum_{\tau_k \in \alpha_i} u_k) - (m - |\alpha_i|)\right) \\
=\ & (t_{q_i} - t_{p_i}) \sum_{\tau_k \in \alpha_i} (1 - u_k)\,. \qquad (3.3)
\end{aligned}$$

For each task τ_k in D_H, the sum of the lengths of the intervals $[t_{p_i}, t_{q_i})$ in which the carry-in job of τ_k executes continuously is at most δ_k. Thus, summing the allocation differences for all the intervals $[t_{p_i}, t_{q_i})$ given by (3.3), we have

$$\begin{aligned}
& A(\mathbf{d}, t_n, t_d, PS) - A(\mathbf{d}, t_n, t_d, S) \\
\leq\ & \sum_{i=1}^{z} \sum_{\tau_k \in D_H} (t_{q_i} - t_{p_i})(1 - u_k) \\
\leq\ & \sum_{\tau_k \in D_H} \delta_k (1 - u_k). \qquad (3.4)
\end{aligned}$$

Setting this value into (3.2), we get $LAG(\mathbf{d}, t_d, S) \leq LAG(\mathbf{d}, t_n, S) + A(\mathbf{d}, t_n, t_d, PS) - A(\mathbf{d}, t_n, t_d, S) \leq LAG(\mathbf{d}, t_n, S) + \sum_{\tau_k \in D_H} \delta_k (1 - u_k)$. □

Lemma 3.2. *$lag(\tau_i, t, S) \leq u_i \cdot x + e_i + u_i \cdot s_i$ for any task τ_i and $t \in [0, t_d]$.*

Proof. Let $d_{i,k}$ be the deadline of the earliest pending job of τ_i, $\tau_{i,k}$, in the schedule S at time t. If such a job does not exist, then $lag(\tau_i, t, S) = 0$, and the lemma holds trivially. Let γ_i be the amount of work $\tau_{i,k}$ performs before t.

By the selection of $\tau_{i,k}$, we have

$$\begin{aligned}
lag(\tau_i, t, S) &= \sum_{h \geq k} lag(\tau_{i,h}, t, S) \\
&= \sum_{h \geq k} (A(\tau_{i,h}, 0, t, PS) - A(\tau_{i,h}, 0, t, S)).
\end{aligned}$$

Given that no job executes before its release time, $A(\tau_{i,h}, 0, t, S) = A(\tau_{i,h}, r_{i,h}, t, S)$. Thus,

$$\begin{aligned} lag(\tau_i, t, S) \quad = \quad & A(\tau_{i,k}, r_{i,h}, t, PS) - A(\tau_{i,k}, r_{i,k}, t, S) \\ & + \sum_{h>i} (A(\tau_{i,h}, r_{i,h}, t, PS) \\ & - A(\tau_{i,h}, r_{i,h}, t, S)). \end{aligned} \tag{3.5}$$

By the definition of PS, $A(\tau_{i,k}, r_{i,h}, t, PS) \leq e_i$, and $\sum_{h>k} A(\tau_{i,h}, r_{i,h}, t, PS) \leq u_i \cdot \max(0, t - d_{i,k})$. By the selection of $\tau_{i,k}$, $A(\tau_{i,k}, r_{i,k}, t, S) = \gamma_i$, and $\sum_{h>k} A(\tau_{i,h}, r_{i,h}, t, S) = 0$. By setting these values into (3.5), we have

$$lag(\tau_i, t, S) \leq e_i - \gamma_i + u_i \cdot \max(0, t - d_{i,k}). \tag{3.6}$$

There are two cases to consider.

Case 1. $d_{i,k} \geq t$. In this case, (3.6) implies $lag(\tau_i, t, S) \leq e_i - \gamma_i$, which implies $lag(\tau_i, t, S) \leq u_i \cdot x + e_i + u_i \cdot s_i$.

Case 2. $d_{i,k} < t$. In this case, because $t \leq t_d$ and $d_{l,j} = t_d$, $\tau_{i,k}$ is not the job $\tau_{l,j}$. Thus, by Property (P), $\tau_{i,k}$ has tardiness at most $x + e_i + s_i$, so $t + e_i - \gamma_i \leq d_{i,k} + x + e_i + s_i$. Thus, $t - d_{i,k} \leq x + \gamma_i + s_i$. Setting this value into (7.4), we have $lag(\tau_i, t, S) \leq u_i \cdot x + e_i + u_i \cdot s_i$. $\square$

Lemma 3.3 below upper bounds $LAG(\mathbf{d}, t_n, S)$.

Definition 3.6. Let E^s_{sum} be the total execution cost of all self-suspending tasks in τ. Let E_{sum} be the total execution cost of all tasks in τ. Let S^s_{sum} be the total suspension length of all tasks in τ. Let u^s_{max} be the maximum utilization of any self-suspending task in τ.

Definition 3.7. Let U^c_L be the sum of the $\min(m-1, c)$ largest computational task utilizations, where c is the number of computational tasks. Let E^c_L be the sum of the $\min(m-1, c)$ largest computational task execution costs.

Lemma 3.3. $LAG(\mathbf{d}, t_n, S) \leq (U^s_{sum} + U^c_L) \cdot x + E^s_{sum} + E^c_L + u^s_{max} \cdot S^s_{sum}$.

Proof. By summing individual task lags at t_n, we can bound $LAG(\mathbf{d}, t_n, S)$. If $t_n = 0$, then $LAG(\mathbf{d}, t_n, S) = 0$, so assume $t_n > 0$. Consider the set of tasks $\beta = \{\tau_i : \exists \tau_{i,v}$ in **d** such that $\tau_{i,v}$ is enabled at $t_n^-\}$. Given that the instant t_n^- is non-busy non-displacing, at most $m-1$ computational tasks in β have jobs executing at t_n^-. Due to suspensions, however, β may contain more than $m-1$ tasks. In the worst case, all suspending tasks in τ have a suspended enabled job at t_n^- and $\min(m-1, c)$ computational tasks have an enabled job executing at t_n^-. If task τ_i does not have an enabled job at t_n^-, then $lag(\tau_i, t_n, S) \leq 0$. Therefore, by (2.5), we have

$$\begin{aligned}
LAG(\mathbf{d}, t_n, S) &= \sum_{\tau_i : \tau_{i,v}^w \in \mathbf{d}} lag(\tau_i, t_n, S) \\
&\leq \sum_{\tau_i \in \beta} lag(\tau_i, t_n, S) \\
&\quad \{\text{by Lemma 3.2}\} \\
&\leq \sum_{\tau_i \in \beta} (u_i \cdot x + e_i + u_i \cdot s_i) \\
&\leq (U_{sum}^s + U_L^c) \cdot x + E_{sum}^s + E_L^c \\
&\quad + u_{max}^s \cdot S_{sum}^s.
\end{aligned}$$

□

The demand placed by jobs in **D** after t_d is $B(\mathbf{D}, t_d, S) = \sum_{\tau_k \in D_H} (e_k - \delta_k)$. Thus, by (3.1) and Lemmas 3.1 and 3.3, we have the following upper bound:

$$\begin{aligned}
Z &\leq (U_{sum}^s + U_L^c) \cdot x + E_{sum}^s + E_L^c + u_{max}^s \cdot S_{sum}^s \\
&\quad + \sum_{\tau_k \in D_H} (\delta_i(1 - u_k) + (e_k - \delta_k)) \\
&\leq (U_{sum}^s + U_L^c) \cdot x + E_{sum}^s + E_L^c + u_{max}^s \cdot S_{sum}^s \\
&\quad + E_{sum}.
\end{aligned} \tag{3.7}$$

3.2.2 Lower Bound

Lemma 3.4, given below, establishes a lower bound on Z that is necessary for the tardiness of $\tau_{l,j}$ to exceed $x + e_l + s_l$.

Definition 3.8. If job $\tau_{i,v}$ is enabled and not suspended at time t but does not execute at t, then it is *preempted* at t.

Definition 3.9. If $\tau_{i,v}$'s first phase is an execution (suspension) phase and it begins executing (a suspension) for the first time at t, then t is called its *start time*, denoted $S(\tau_{i,v})$. If $\tau_{i,v}$'s last phase (be it execution or suspension) completes at time t', then t' is called its *finish time*, denoted $F(\tau_{i,v})$.

Definition 3.10. Let $S_{max}^{H} = \max\{S_1^H, S_2^H, ..., S_n^H\}$. ($S_i^H$ was defined earlier in Section 3.1. We repeat its definition for ease of readability.)

Definition 3.11. Let $\xi_i^H = \dfrac{S_{max}^H}{S_{max}^H + H \cdot e_i}$ be the *suspension ratio* of τ_i. Let $\xi_{max}^H = \max\{\xi_1, \xi_2, ..., \xi_n\}$ be the *maximum suspension ratio*.

Lemma 3.4. *If the tardiness of* $\tau_{l,j}$ *exceeds* $x + e_l + s_l$*, then* $Z > (1 - \xi_{max}^H) \cdot mx - (m-1)e_l - m \cdot s_l - n \cdot (S_{max}^H + 2S_{max}^1)$.

Proof. We prove the contrapositive: we assume that

$$\begin{aligned} Z \;\leq\; & (1 - \xi_{max}^H) \cdot mx - (m-1)e_l \\ & -m \cdot s_l - n \cdot (S_{max}^H + 2S_{max}^1) \end{aligned} \tag{3.8}$$

holds and show that the tardiness of $\tau_{l,j}$ cannot exceed $x + e_l + s_l$. Let η_l be the amount of work $\tau_{l,j}$ performs by time t_d in S. Define y as follows.

$$y = (1 - \xi_{max}^H) \cdot x + \frac{\eta_l}{m} \tag{3.9}$$

Let W be the amount of work due to jobs in **d** $\cup$ **D** that can compete with $\tau_{l,j}$ at or after $t_d + y$, including the work due for $\tau_{l,j}$. Let $t_f = F(\tau_{l,j})$. We consider two cases.

Case 1. $[t_d, t_d + y)$ *is a busy interval for* **d** $\cup$ **D**. In this case, by (3.8) and (3.9), we have

$$\begin{aligned} & W \\ & = Z - my \leq (1 - \xi_{max}^H) \cdot mx - (m-1)e_l - m \cdot s_l - n \cdot (S_{max}^H + 2S_{max}^1) - my \\ & = (1 - \xi_{max}^H) \cdot mx - (m-1)e_l - m \cdot s_l - n \cdot (S_{max}^H + 2S_{max}^1) - (1 - \xi_{max}^H) \cdot mx - \eta_l \\ & < 0. \end{aligned}$$

Since $\tau_{l,j}$ can suspend for at most s_l time units after $t_d + y$ (and at least one task executes while it is not suspended), the amount of work performed by the system for jobs in **d** $\cup$ **D** during the interval $[t_d + y, t_f)$ is at least $t_f - t_d - y - s_l$. Hence, $t_f - t_d - y - s_l \leq W < 0$. Therefore, the tardiness of $\tau_{l,j}$ is $t_f - t_d < y + s_l = (1 - \xi^H_{max}) \cdot x + \frac{\eta_l}{m} + s_l \leq x + e_l + s_l$.

Case 2. $[t_d, t_d + y)$ *is a non-busy interval for* **d** $\cup$ **D**. Let $t_s \geq t_d$ be the earliest non-busy instant in $[t_d, t_d + y)$. Job $\tau_{l,j}$ cannot become enabled until its predecessor (if it exists) completes. Let t_p be the finish time of $\tau_{l,j}$'s predecessor (i.e., $\tau_{i,j-1}$), if it exists; otherwise ($j = 1$), let $t_p = 0$. We consider three subcases.

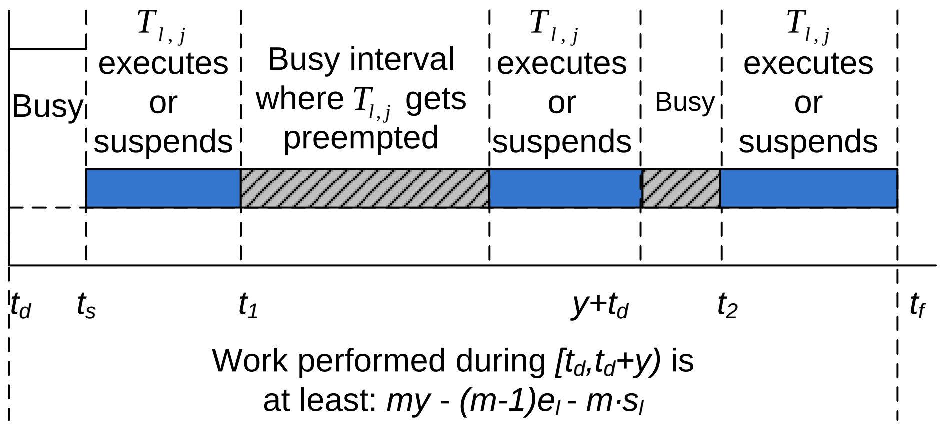
Subcase 2.1. $t_p \leq t_s$ *and* $\tau_{l,j}$ *is not preempted after* t_s. In this case, $\tau_{l,j}$ performs its remaining execution and suspension phases in sequence without preemption after t_s (note that, by Definition 7.5, $\tau_{l,j}$ is not considered to be preempted when it is suspended). Thus, because $t_s < t_d + y$, by (3.9), the tardiness of $\tau_{l,j}$ is at most $t_s + e_l - \eta_l + s_l - t_d < t_d + y + e_l - \eta_l + s_l - t_d = (1 - \xi^H_{max}) \cdot x + \frac{\eta_l}{m} + e_l - \eta_l + s_l \leq x + e_l + s_l$.

The claim below will be used in the next two subcases.

Claim 3. The amount of work due to **d** $\cup$ **D** performed within $[t_1, t_2)$, where $S(\tau_{l,j}) \leq t_1 < t_2 \leq F(\tau_{l,j})$, is at least $m(t_2 - t_1) - (m-1)e_l - m \cdot s_l$.

Proof. Within $[t_1, t_2)$, all intervals during which $\tau_{l,j}$ is preempted are busy, and $\tau_{l,j}$ can execute for at most e_l time. Within intervals where $\tau_{l,j}$ executes, at least one processor is occupied by $\tau_{l,j}$. Thus, at most $m - 1$ processors are idle while $\tau_{l,j}$ executes (for at most e_l time units) in $[t_1, t_2)$. Also, all processors can be idle while $\tau_{l,j}$ is suspended and this happens for at most s_l time units in $[t_1, t_2)$. □

Subcase 2.2. $t_p \leq t_s$ *and* $\tau_{l,j}$ *is preempted after* t_s. Let t_1 be the earliest time when $\tau_{l,j}$ is preempted after t_s, and let t_2 be the last time $\tau_{l,j}$ resumes execution after being preempted. (A finite number of jobs have higher priority than $\tau_{l,j}$, so t_2 exists.) Then, as shown in Figure 7.5, $\tau_{l,j}$ executes or suspends within $[t_s, t_1)$. Also, because $\tau_{l,j}$ is preempted at t_1, t_1 is busy with respect to **d** $\cup$ **D**. Within $[t_1, t_2)$, $\tau_{l,j}$ could be repeatedly preempted. All such intervals during which $\tau_{l,j}$ is preempted must be busy in order for the preemption to happen. Note that $t_f \leq t_2 + e_l - \eta_l + s_l$. Thus, if $t_2 \leq y + t_d$, then $t_f \leq y + t_d + e_l - \eta_l + s_l$, which by (3.9) implies $\tau_{l,j}$'s tardiness is $t_f - t_d \leq y + e_l - \eta_l + s_l \leq (1 - \xi^H_{max}) \cdot x + e_l + s_l \leq x + e_l + s_l$, as required. If



Busy
$T_{l,j}$ executes or suspends
Busy interval where $T_{l,j}$ gets preempted
$T_{l,j}$ executes or suspends
Busy
$T_{l,j}$ executes or suspends
t_d
t_s
t_1
$y+t_d$
t_2
t_f
Work performed during $[t_d,t_d+y)$ is at least: $my - (m-1)e_l - m\cdot s_l$

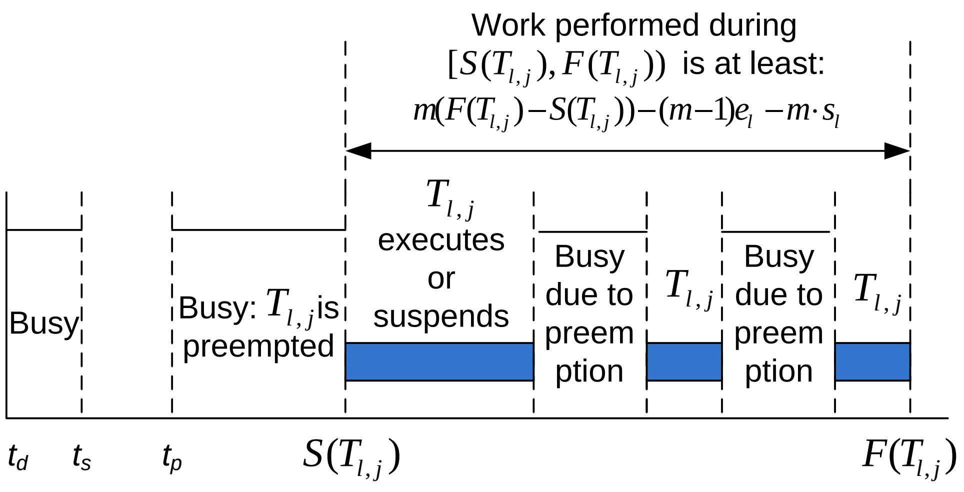
Figure 3.1: Subcase 2.2.

$t_2 > t_d + y$, then by Claim 3, the amount of work due to $\mathbf{d} \cup \mathbf{D}$ performed within $[t_s, t_d + y)$ is at least $m(t_d + y - t_s) - (m-1)e_l - m \cdot s_l$. Because $[t_d, t_s)$ is busy, the work due to $\mathbf{d} \cup \mathbf{D}$ performed within $[t_d, t_d + y)$ is thus at least $my - (m-1)e_l - m \cdot s_l$. Hence, the amount of work that can compete with $\tau_{l,j}$ (including work due to $\tau_{l,j}$) at or after $t_d + y$ is

$$\begin{aligned}
W &\leq Z - (my - (m-1)e_l - m \cdot s_l) \\
&\quad \{\text{by (3.8)}\} \\
&\leq (1 - \xi^H_{max}) \cdot mx - (m-1)e_l - m \cdot s_l - \\
&\quad n{\cdot}(S^H_{max}{+}2S^1_{max}) - (my - (m-1)e_l - m \cdot s_l) \\
&= (1 - \xi^H_{max}) \cdot mx - n{\cdot}(S^H_{max}{+}2S^1_{max}) - my \\
&\quad \{\text{by (3.9)}\} \\
&= -n{\cdot}(S^H_{max}{+}2S^1_{max}) - \eta_l \\
&\leq 0.
\end{aligned}$$

Therefore, the tardiness of $\tau_{l,j}$ is $t_f - t_d \leq y + W \leq y = (1 - \xi^H_{max}) \cdot x + \dfrac{\eta_l}{m} < x + e_l + s_l$.

Subcase 2.3: $t_p > t_s$. The earliest time $\tau_{l,j}$ can commence its first phase (be it an execution or suspension phase) is t_p, as shown in Figure 3.2. If fewer than m tasks have enabled jobs in $\mathbf{d} \cup \mathbf{D}$ at any time instant within $[t_s, t_p)$, then $\tau_{l,j}$ will begin its first phase at t_p and finish by time



Work performed during
$[S(T_{l,j}), F(T_{l,j}))$ is at least:
$m(F(T_{l,j})-S(T_{l,j}))-(m-1)e_l - m\cdot s_l$
Busy
Busy: $T_{l,j}$ is preempted
$T_{l,j}$ executes or suspends
Busy due to preemption
$T_{l,j}$
Busy due to preemption
$T_{l,j}$
t_d
t_s
t_p
$S(T_{l,j})$
$F(T_{l,j})$

Figure 3.2: Subcase 2.3.

$t_p + e_l + s_l$. (Note that the number of enabled jobs in $\mathbf{d} \cup \mathbf{D}$ does not increase after t_d.) By Property (P) (applied to $\tau_{l,j}$'s predecessor), $t_p \le t_d - p_l + x + e_l + s_l \le t_d + x$. Thus, the tardiness of $\tau_{l,j}$ is $t_f - t_d \le t_p + e_l + s_l - t_d \le x + e_l + s_l$.

The remaining possibility (which requires a much lengthier argument) is: $t_p > t_s$ and at least m tasks have enabled jobs in $\mathbf{d} \cup \mathbf{D}$ at each time instant within $[t_s, t_p)$. In this case, given that at least m tasks have enabled jobs in $\mathbf{d} \cup \mathbf{D}$ at t_s, t_s is non-busy due to suspensions.

Let W' be the amount of work due to $\mathbf{d} \cup \mathbf{D}$ performed during $[t_s, t_p)$. Let I be the total idle time in $[t_s, t_p)$, where the idle time at each instant is the number of idle processors at that instant. Then, $W' + I = m \cdot (t_p - t_s)$. The following claim will be used to complete the proof of Subcase 2.3.

Claim 4. $W' \ge (1 - \xi^H_{max}) \cdot m(t_p - t_s) - n \cdot (S^H_{max} + 2S^1_{max})$.

Proof. We begin by dividing the interval $[t_s, t_p)$ into subintervals on a per-processor basis. The subintervals on processor k are denoted $[IT^{(k)}_i, ET^{(k)}_i)$, where $1 \le i \le q_k$, $IT^{(k)}_i = t_s$, $IT^{(k)}_{i+1} = ET^{(k)}_i$, and $ET^{(k)}_{q_k} = t_p$, as illustrated in Fig 3.3. With each such subinterval $[IT^{(k)}_i, ET^{(k)}_i)$, we associate a unique task, denoted $\tau^{(k)}_i$. We assume that during $[IT^{(k)}_i, ET^{(k)}_i)$, $\tau^{(k)}_i$ executes only on processor k, and $ET^{(k)-}_i$ is the last time $\tau^{(k)}_i$ is enabled within $[t_s, t_p)$. Thus, if $ET^{(k)}_i < t_p$, then the last job of $\tau^{(k)}_i$ to be enabled within $[t_s, t_p)$ finishes its last phase (be it execution or suspension) at time $ET^{(k)-}_i$; if $ET^{(k)}_i = t_p$, then $\tau^{(k)}_i$ has enabled jobs throughout $[IT^{(k)}_i, t_p)$. Note that it is possible that $\tau^{(k)}_i$ executes or suspends within $[t_s, t_p)$ prior to $IT^{(k)}_i$. We call the subinterval

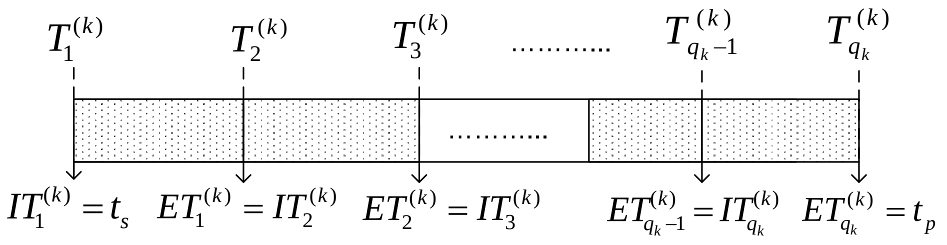


Figure 3.3: Presence intervals within $[t_s, t_p]$.

$[IT_i^{(k)}, ET_i^{(k)})$ the *presence interval* of $\tau_i^{(k)}$. The fact that a unique task can be associated with each subinterval follows from the assumption that at least m tasks have jobs in **d** $\cup$ **D** that are enabled at each time instant in $[t_s, t_p)$.[3] (Note that multiple jobs of $\tau_i^{(k)}$ may execute during its presence interval.) We let $\lambda^{(k)}$ denote the set of all tasks that have presence intervals on processor k.

We now upper-bound the idleness on processor k by bounding its idleness within one of its presence intervals. For conciseness, we denote this interval and its corresponding task as $[IT(T), ET(T))$ and T, respectively. If processor k is idle at any time in $[IT(T), ET(T))$, then some job of T is suspended at that time. Thus, the total suspension time of jobs of T in $[IT(T), ET(T))$, denoted $I(T)$, upper-bounds the idle time on processor k in $[IT(T), ET(T))$.

Task T may have multiple jobs that are enabled within its presence interval. Such a job is said to *fully execute* in the presence interval if it starts its first phase (be it execution or suspension) within the presence interval and also completes all of its execution phases in that interval (note that it may not complete all of its suspension phases). A job is said to *partially execute* in the presence interval if it starts its first phase (be it execution or suspension) before the presence interval or completes its last execution phase after the presence interval. Note that at most two jobs of T could partially execute in its presence interval (namely, the first and last jobs to be enabled in that interval). We now prove that $I(T)$, and hence the idleness within $[IT(T), ET(T))$ on processor k, is at most $\xi_{max}^H \cdot (ET(T) - IT(T)) + S_{max}^H + 2S_{max}^1$ (see Definition 3.10). Depending on the number of jobs of T that execute during T's presence interval, we have two cases.

[3]As time increases from t_s to t_p, whenever a presence interval ends on processor k, a task exists that can be used to define the next presence interval on processor k. It can also be assumed without loss of generality that this task executes only on processor k during its presence interval.

Case 1. *T has at most H jobs that fully execute in its presence interval.* (Additionally, T may have at most two jobs that partially execute in its presence interval.) In this case, $I(T)$ is clearly at most $S^H_{max} + 2S^1_{max}$.

Case 2. *T has more than H jobs that fully execute in its presence interval.* (Again, T may have at most two jobs that partially execute in its presence interval.) In this case, the jobs of T that are enabled in its presence interval can be divided into n_T sets, where one set contains fewer than H fully executed jobs plus at most two partially-executed jobs and each of the remaining $n_T - 1$ sets contains exactly H fully-executed jobs. Let θ denote the union of the latter $n_T - 1$ job sets. Without loss of generality, we assume that θ contains jobs that are enabled consecutively. The total suspension time for the first job set defined above is clearly at most $S^H_{max} + 2S^1_{max}$. We complete this case by showing that the total suspension time for all jobs in θ is at most $\xi^H_{max} \cdot (ET(T) - IT(T))$.

To ease the analysis, let $IT'(T)$ be the start time of the first enabled job in θ, and $ET'(T) = min(ET(T), FT)$, where FT is the finish time of the last enabled job in θ. Then $ET'(T) - IT'(T) \leq ET(T) - IT(T)$. Also, $ET'(T) - IT'(T) = I(\theta) + \Delta(\theta) + E(\theta)$, where $I(\theta)$ is the total suspension time of all jobs in θ within $[IT'(T), ET'(T))$, $\Delta(\theta)$ is the total preemption time of all jobs in θ within $[IT'(T), ET'(T))$, and $E(\theta)$ is the total execution time of all jobs in θ within $[IT'(T), ET'(T))$. (Recall that, by Definition 7.5, a suspended task is not considered to be preempted.) Given that θ contains $(n_T - 1)$ H fully-executed jobs, $I(\theta) \leq (n_T - 1) \cdot S^H_T$. Moreover, given our assumption that each job executes for the corresponding task's WCET, $E(\theta) = (n_T - 1) \cdot H \cdot e(T)$, where $e(T)$ is the WCET of T. Thus,

$$\begin{aligned}
I(\theta) &= \frac{I(\theta)}{ET'(T) - IT'(T)} \cdot (ET'(T) - IT'(T)) \\
&= \frac{I(\theta)}{I(\theta) + \Delta(\theta) + E(\theta)} \cdot (ET'(T) - IT'(T)) \\
&\leq \frac{I(\theta)}{I(\theta) + E(\theta)} \cdot (ET'(T) - IT'(T)) \\
&\quad \{ \text{ because } I(\theta) \leq (n_T - 1) \cdot S^H_T \} \\
&\leq \frac{(n_T - 1) \cdot S^H_T}{(n_T - 1) \cdot S^H_T + E(\theta)} \cdot (ET'(T) - IT'(T)) \\
&\quad \{ \text{ because } E(\theta) = (n_T - 1) \cdot H \cdot e(T) \}
\end{aligned}$$

$$
\begin{aligned}
&\leq \frac{(n_T-1)\cdot S_T^H\cdot(ET'(T)-IT'(T))}{(n_T-1)\cdot S_T^H+(n_T-1)\cdot H\cdot e(T)}\\
&= \frac{S_T^H}{S_T^H+H\cdot e(T)}\cdot(ET'(T)-IT'(T))\\
&\quad \{\text{by Definition 3.10}\}\\
&\leq \frac{S_{max}^H}{S_{max}^H+H\cdot e(T)}\cdot(ET'(T)-IT'(T))\\
&\quad \{\text{by Definition 3.11}\}\\
&= \xi_{max}^H\cdot(ET'(T)-IT'(T))\\
&\leq \xi_{max}^H\cdot(ET(T)-IT(T)).
\end{aligned}
$$

This concludes the proof of Case 2 of Claim 4.

Given that a task can be identified with only one presence interval and there are at most n tasks, the idleness within $[t_s, t_p)$ on m processor satisfies

$$
\begin{aligned}
I &\leq n\cdot(S_{max}^H+2S_{max}^1)\\
&\quad + \sum_{T\in\lambda^{(1)}\cup\ldots\cup\lambda^{(k)}} \xi_{max}^H\cdot(ET(T)-IT(T))\\
&= \xi_{max}^H\cdot m(t_p-t_s)+n\cdot(S_{max}^H+2S_{max}^1).
\end{aligned}
$$

Thus, $W' = m(t_p - t_s) - I \geq (1-\xi_{max}^H)\cdot m(t_p-t_s) - n\cdot(S_{max}^H+2S_{max}^1)$. This completes the proof of Claim 4. □

We now complete the proof of Subcase 3.2.2 (and thereby Lemma 3.4). As shown in Figure 3.2, $[t_d, t_s)$ and $[t_p, S(\tau_{l,j}))$ are busy for $\mathbf{d}\cup\mathbf{D}$. By Claim 3, the amount of work due to $\mathbf{d}\cup\mathbf{D}$ performed in $[S(\tau_{l,j}), F(\tau_{l,j}))$ is at least $m(F(\tau_{l,j}) - S(\tau_{l,j})) - (m-1)e_l - m\cdot s_l$. By Claim 4, the amount of work due to $\mathbf{d}\cup\mathbf{D}$ performed in $[t_s, t_p)$ is at least $(1-\xi_{max}^H)\cdot m(t_p-t_s) - n\cdot(S_{max}^H - 2S_{max}^1)$. By summing over all of these subintervals, we can lower-bound the amount of work due to $\mathbf{d}\cup\mathbf{D}$ performed in $[t_d, F(\tau_{l,j}))$, i.e., Z:

$$
\begin{aligned}
Z &\geq m(t_s-t_d)+(1-\xi_{max}^H)\cdot m(t_p-t_s)\\
&\quad - n\cdot(S_{max}^H+2S_{max}^1)+m(S(\tau_{l,j})-t_p)\\
&\quad + m(F(\tau_{l,j})-S(\tau_{l,j}))-(m-1)e_l-m\cdot s_l. \qquad (3.10)
\end{aligned}
$$

By (3.8) and (3.10), we therefore have

$$\begin{aligned}
& (1-\xi^H_{max})\cdot mx-(m-1)e_l-m\cdot s_l \\
& -n\cdot(S^H_{max}+2S^1_{max}) \\
\geq\ & m(t_s-t_d)+(1-\xi^H_{max})\cdot m(t_p-t_s) \\
& -n\cdot(S^H_{max}+2S^1_{max})+m(S(\tau_{l,j})-t_p) \\
& +m(F(\tau_{l,j})-S(\tau_{l,j}))-(m-1)e_l-m\cdot s_l,
\end{aligned}$$

which gives,

$$F(\tau_{l,j})-t_d \leq (1-\xi^H_{max})\cdot x+\xi^H_{max}\cdot(t_p-t_s).$$

According to Property (P) (applied to $\tau_{l,j}$'s predecessor), $t_p-t_s\leq t_p-t_d\leq x-p_l+e_l+s_l\leq x$. Therefore, $F(\tau_{l,j})-t_d\leq(1-\xi^H_{max})\cdot x+\xi^H_{max}\cdot x<x+e_l+s_l$. □

3.2.3 Determining x

Setting the upper bound on $LAG(\mathbf{d},t_d,S)+B(\mathbf{D},t_d,S)$ in (3.7) to be at most the lower bound in Lemma 3.4 will ensure that the tardiness of $\tau_{l,j}$ is at most $x+e_l+s_l$. The resulting inequality can be used to determine a value for x. By (3.7) and Lemma 3.4, this inequality is $(U^s_{sum}+U^c_L)\cdot x+E^s_{sum}+E^c_L+u^s_{max}\cdot S^s_{sum}+E_{sum}\leq(1-\xi^H_{max})\cdot mx-(m-1)e_l-m\cdot s_l-n\cdot(S^H_{max}+2S^1_{max})$.

Let $V=E^s_{sum}+E^c_L+u^s_{max}\cdot S^s_{sum}+E_{sum}+(m-1)e_l+m\cdot s_l+n\cdot(S^H_{max}+2S^1_{max})$. Solving for x, we have

$$x\geq\frac{V}{(1-\xi^H_{max})\cdot m-U^s_{sum}-U^c_L}. \tag{3.11}$$

x is well-defined provided $U^s_{sum}+U^c_L<(1-\xi^H_{max})\cdot m$. If this condition holds and x equals the right-hand side of (3.11), then the tardiness of $\tau_{l,j}$ will not exceed $x+e_l+s_l$. A value for x that is independent of the parameters of τ_l can be obtained by replacing $(m-1)e_l+m\cdot s_l$ with $max_l((m-1)e_l+m\cdot s_l)$ in V.

Theorem 3.1. *With x as defined in (3.11), the tardiness of any task τ_l scheduled under GSA is at most $x + e_l + s_l$, provided $U^s_{sum} + U^c_L < (1 - \xi^H_{max}) \cdot m$.*

For GFIFO and GEDF, the bound in Theorem 3.1 can be improved.

Corollary 1. For GFIFO, Theorem 3.1 holds with V replaced by $V - E_{sum} + \sum_{p_i > p_l} e_i$ in the numerator of (3.11).

Proof. Under GFIFO, D_H consists of carry-in jobs that are released before $r_{l,j}$ and have deadlines later than t_d, which implies that these jobs have periods greater than p_l. Thus, the upper bound in (3.7) can be refined to obtain $LAG\,(\mathbf{d}, t_d, S) + B(\mathbf{D}, t_d, S) \leq (U^s_{sum} + U^c_L) \cdot x + E^s_{sum} + E^c_L + u^s_{max} \cdot S^s_{sum} + \sum_{p_i > p_l} e_i$. Using this upper bound to solve for x, the corollary follows. $\square$

Corollary 2. For GEDF, Theorem 3.1 holds with V replaced by $V - E_{sum}$ in the numerator of (3.11).

Proof. Under GEDF, the demand placed by jobs in **D** after t_d is zero because $\mathbf{D} = \emptyset$. Thus, under GEDF, $LAG\,(\mathbf{d}, t_d, S) + B(\mathbf{D}, t_d, S) \leq (U^s_{sum} + U^c_L) \cdot x + E^s_{sum} + E^c_L + u^s_{max} \cdot S^s_{sum}$. Using this upper bound to solve for x, the corollary follows. $\square$

3.2.4 A Counterexample

Previous research has shown that every sporadic task system for which $U_{sum} \leq m$ without self-suspensions has bounded tardiness under GEDF and GFIFO [41, 78]. We now show that it is possible for a task system containing self-suspending tasks to have unbounded tardiness under GEDF or GFIFO if the utilization constraint in Theorem 1 is violated.

Consider a two-processor task set τ that consists of three self-suspending tasks: τ_1 = (e3, s7, 10), τ_2 = (e1, s8, e1, 10), and τ_3 = (e1, s8, e1, 10). For this system, $\xi^H_{max} = 0.8$ (assuming $H = 1$) and $U^s_{sum} + U^c_L = 0.7$. Thus, $(1 - \xi^H_{max}) \cdot m = 0.4 < U^s_{sum} + U^c_L$, which violates the condition stated in Theorem 1. Figure 3.4 shows the tardiness of each task in this system under GFIFO/GEDF by job index assuming each job is released as early as possible. We have verified analytically that the tardiness growth rate seen in Figure 3.4 continues indefinitely.

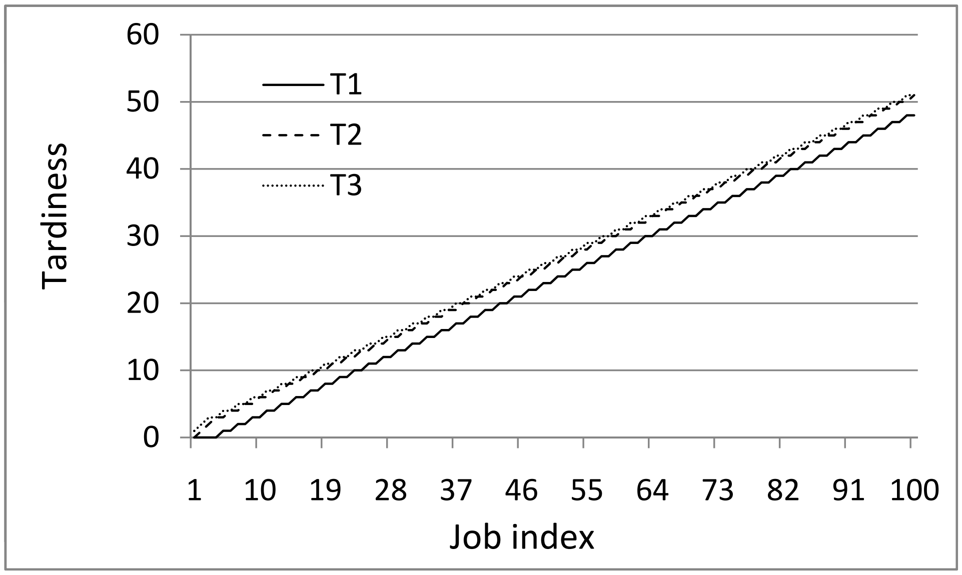


Figure 3.4: Tardiness growth rates in counterexample.

3.2.5 Experimental Evaluation

In this section, we describe experiments conducted using randomly-generated task sets to evaluate the applicability of the tardiness bound in Theorem 3.1. Our goal is to examine how restrictive the theorem's utilization cap is, and to compare it with the suspension-oblivious approach (recall Section 1.7.1.1 in Chapter 1), wherein all suspension phases are treated as computation phases. From [41, 78], tardiness is bounded under the suspension-oblivious approach provided $U_{sum} \leq m$ and $U_L \leq m$, where U_L is the sum of the $\min(m-1, n)$ largest task utilizations. Note that under the suspension-oblivious approach, suspensions also contribute to tasks' utilizations. That is, after treating all suspensions as computation, $u_i = \dfrac{e_i + s_i}{p_i}$ holds.

In our experiments, task sets were generated as follows. Task periods were uniformly distributed over [50*ms*,100*ms*]. Task utilizations were uniformly distributed over [0.001,0.5]. Task execution costs were calculated from periods and utilizations. We varied U^s_{sum} as follows: $U^s_{sum} = 0.1 \cdot U_{sum}$ (suspensions are relatively infrequent), $U^s_{sum} = 0.4 \cdot U_{sum}$ (suspensions are moderately frequent), and $U^s_{sum} = 0.7 \cdot U_{sum}$ (suspensions are frequent). Moreover, we varied ξ_{max} as follows: 0.05 (suspensions are short), 0.2 (suspensions are moderate), and 0.5 (suspensions are long). Table 3.2 shows suspension-length ranges generated by these parameters. We also varied U_{sum} within $\{1,$

per-task utilization \ suspension length		short suspensions ξ_{max} = 0.05	moderate suspensions ξ_{max} = 0.2	long suspensions ξ_{max} = 0.5
light	min:	2.6 *μs*	12 *μs*	50 *μs*
	avg:	197 *μs*	938 *μs*	3.75 *ms*
	max:	526 *μs*	2.5 *ms*	10 *ms*
medium	min:	263 *μs*	1.25 *ms*	5 *ms*
	avg:	789 *μs*	3.75 *ms*	15 *ms*
	max:	1.6 *ms*	7.5 *ms*	30 *ms*
heavy	min:	789 *μs*	3.75 *ms*	15 *ms*
	avg:	2.2 *ms*	10.3 *ms*	41.25 *ms*
	max:	4.2 *ms*	20 *ms*	80 *ms*

Table 3.1: Per-job suspension-length ranges.

2, ..., 8}. For each combination of (ξ_{max}, U^s_{sum}, U_{sum}), 1,000 task sets were generated for an eight-processor system. For each generated system, soft real-time schedulability (i.e., the ability to ensure bounded tardiness) was checked under the suspension-oblivious approach and using the condition stated in Theorem 3.1.

The schedulability results that were obtained are shown in Figures 3.5-3.13. In these figures, each curve plots the fraction of the generated task sets the corresponding approach successfully scheduled, as a function of total utilization. Each graph gives three curves per tested approach for the cases of short, moderate, and long suspensions, respectively. The label "**LA-s(m/l)**" indicates our proposed approach assuming short (moderate/long) suspensions. Similar "**SC**" (suspensions as computation) labels are used for the suspension-oblivious approach.

In most tested scenarios summarized in Figures 3.5-3.10, our approach proved to be superior, sometimes by a substantial margin. However, in many of the scenarios summarized in Figures 3.11-3.13, the suspension-oblivious approach proved to be superior. In these scenarios, task utilizations are high and suspensions are long or frequent. Our analysis is negatively impacted in such cases because U^c_L tends to be large when utilizations are high, and ξ_{max} tends to be large when suspensions are long. It is worth noting, however, that our approach allows certain tasks to be designated as computational tasks. Thus, the suspension-oblivious approach is really a special case of our approach. Motivated by this, we investigate in the next section intermediate choices between the two extremes of modeling all versus no suspensions as computation.

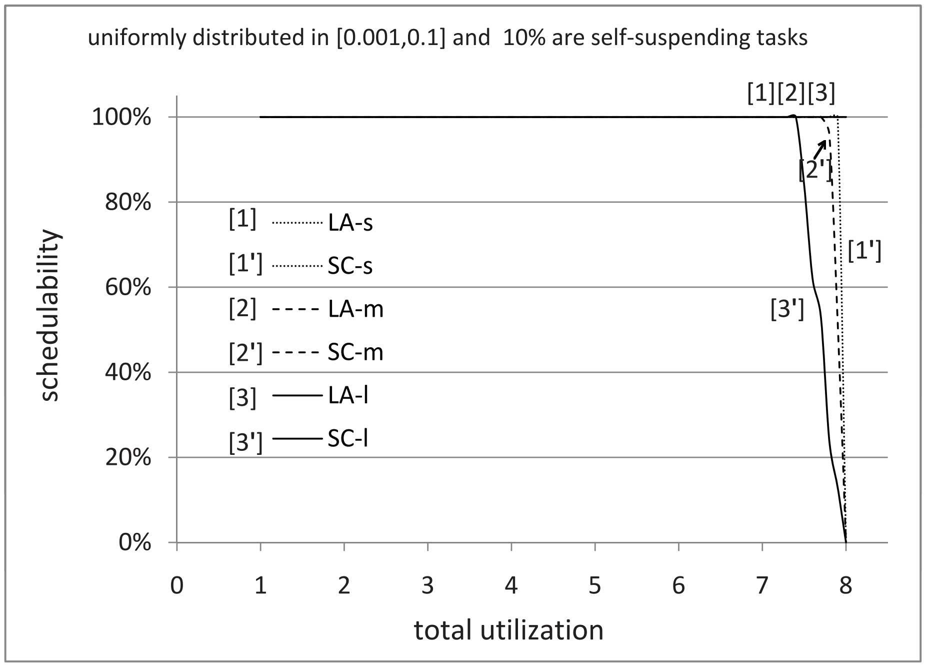


Figure 3.5: Light per-task utilizations, relatively infrequent suspensions

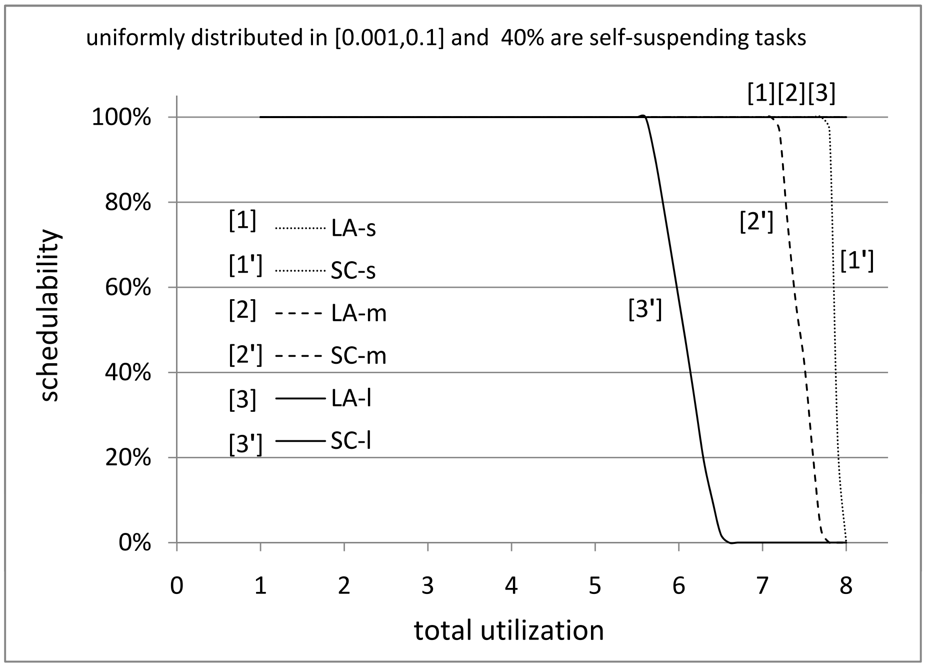


Figure 3.6: Light per-task utilizations, moderately frequent suspensions

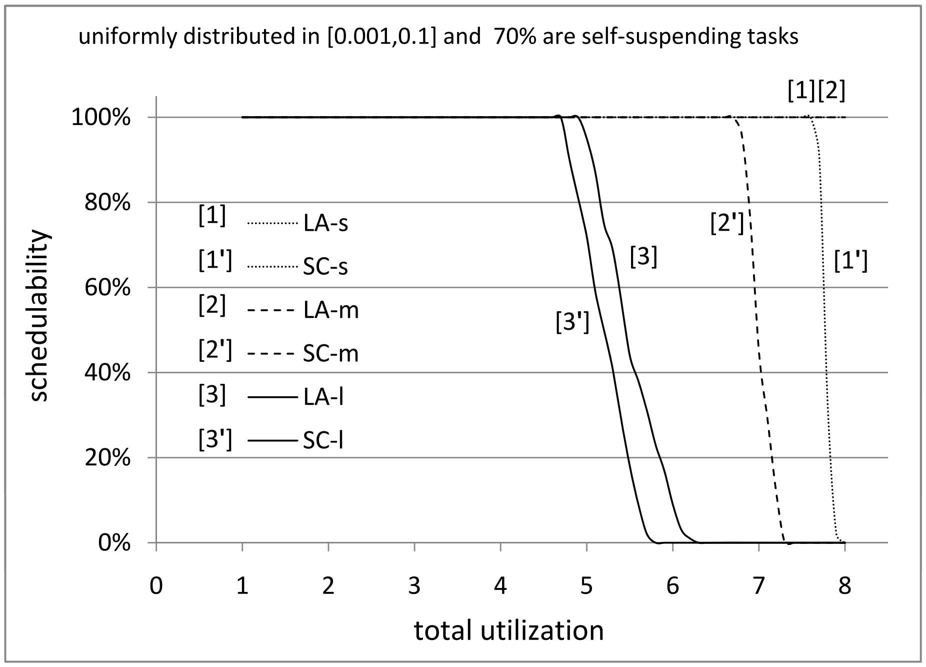


Figure 3.7: Light per-task utilizations, frequent suspensions

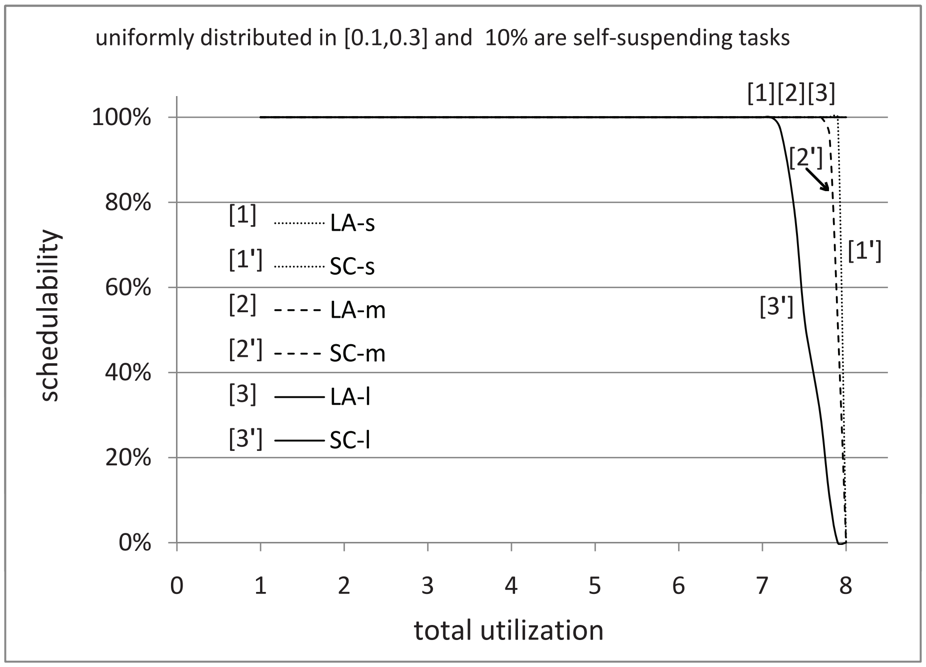


Figure 3.8: Medium per-task utilizations, relatively infrequent suspensions

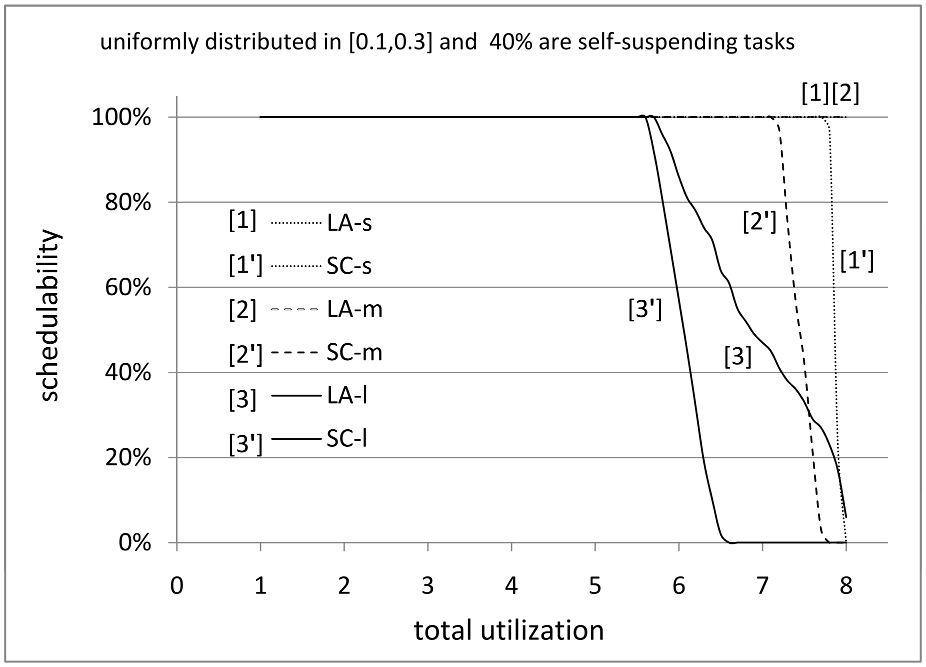


Figure 3.9: Medium per-task utilizations, moderately frequent suspensions

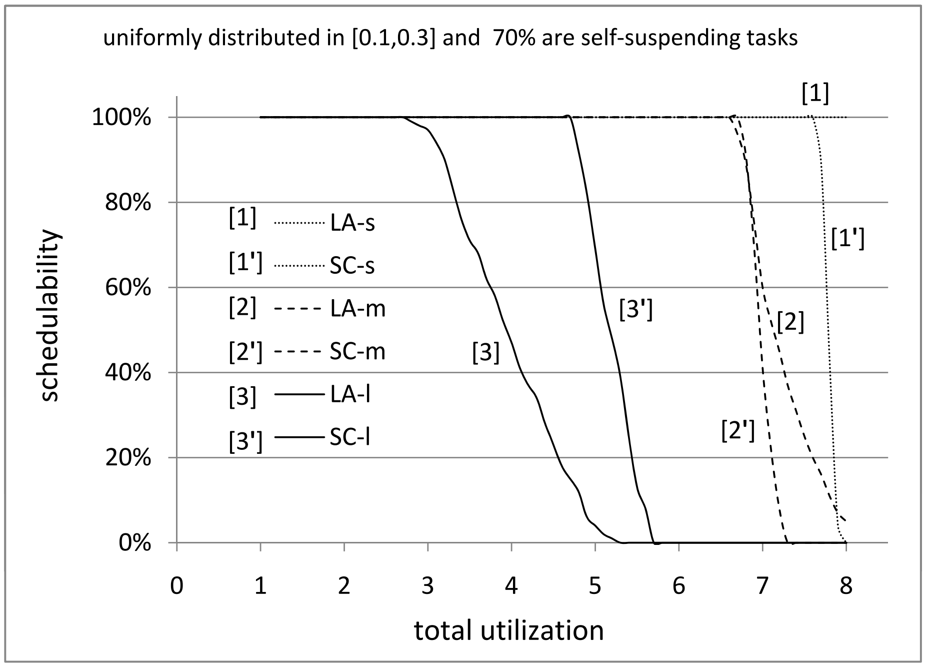


Figure 3.10: Medium per-task utilizations, frequent suspensions

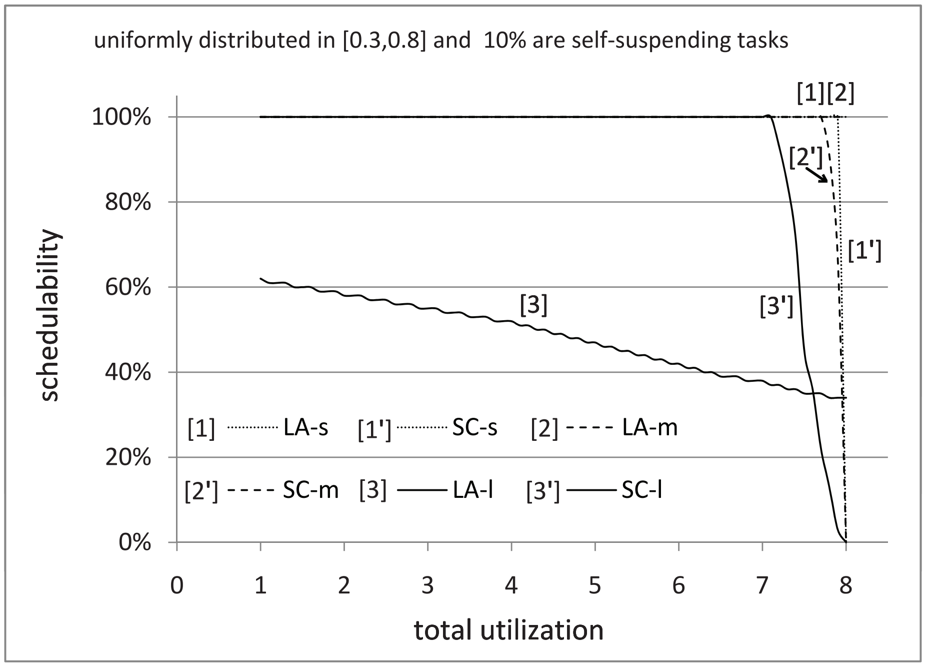


Figure 3.11: Heavy per-task utilizations, relatively infrequent suspensions

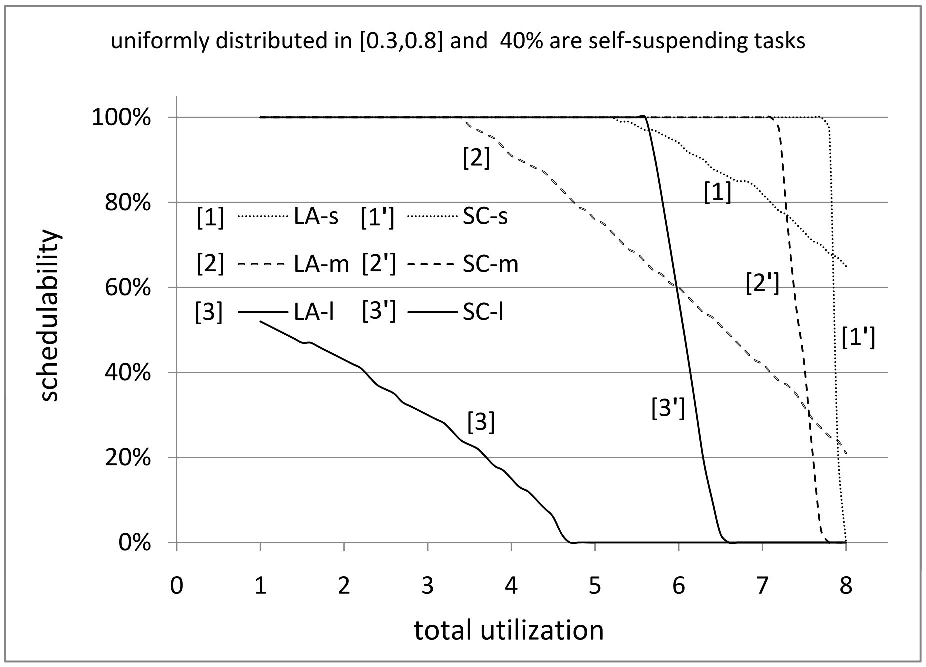


Figure 3.12: Heavy per-task utilizations, moderately frequent suspensions

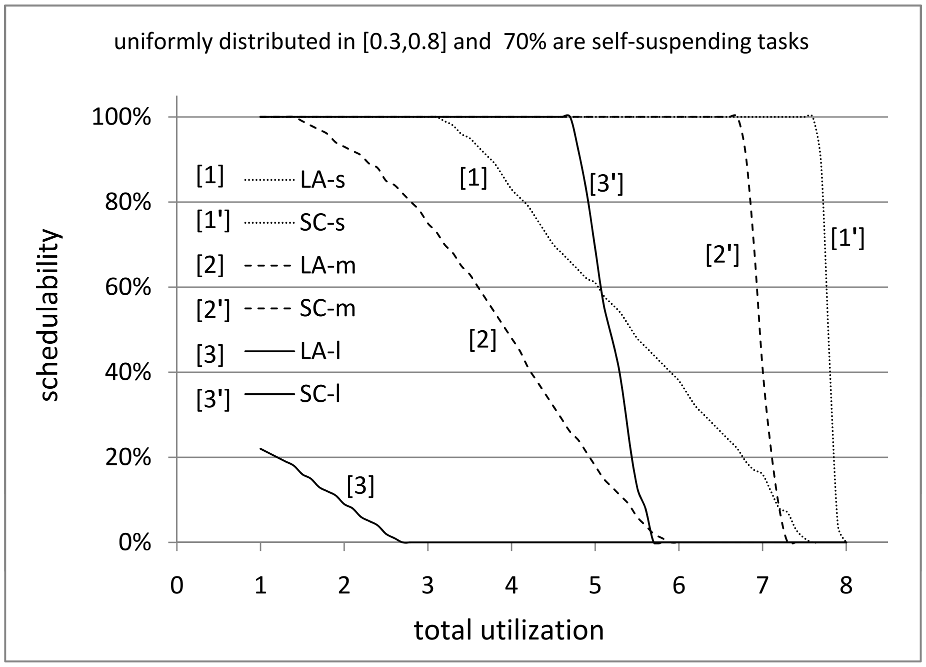


Figure 3.13: Heavy per-task utilizations, frequent suspensions

3.3 An Effective Technique to Improve Schedulability

In the previous section, we showed that under GEDF and GFIFO, bounded tardiness can be ensured even if tasks self-suspend. Specifically we showed that tardiness in such a system is bounded provided

$$U^s_{sum} + U^c_L < (1 - \xi_{max}) \cdot m, \tag{3.12}$$

where U^s_{sum} is the total utilization of all self-suspending tasks, c is the number of computational tasks (which do not self-suspend), m is the number of processors, U^c_L is the sum of the $\min(m-1, c)$ largest computational task utilizations, and ξ_{max} is the maximum suspension ratio defined in Definition 3.11 (where $H = 1$). Significant capacity loss may occur when using (3.12) if ξ_{max} is large. Unfortunately, it is unavoidable that many self-suspending task systems will have large ξ_{max} values. For example, consider a task system with three tasks scheduled on two processors: τ_1 has an execution cost of 5, a suspension length of 5, and a period of 10, τ_2 has an execution cost of 2, a suspension length of 0, and a period of 8, and τ_3 has an execution cost of 2, a suspension length of 2, and a period of 8. For this system, $U^s_{sum} = u_1 + u_3 = \frac{5}{10} + \frac{2}{8} = 0.75$, $U^c_L = u_2 = \frac{2}{8} = 0.25$, $\xi_{max} = \xi_1 = \frac{5}{5+5} = 0.5$. Although the total utilization of this task system is only half of the overall processor capacity, it is not schedulable using the prior analysis since it violates the utilization constraint in (3.12) (since $U^s_{sum} + U^c_L = 1 = (1 - \xi_{max}) \cdot m$). Notice that the main reason for the violation is a large value of ξ_{max}. In this section, we propose an approach, which we call PSAC (Partial Suspensions As Computation), that relaxes the utilization constraint in (3.12).

As shown in Section 3.2, the suspension-oblivious approach, which treats *all* suspensions as computation, is superior to the suspension-aware analysis when per-task utilizations are high and suspensions are long. By observing that the analysis in Section 3.2 actually treats no suspension as computation, our motivation is to investigate intermediate choices between the two extremes of treating all versus no suspensions as computation.

The goal of PSAC is to decrease ξ_{max} for any given task system τ. To motivate the details that follow, suppose that τ_i has the maximum suspension ratio and it is the only task with that suspension ratio, and after converting z_i time units of its suspension time to computation, it still has the maximum suspension ratio. Treating z_i time units of the suspension time of τ_i as computation, (3.12) becomes $U^s_{sum} + U^c_L + z_i/p_i < (1 - \frac{s_i}{s_i+e_i} + \frac{z_i}{s_i+e_i}) \cdot m$. Given that $e_i + s_i \leq p_i$ and $m \geq 2$, $z_i/p_i < \frac{z_i}{s_i+e_i} \cdot m$ holds. Therefore, as long as $U_{sum} + z_i/p_i \leq m$ holds, the utilization constraint in (3.12) becomes less stringent if z_i time units of the suspension time of τ_i are treated as computation.

Let c_i denote the length of the suspension time of task τ_i that is treated as computation by PSAC. For any task τ_i, after treating c_i time units of suspensions as computation, its suspension ratio becomes $\frac{s_i - c_i}{e_i + s_i}$, and its utilization becomes $u_i + \frac{c_i}{p_i}$. A set of values c_i $(1 \leq i \leq n)$ is *valid* if the following conditions (3.13)-(3.15) hold.

$$U^s_{sum} + U^c_L + \sum_{i=1}^{n} \frac{c_i}{p_i} < (1 - \xi_{max}) \cdot m \tag{3.13}$$

$$U_{sum} + \sum_{i=1}^{n} \frac{c_i}{p_i} \leq m \tag{3.14}$$

$$0 \leq c_i \leq s_i \tag{3.15}$$

If (3.13) and (3.14) hold, then, after treating c_i time units of the suspension time of each task τ_i as computation, τ satisfies (3.12) and its total utilization is at most m (note that, in (3.13), ξ_{max} is the maximum suspension ratio after treating suspensions as computation). Thus, (3.13) and (3.14) guarantee that τ becomes schedulable. Moreover, (3.15) trivially must hold.

Therefore, our goal is to find a set of valid values c_i $(1 \leq i \leq n)$. We first present in this section a solution using linear programming, and then an alternative algorithm that optimally finds valid c_i values (if they exist) and that has low time complexity. For scenarios where the number of variables and constraints in the linear program is large, applying the optimal algorithm is preferred since the linear programming approach may exhibit a long program running time.

3.3.1 Linear Programming Approach

We formalize the problem of finding a set of valid c_i values by specifying a linear program as follows. First, note that we want to minimize $\sum_{i=1}^{n} c_i$ because this results in a lower tardiness bound, according to Theorem 3.1. Given this, an appropriate linear program can be specified based on the constraints (3.13)-(3.15) given earlier. Let ϵ be a constant that can be arbitrarily small. The linear program we wish to solve is as follows

$$\text{Minimize} \quad \sum_{i=1}^{n} c_i$$

subject to:

$$U_{sum}^{s} + U_{L}^{c} + \sum_{i=1}^{n} \frac{c_i}{p_i} \leq (1 - \frac{s_i - c_i}{e_i + s_i}) \cdot m - \epsilon, \quad i = 1, ...n \tag{3.16}$$

$$U_{sum} + \sum_{i=1}^{n} \frac{c_i}{p_i} \leq m \tag{3.17}$$

$$c_i \leq s_i, \; i = 1, ...n \tag{3.18}$$

$$c_i \geq 0, \; i = 1, ...n. \tag{3.19}$$

In order to formalize the problem as a linear program, we replace the constraint in (3.13) by the set of constraints in (3.16). In particular, instead of a single constraint involving ξ_{max}, we have a constraint for each individual ξ_i. Also, these constraints are expressed so that equality is allowed, which is required in linear programs. This requires the introduction of a small ϵ term, as seen. This linear programming approach can find valid c_i values in polynomial time.

Henceforth, when considering algorithms for computing valid c_i values, we regard c_i as a variable and $\overline{c}_i$ as a value assigned to that variable. This is in keeping with how variables and values are denoted in work on linear programming.

Example. Consider a task system scheduled on four processors under GEDF that contains seven tasks: $\tau_1(e3, s6, e1, 10)$, $\tau_2(e4, s2, e2, 8)$, $\tau_3(e1, 3)$, $\tau_4(e8, s1, e6, 20)$, $\tau_5(e1, 6)$, $\tau_6(e2, 10)$, $\tau_7(e3, 15)$, $\tau_8(e1, 5)$, $\tau_9(e1, 10)$, and $\tau_{10}(e4, 20)$, where τ_1, τ_2, sand τ_4 are self-suspending tasks and all other tasks are computational tasks. This task system is deemed to be unschedulable by treating all suspensions as computation because this causes total utilization to exceed 2.0. It is also deemed to be unschedulable by using the analysis presented in Section 3.2 because it violates (3.12): $U^s_{sum} + U^c_L = 0.4 + 0.75 + 0.7 + \frac{1}{3} + \frac{1}{5} + \frac{1}{5} > (1 - \xi_{max}) \cdot m = (1 - 0.6) \cdot 4 = 1.6$.

However, this task system is determined to be schedulable by PSAC. By solving the above linear program, a set of valid c_i values can be obtained, where $\overline{c}_1 = \frac{59}{18}$ and $\overline{c}_i = 0$ (where $2 \leq i \leq 7$). The existence of valid c_i values implies that the system is schedulable.

3.3.2 An Optimal Polynomial-Time Algorithm

Although the linear programming approach presented in Section 3.3.1 enables valid c_i values (if they exist) to be obtained in polynomial time, algorithms for solving linear programs may exhibit long program running times for scenarios where the number of variables and constraints is large. As an alternative, we now present an optimal algorithm that generates minimal valid c_i values in $O((N^s)^2)$ time. A set of valid values $\{\overline{c}_1, \overline{c}_2, ..., \overline{c}_n\}$ is said to be *minimal* if $\sum_{i=1}^{n} \overline{c}_i$ is minimal over all valid sets of c_i values. An algorithm is *optimal* if it can find a minimal valid set of c_i values if a valid set of c_i values exists.

Let $\lambda_i = \{\tau_j \mid t_j \in \tau$ and ξ_j is ranked as the i^{th} largest suspension ratio$\}$. Thus, tasks in λ_1 have the same suspension ratio, which equals ξ_{max}. Obviously, we can decrease ξ_{max} by treating some suspension time of every task in λ_1 as computation, at the cost of increasing U^s_{sum} by an amount commensurate with the additional computation. Let $\{\lambda_1^1, \lambda_1^2 ... \lambda_1^{q_i}\}$ denote the tasks in λ_1. Let $c(\lambda_i^j)$ denote the amount of the suspension time of task λ_i^j that is treated as computation by PSAC, $e(\lambda_i^j)$ denote λ_i^j's execution cost, $s(\lambda_i^j)$ denote its suspension length, $p(\lambda_i^j)$ denote its period, $u(\lambda_i^j)$ denote its utilization, and $\xi(\lambda_i^j)$ denote its suspension ratio. Since we want to decrease ξ_{max}, we initially only care about tasks in λ_1.

We apply two constraints in order for the algorithm to run in $O((N^s)^2)$ time.

First, we desire to reduce the suspension ratio of every task in λ_1 by the same amount, as stated in (3.20). Reducing any task λ_1^i's suspension ratio by a greater amount than other tasks in λ_1 does not further decrease ξ_{max}, but only increases the utilization of λ_1^i.

$$\forall j, k :: \frac{s(\lambda_1^j) - c(\lambda_1^j)}{e(\lambda_1^j) + s(\lambda_1^j)} = \frac{s(\lambda_1^k) - c(\lambda_1^k)}{e(\lambda_1^k) + s(\lambda_1^k)} \tag{3.20}$$

Second, we desire to reduce the suspension ratio of each task in λ_1 to no less than $\xi(\lambda_2^1)$ (or 0 if λ_2^1 does not exist), as stated in (3.21). Further reducing some $\xi(\lambda_1^i)$ does not decrease ξ_{max}, but only increases the utilization of λ_1^i.

$$\forall j :: \frac{s(\lambda_1^j) - c(\lambda_1^j)}{e(\lambda_1^j) + s(\lambda_1^j)} \geq \xi(\lambda_2^1) \tag{3.21}$$

With the above preliminaries in place, the algorithm can be stated as a two-step procedure.

1. Find valid values $\{\overline{c}(\lambda_1^1), \overline{c}(\lambda_1^2), ..., \overline{c}(\lambda_1^{q_1})\}$ that also satisfy (3.20) and (3.21). If such values exist, then τ is schedulable. Otherwise, go to Step 2.

2. Decrease the suspension ratio of every task in λ_1 to $\xi(\lambda_2^1)$ by defining $\overline{c}(\lambda_1^j)$ for $\lambda_1^j \in \lambda_1$ so that $\frac{s(\lambda_1^j) - \overline{c}(\lambda_1^j)}{e(\lambda_1^j) + s(\lambda_1^j)} = \xi(\lambda_2)$ and (3.14) hold. If valid values do not exist, then τ is unschedulable. Otherwise, update the set λ and go to Step 1.

Time complexity. In the worst case, Steps 1 and 2 can be executed for N^s times. Step 1 has $O(N^s)$ time complexity if valid c_i values are obtained by the following procedure. First, solve (3.20) for each task λ_1^j, where $j \neq 1$, to specify $c(\lambda_1^j)$ as a function of $c(\lambda_1^1)$. This takes $O(N^s)$ time. Second, substitute all $c(\lambda_1^j)$ values as specified into (3.13)-(3.15) and (3.21), and solve for $c(\lambda_1^1)$. This also takes $O(N^s)$ time. Step 2 has $O(N^s)$ time complexity since it only needs to solve at most N^s equations. Therefore, the overall time complexity of the proposed algorithm is $O((N^s)^2)$.

example Consider the task system described in Section 3.3.1. For this system, λ_1 contains only τ_1, which has the maximum suspension ratio. We find valid values for $c(\lambda_1^1)$ that satisfy (3.13)-(3.15) and (3.20)-(3.21) (note that satisfying (3.20) is actually not required because τ_1 is the only task in λ_1). (3.13) requires that $U^s_{sum} + U^c_L + \sum_{j=1}^{q_1} \frac{c(\lambda_1^j)}{p(\lambda_1^j)} = 1.85 + \frac{11}{15} + \frac{c(\tau_1)}{10} < \left(1 - \frac{6 - c(\tau_1)}{10}\right) \cdot 4$, that is, $c(\tau_1) > \frac{59}{28}$ must hold. (3.14) requires $U_{sum} + \sum_{j=1}^{q_1} \frac{c(\lambda_1^j)}{p(\lambda_1^j)} = \frac{13}{4} + \frac{c(\tau_1)}{10} \leq m = 4$, that is, $c(\tau_1) \leq 7.5$ must hold. (3.15) requires that $c_1 \leq s_i = 6$. (3.21) requires $\frac{s_1 - c(\tau_1)}{e_1 + s_1} = \frac{6 - c(\tau_1)}{4 + 6} > \xi(\lambda_2^1) = \xi_2 = 0.25$, that is, $c(\tau_1) < 3.5$ must hold. Thus, any value satisfying $59/18 \leq c(\tau_1) \leq 3.5$ is a valid value for $c(\tau_1)$ and $\bar{c}(\tau_i) = 59/18$ is minimal. The existence of a valid value implies that the system is schedulable.

Optimality of the proposed algorithm. The optimality of the proposed algorithm is proved in the following theorem.

Theorem 3.2. *For any sporadic self-suspending task system τ, the proposed algorithm will find a minimal valid set of c_i values if a valid set of c_i values exists.*

Proof. Let $\{\bar{c}_1, \bar{c}_2, ..., \bar{c}_n\}$ be a minimal valid set of c_i values. We first prove by contradiction that, for any j, if $\bar{c}_j > 0$, then $\frac{s_j - \bar{c}_j}{e_j + s_j} = \xi_{max}$.

For any $\bar{c}_j > 0$, if $\frac{s_j - \bar{c}_j}{e_j + s_j} < \xi_{max}$ holds, then less than $\bar{c}_j$ time units of the suspension time of τ_j can be treated as computation to make ξ_j equal ξ_{max} or τ_j's original suspension ratio, whichever is smaller. Note that this does not increase ξ_{max} but will decrease the utilization of τ_j, which implies that (3.13)-(3.15) can still be satisfied. However, this contradicts our assumption that $\{\bar{c}_1, \bar{c}_2, ..., \bar{c}_n\}$ is minimal.

Therefore, every task τ_i with $\bar{c}_i > 0$ has the same suspension ratio, which is ξ_{max}. Thus, all such tasks are within the set λ_1, defined above.

The proposed algorithm determines c_i values in exactly the same way: it first checks whether it can find a valid set of c_i values by just considering tasks in λ_1. If there exists no such set, then it treats some suspension time of every task in λ_1 as computation to force them to have the same suspension ratio as tasks in λ_2. As a result λ_1 is redefined to include both the original λ_1 and λ_2. Continuing in this manner, it will find a minimal valid set of c_i values if one exists, because considering tasks in other λ_i sets does not decrease ξ_{max} but will increase both $\sum_{i=1}^{n} c_i$ and such a task's utilization. □

3.3.3 Tardiness Bound

If PSAC finds valid c_i values, then τ is schedulable. In this case, Theorem 3.1 can be applied to compute a tardiness bound, assuming execution costs and suspension times are updated in accordance with the c_i values PSAC produces.

3.3.4 Experimental Evaluation

In this section, we describe experiments conducted using randomly-generated task sets to evaluate the effectiveness of PSAC assuming GEDF scheduling. We compare PSAC with the two extremes of treating no suspension as computation (i.e., using the analysis stated in Section 3.2), denoted NSAC, and the suspension-oblivious approach of treating all suspensions as computation, denoted ASAC. In the variant of PSAC examined here, if valid c_i values cannot be found, then schedulability is checked via ASAC. That is, we exploit the fact that if all suspensions must be viewed as computation, then the less stringent utilization constraint from [41] can be applied.

In our experiments, we generated task sets based upon distributions proposed by Baker [8]. Task periods were uniformly distributed over [10*ms*,100*ms*]. Task utilizations were distributed differently for each experiment using three uniform and three bimodal distributions. The ranges for the uniform distributions were [0.001,0.1] (light), [0.1,0.4] (medium), and [0.4,0.9] (heavy). In the three bimodal distributions, utilizations were distributed uniformly over either [0.001, 0.4) or [0.4, 0.9] with respective probabilities of 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy). Task execution costs were calculated from periods and utilizations. Given that it is common case for real-time workloads to have both self-suspending tasks and computational tasks, we varied U^s_{sum} as follows: $U^s_{sum} = 0.1 \cdot U_{sum}$ (suspensions are relatively infrequent), $U^s_{sum} = 0.4 \cdot U_{sum}$ (suspensions are moderately frequent), and $U^s_{sum} = 0.7 \cdot U_{sum}$ (suspensions are frequent). Moreover, we varied ξ_{max} as follows: 0.1 (suspensions are short), 0.3 (suspensions are moderate), and 0.6 (suspensions are long). Table 3.2 shows suspension-length ranges generated by these parameters. U_{sum} was varied within $\{1, 2, ..., 8\}$. For each combination of (task utilization distribution, U^s_{sum}, U_{sum}, ξ_{max}), 1,000 task sets were generated for an eight-processor system. For each generated system, soft real-time schedulability (i.e., the ability to ensure bounded tardiness) was checked for PSAC, NSAC, and ASAC. In all figures presented in this section, the label "**PSAC-s(m/l)**" indicates

suspension length / per-task utilization		short suspensions ξ_{max} = 0.1	moderate suspensions ξ_{max} = 0.3	long suspensions ξ_{max} = 0.6
light	min:	26 *μs*	409 *μs*	750 *μs*
	avg:	144 *μs*	2.25 *ms*	4.125 *ms*
	max:	263 *μs*	4.09 *ms*	7.5 *ms*
medium	min:	131 *μs*	2.05 *ms*	3.75 *ms*
	avg:	723 *μs*	11.25 *ms*	20.6 *ms*
	max:	1.3 *ms*	20.5 *ms*	37.5 *ms*
heavy	min:	342 *μs*	5.3 *ms*	9.75 *ms*
	avg:	1.9 *ms*	29.25 *ms*	53.625 *ms*
	max:	3.42 *ms*	53.2 *ms*	97.5 *ms*

Table 3.2: Per-job suspension-length ranges.

the PSAC approach assuming short (moderate/long) suspensions. Similar "**NSAC**" and "**ASAC**" labels are used for NSAC and ASAC. Each figure gives three curves per tested approach for the cases of short, moderate, and long suspensions, respectively.

Schedulability results obtained using uniform and bimodal light task utilization distributions are shown in Figures 3.14-3.19. Each curve plots the fraction of the generated task sets the corresponding approach successfully scheduled, as a function of total utilization.

As shown in Figures 3.14-3.19, PSAC proved to be able to significantly improve schedulability in most tested scenarios. Both NSAC and ASAC are negatively impacted when increasing task utilizations, U^s_{sum}, or ξ_{max}. PSAC tries to treat partial suspensions as computation, which effectively decreases the value of ξ_{max} at the cost of only marginally increasing the left side of (3.12). By examining the right side of (3.12), decreasing ξ_{max} can quite significantly relax the utilization constraint (due to the multiplication factor m). An interesting observation is that when suspensions are short, PSAC can guarantee 100% schedulability in all scenarios. Another interesting observation is that when the total utilization equals 8.0, PSAC yields the same schedulability as NSAC. This is because treating any suspension as computation in this case causes total utilization to exceed 8.0.

Schedulability results obtained using uniform and bimodal medium task utilization distributions are shown in Figures 3.20-3.25. Again, PSAC proved to be able to significantly improve schedulability in all tested scenarios. As shown in Figures 3.20-3.21 and 3.23-3.24, when total utilization is no greater than 6.0 and suspensions are infrequent or moderately frequent, almost 100% of all task sets were schedulable using PSAC. Moreover, if suspensions are less frequent or total utilization is smaller, then PSAC is considerably better than ASAC and NSAC. These trends are due to the fact that when total utilization is small and suspensions are not frequent, the left side of (3.12) is small. This gives PSAC more freedom in treating suspensions as computation. Another interesting observation is that, in comparison to the light-utilization case, the improvement seen in PSAC is less. This may be due to the fact that when task utilizations become higher, U_L^c becomes larger, which constrains PSAC's ability to treat suspensions as computation.

Schedulability results obtained using uniform and bimodal heavy task utilization distributions are shown in Figure 3.26-3.31. Once again, PSAC proved to be able to improve schedulability in all tested scenarios. When total utilization is less than 7.0 and suspensions are short, PSAC can achieve almost 100% schedulability. Interestingly, PSAC exhibited greater improvement in the case of bimodal task utilization distributions, as shown in Figures 3.29-3.31. When using uniform heavy task utilization distributions, every self-suspending task or computational task has a heavy utilization and U_{sum}^s and U_L^c tend to be large, which constrains PSAC. On the other hand, when using bimodal heavy task utilization distributions, U_{sum}^s and U_L^c have a higher chance of being smaller than in the uniform case, which constrains PSAC less. Moreover, when using bimodal distributions, generated task sets tend to have fewer tasks that have long suspensions. Therefore, PSAC can treat less suspension time as computation and has a better chance of finding valid c_i values.

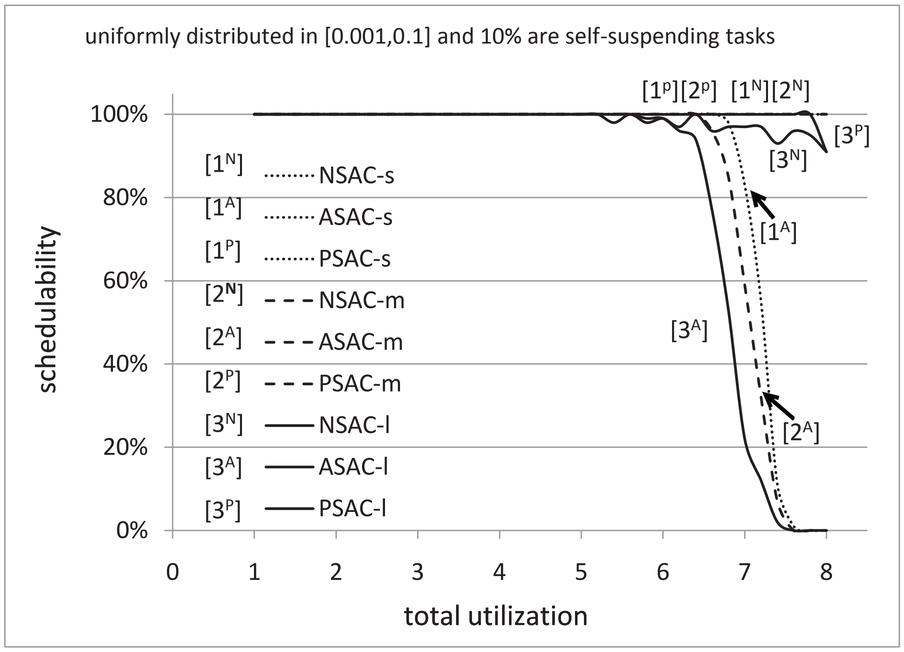


Figure 3.14: Soft real-time schedulability results for uniform light task utilization distributions. Relatively infrequent suspensions are assumed.

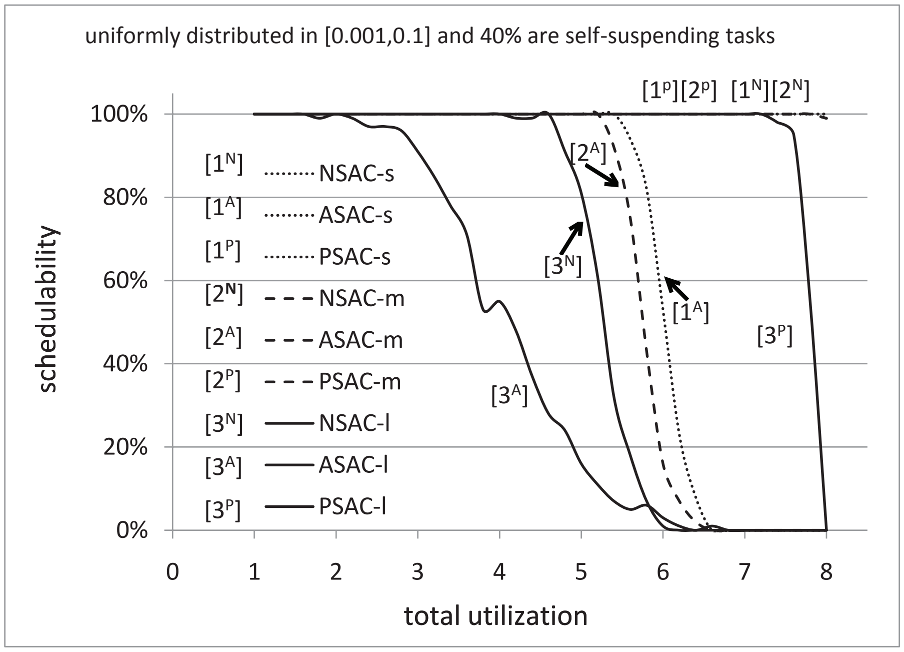


Figure 3.15: Soft real-time schedulability results for uniform light task utilization distributions. Moderately frequent suspensions are assumed.

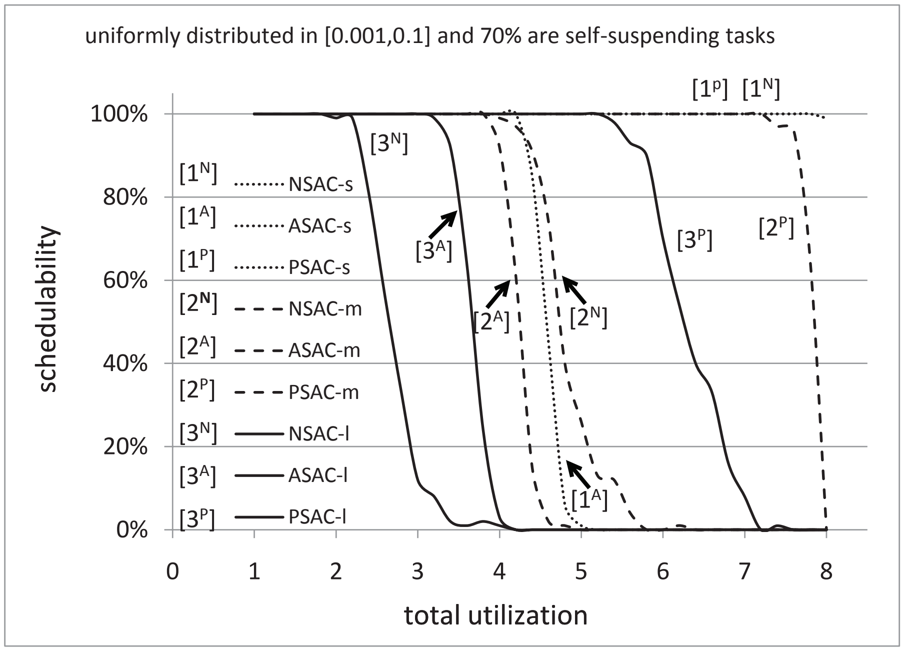


Figure 3.16: Soft real-time schedulability results for uniform light task utilization distributions. Frequent suspensions are assumed.

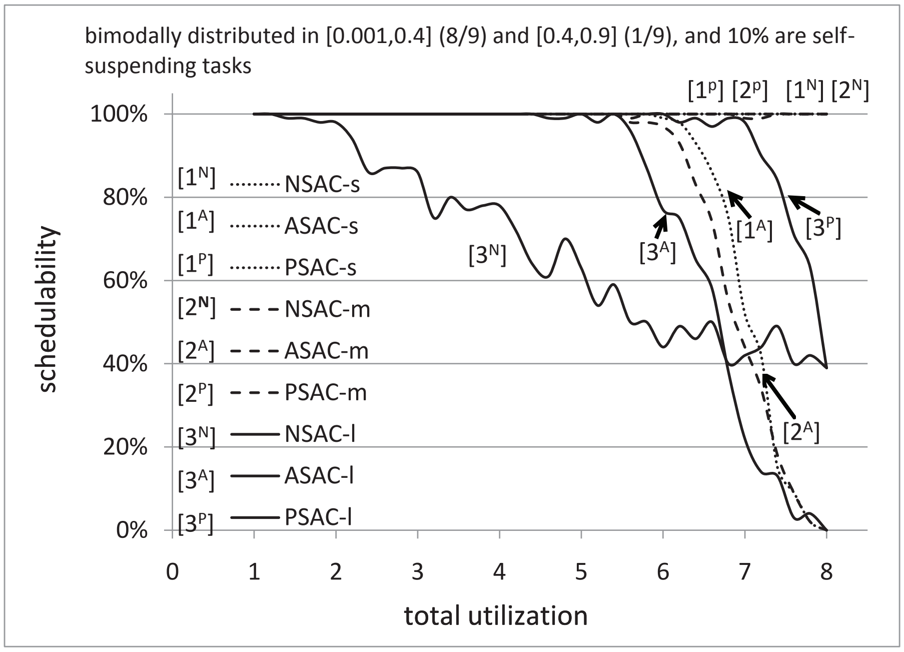


Figure 3.17: Soft real-time schedulability results for bimodal light task utilization distributions. Relatively infrequent suspensions are assumed.

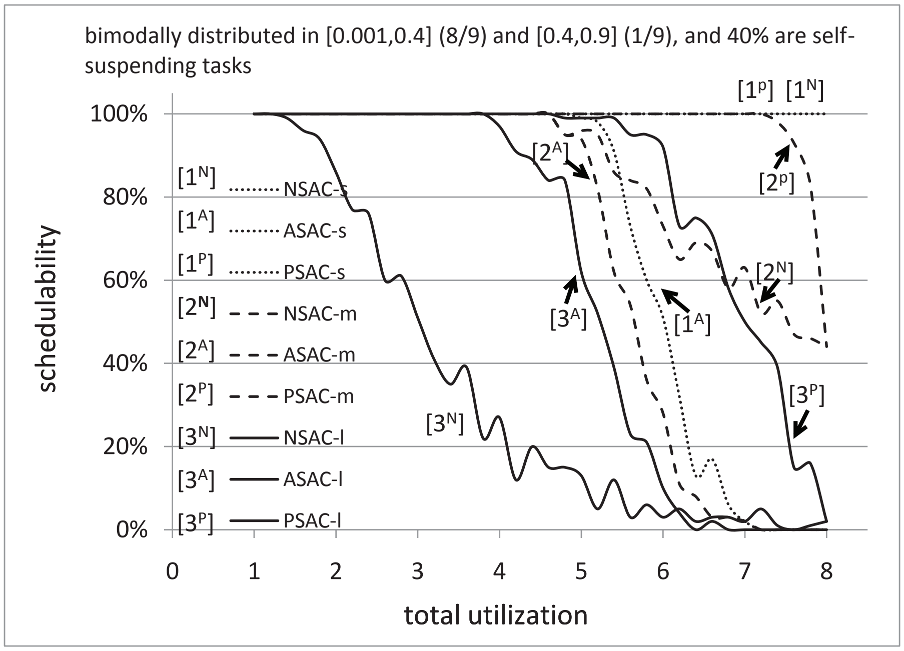


Figure 3.18: Soft real-time schedulability results for bimodal light task utilization distributions. Moderately frequent suspensions are assumed.

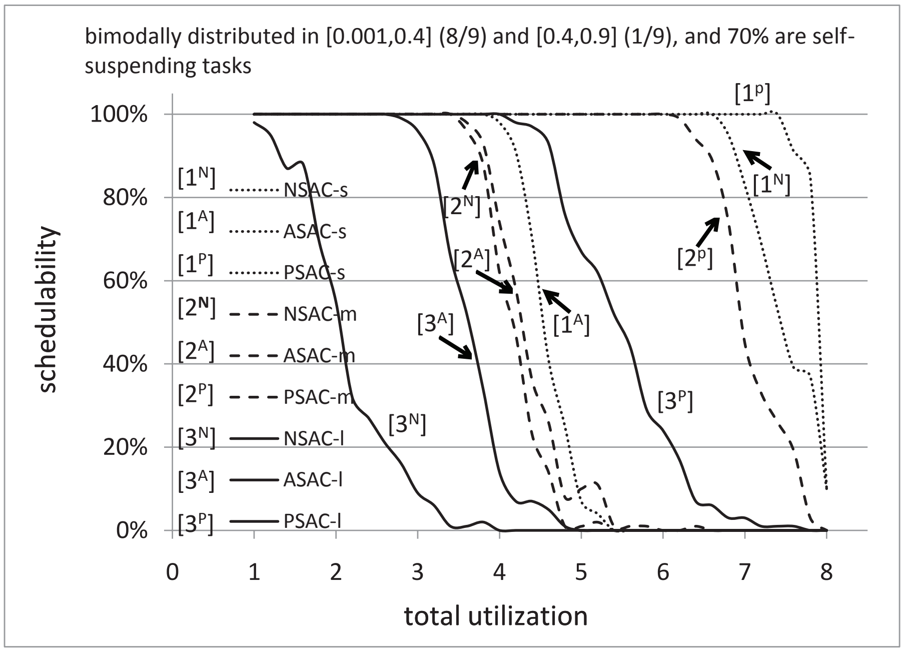


Figure 3.19: Soft real-time schedulability results for bimodal light task utilization distributions. Frequent suspensions are assumed.

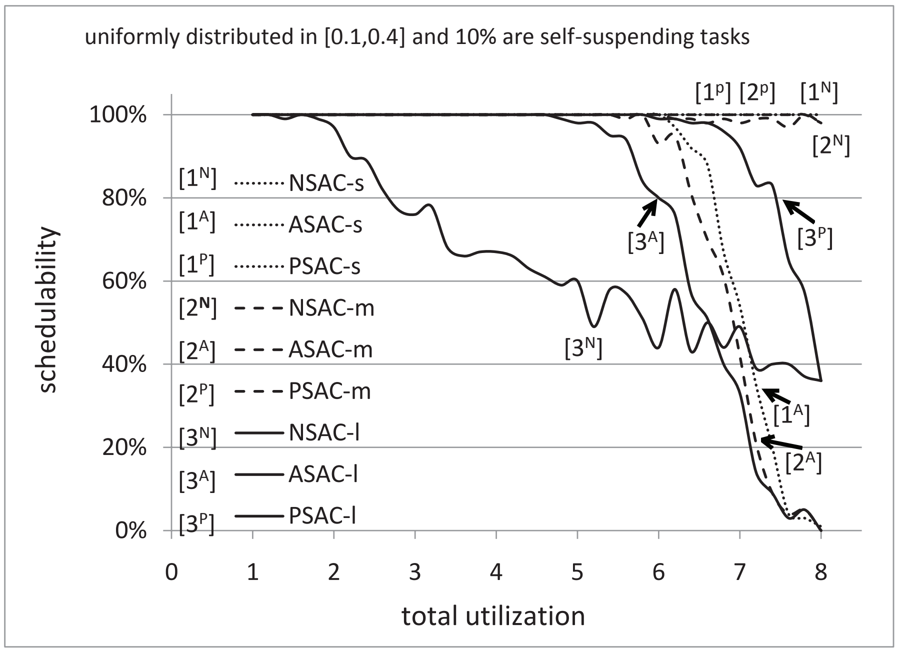


Figure 3.20: Soft real-time schedulability results for uniform medium task utilization distributions. Relatively infrequent suspensions are assumed.

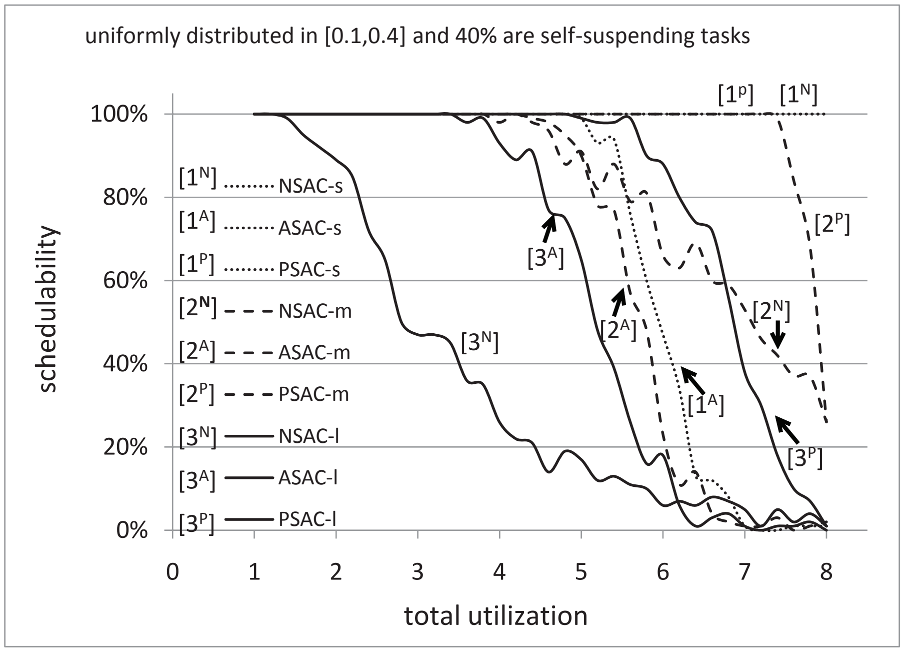


Figure 3.21: Soft real-time schedulability results for uniform medium task utilization distributions. Moderately frequent suspensions are assumed.

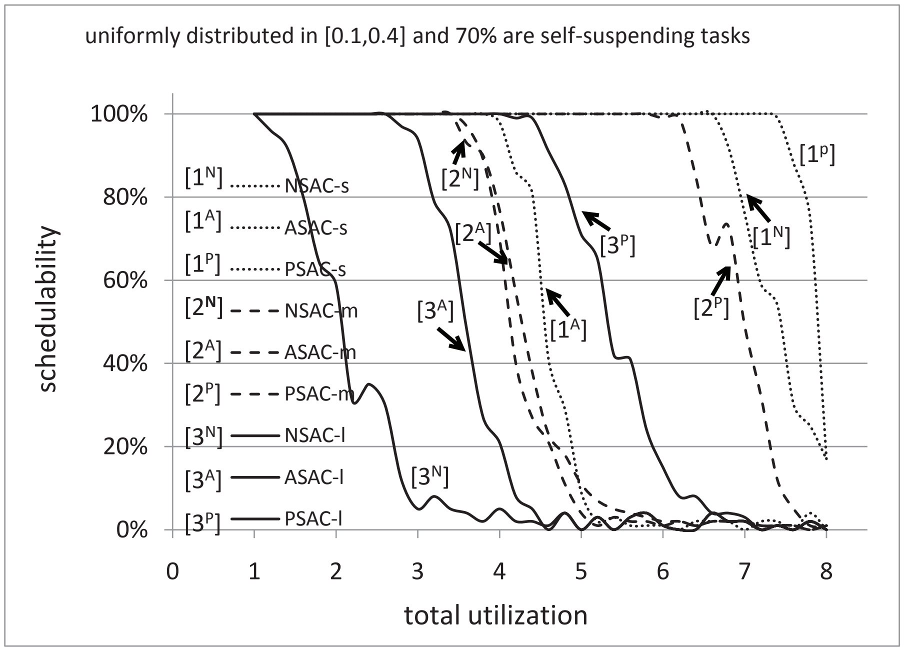


Figure 3.22: Soft real-time schedulability results for uniform medium task utilization distributions. Frequent suspensions are assumed.

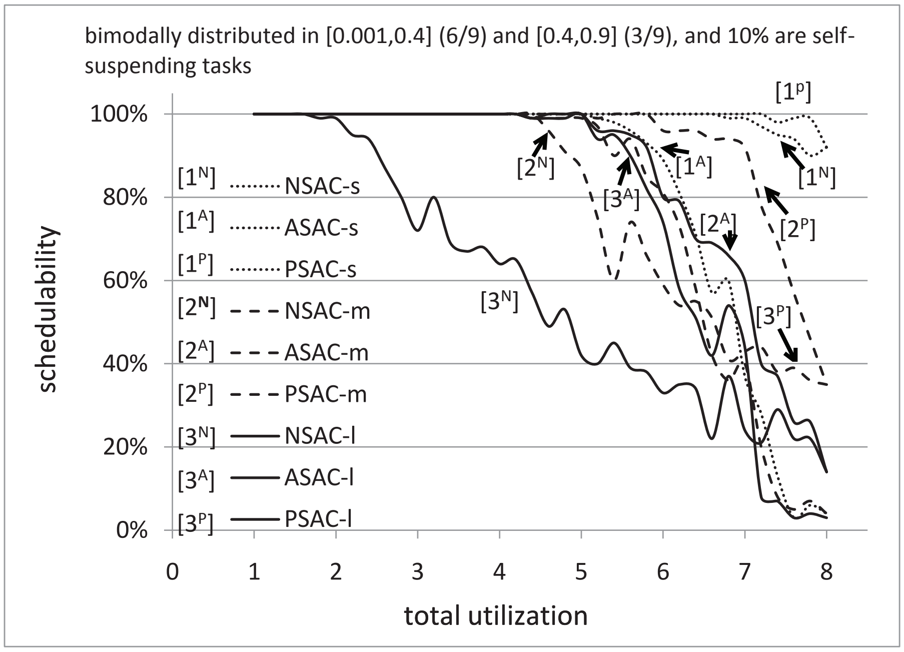


Figure 3.23: Soft real-time schedulability results for bimodal medium task utilization distributions. Relatively infrequent suspensions are assumed.

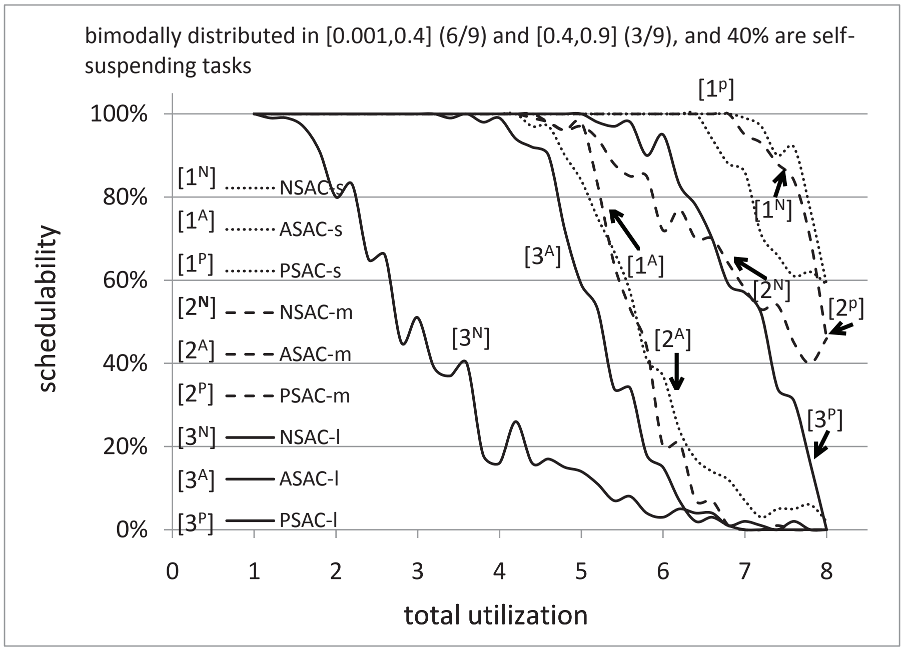


Figure 3.24: Soft real-time schedulability results for bimodal medium task utilization distributions. Moderately frequent suspensions are assumed.

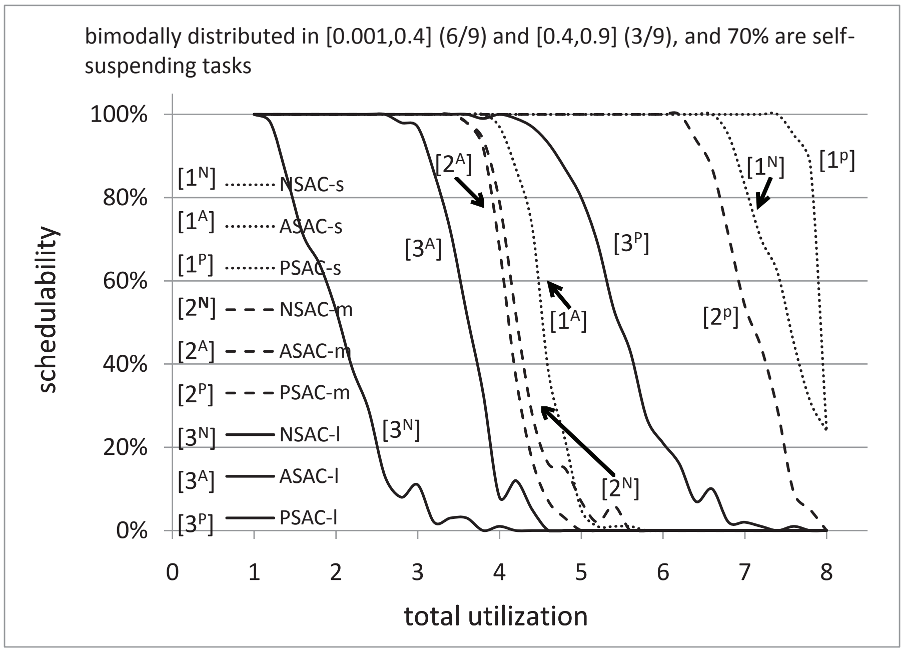


Figure 3.25: Soft real-time schedulability results for bimodal medium task utilization distributions. Frequent suspensions are assumed.

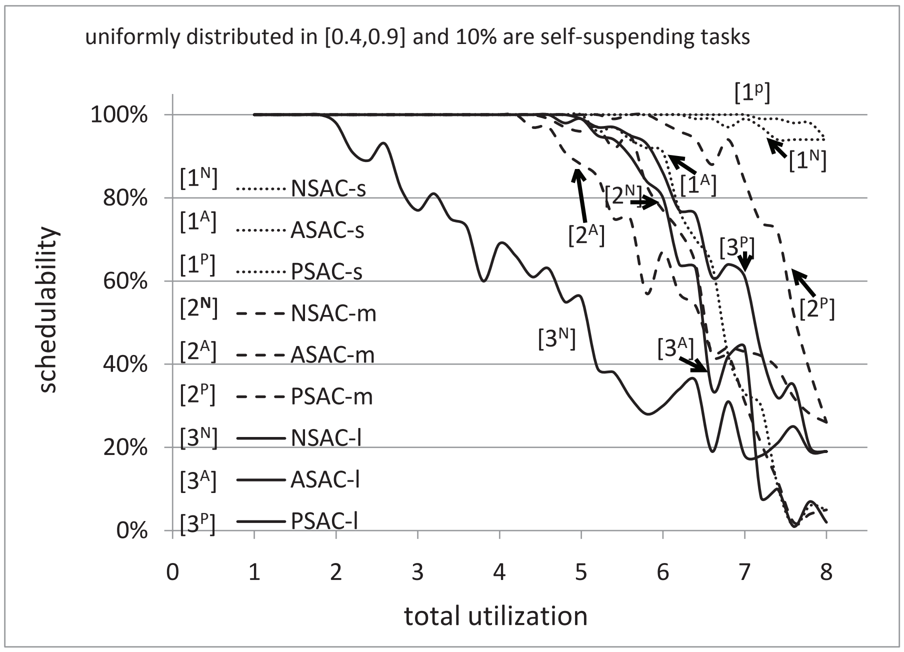


Figure 3.26: Soft real-time schedulability results for uniform heavy task utilization distributions. Relatively infrequent suspensions are assumed.

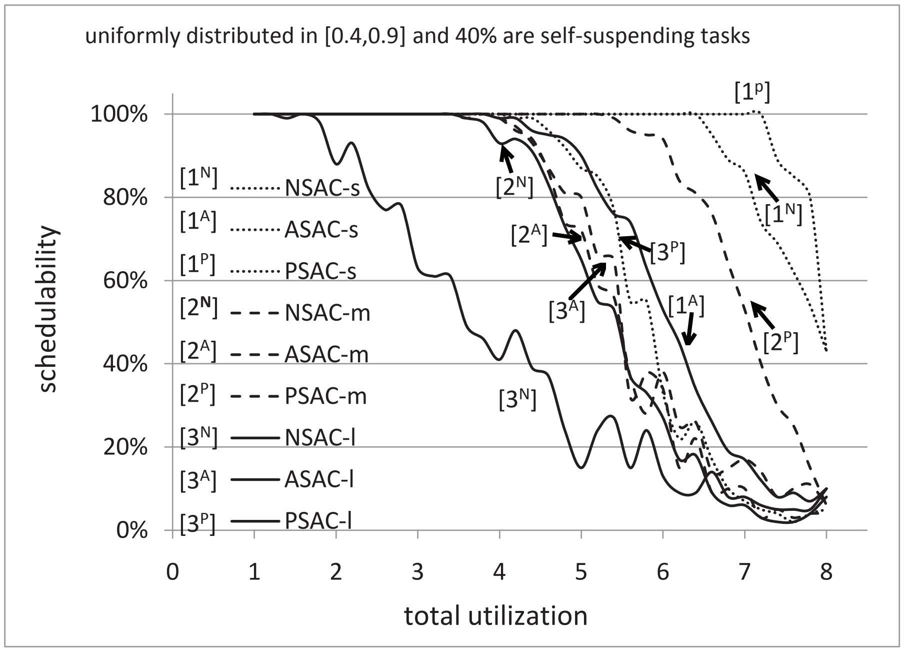


Figure 3.27: Soft real-time schedulability results for uniform heavy task utilization distributions. Moderately frequent suspensions are assumed.

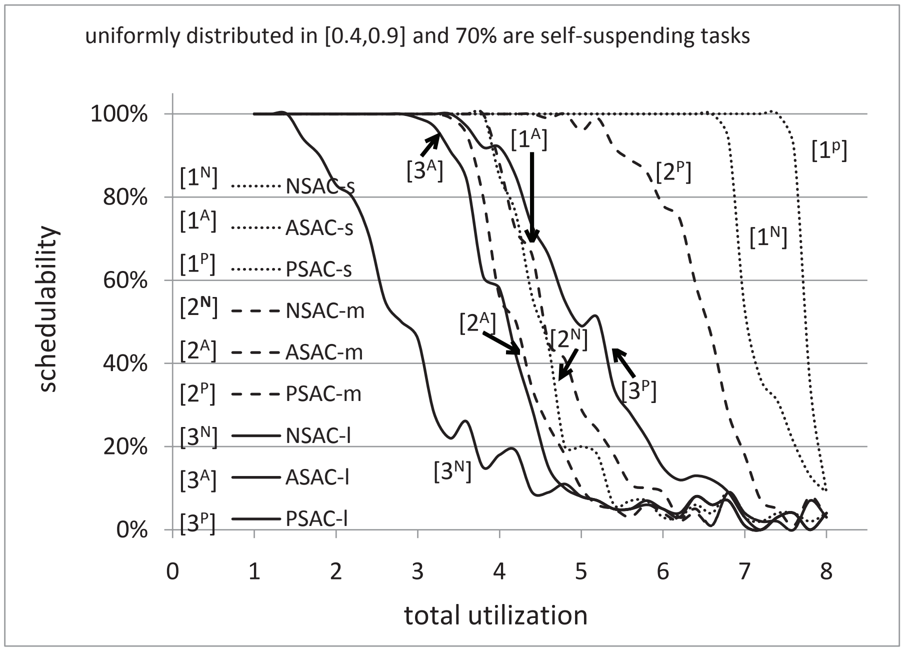


Figure 3.28: Soft real-time schedulability results for uniform heavy task utilization distributions. Frequent suspensions are assumed.

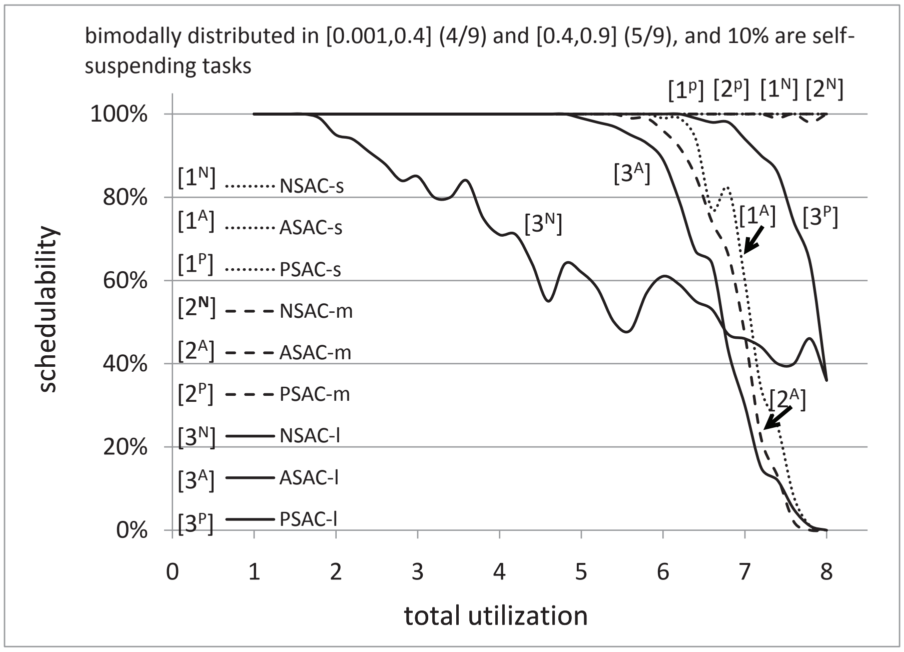


Figure 3.29: Soft real-time schedulability results for bimodal heavy task utilization distributions. Relatively infrequent suspensions are assumed.

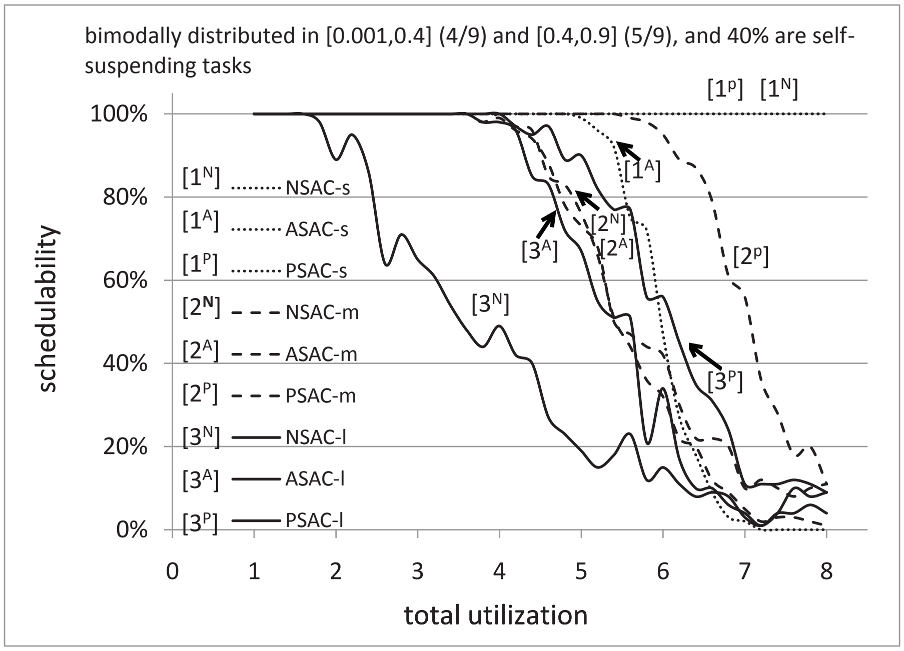


Figure 3.30: Soft real-time schedulability results for bimodal heavy task utilization distributions. Moderately frequent suspensions are assumed.

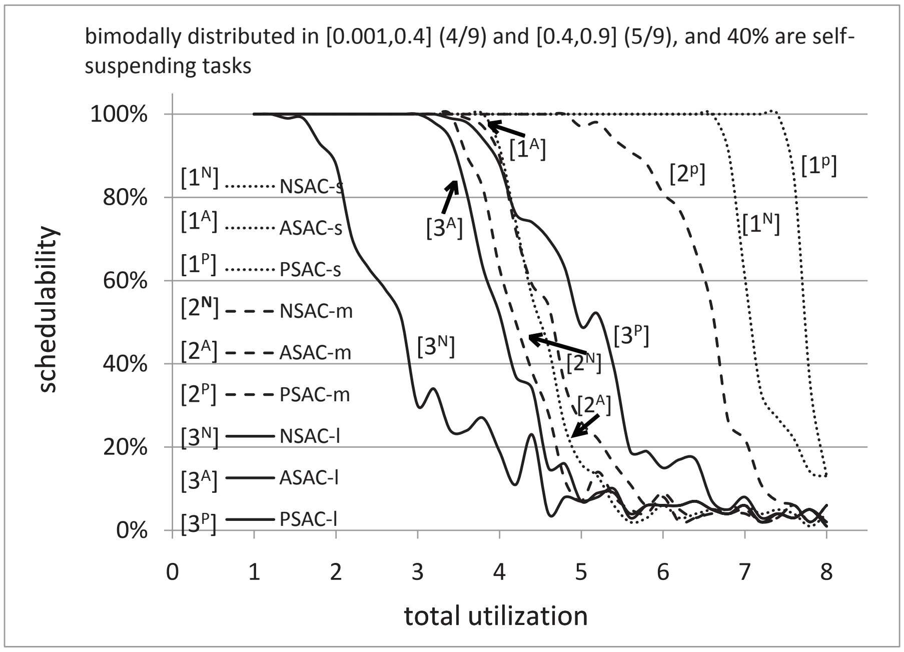


Figure 3.31: Soft real-time schedulability results for bimodal heavy task utilization distributions. Frequent suspensions are assumed.

3.4 An O(m) Schedulability Test

As described in Section 3.2, the proposed suspension-aware analysis for globally scheduled SRT sporadic self-suspending task systems improves upon the suspension-oblivious analysis for many task systems. Unfortunately, it does not fully address the root cause of pessimism due to suspensions, and thus may still cause significant capacity loss. Indeed, both suspension-aware analysis (presented in Section 3.2) and suspension-oblivious analysis yield $O(n)$ utilization loss[4] where n is the number of self-suspending tasks in the system.

The cause of pessimism in prior analysis. A key step in prior suspension-aware analysis presented in Section 3.2 involves bounding the number of tasks that have enabled jobs (i.e., eligible for executing or suspending) at a specifically defined non-busy time instant t (i.e., at least one processor is idle at t). For ordinary task systems without suspensions, this number of tasks can be safely upper-bounded by $m - 1$, where m is the number of processors, for otherwise, t would be busy. For sporadic self-suspending task systems, however, idle instants can exist due to suspensions even if m or more tasks have enabled jobs. The worst-case scenario that serves as the root source of pessimism in prior analysis is the following: *all n self-suspending tasks have jobs that suspend at some time t simultaneously, thus causing t to be non-busy*.

Key observation that motivates the approach in this section. Interestingly, the suspension-oblivious approach eliminates the worst-case scenario just discussed, albeit at the expense of pessimism elsewhere in the analysis. That is, by converting all n tasks' suspensions into computation, the worst-case scenario is avoided because then at most $m - 1$ tasks can have enabled jobs at any non-busy time instant. *However, converting all n tasks' suspensions into computation is clearly overkill when attempting to avoid the worst-case scenario; rather, converting at most m tasks' suspensions into computation should suffice.* This observation motivates the new analysis technique we propose, which yields a much improved schedulability test with only $O(m)$ suspension-related utilization loss. Recent experimental results suggest that global algorithms should be limited to (sub-)systems with modest core counts (e.g., up to eight cores) [15]. Thus, m is likely to be small and much less than n in many settings.

[4]Task systems exist for which utilization loss is $\Theta(n)$ under either of these analysis approaches.

In the rest of this section, we derive a GEDF[5] schedulability test that only results in an $O(m)$ capacity loss.

3.4.1 Schedulability Analysis

We now present our proposed new schedulability analysis for SRT sporadic self-suspending task systems.

We focus on a given sporadic self-suspending task system τ. Let $\tau_{l,j}$ be a job of task τ_l in τ, $t_d = d_{l,j}$, and S be a GEDF schedule for τ with the following property.

(P1) The tardiness of every job $\tau_{i,k}$, where $\tau_{i,k}$ has higher priority than $\tau_{l,j}$, is at most $x + e_i + s_i$ in S, where $x \geq 0$.

Our objective is to determine the smallest x such that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$. This would by induction imply a tardiness of at most $x + e_i + s_i$ for all jobs of every task τ_i, where $\tau_i \in \tau$. We assume that $\tau_{l,j}$ finishes after t_d, for otherwise, its tardiness is trivially zero. The steps for determining the value for x are as follows.

1. Determine a lower bound on the amount of work pending for tasks in τ that can compete with $\tau_{l,j}$ after t_d, required for the tardiness of $\tau_{l,j}$ to exceed $x + e_l + s_l$. This is dealt with in Lemma 3.6 in Section 3.4.3.

2. Determine an upper bound on the work pending for tasks in τ that can compete with $\tau_{l,j}$ after t_d. This is dealt with in Lemmas 3.7 and 3.8 in Section 3.4.4.

3. Determine the smallest x such that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$, using the above lower and upper bounds. This is dealt with in Theorem 3.3 in Section 3.4.5.

Definition 3.12. We categorize jobs based on the relationship between their priorities and those of $\tau_{l,j}$:

$$\mathbf{d} = \{\tau_{i,v} : (d_{i,v} < t_d) \vee (d_{i,v} = t_d \wedge i \leq l)\}.$$

[5]We specifically focus on GEDF in this work. As discussed in Chapter 8, a useful future work is to enable more general results by considering more general global schedulers such as GSA and other window-constrained scheduling algorithms.

By Definition 3.12, **d** is the set of jobs with deadlines at most t_d with priority at least that of $\tau_{l,j}$. These jobs do not execute beyond t_d in the PS schedule (as defined in Definition 2.5). Note that $\tau_{l,j}$ is in **d**. Also note that jobs not in **d** have lower priority than those in **d** and thus do not affect the scheduling of jobs in **d**. For simplicity, we will henceforth assume that jobs not in **d** do not execute in either the GEDF schedule S or the corresponding PS schedule. To avoid distracting "boundary cases," we also assume that the schedule being analyzed is prepended with a schedule in which no deadlines are missed that is long enough to ensure that all previously released jobs referenced in the proof exist.

Similar to Section 3.2, our schedulability test is obtained by comparing the allocations to **d** in the GEDF schedule S and the corresponding PS schedule, both on m processors, and quantifying the difference between the two. We analyze task allocations on a per-task basis.

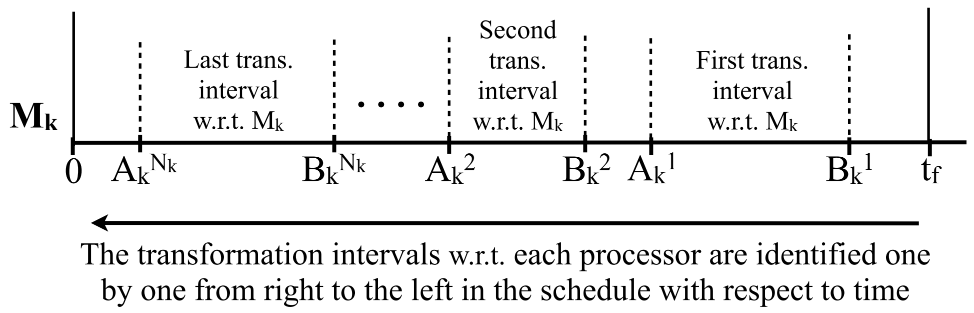
Definition 3.13. A time instant t is *busy* (respectively, *non-busy*) for a job set J if all (respectively, not all) m processors execute jobs in J at t. A time interval is *busy* (respectively, *non-busy*) for J if each instant within it is busy (respectively, non-busy) for J. A time instant t is *busy on processor* M_k (respectively, *non-busy on processor* M_k) for J if M_k executes (respectively, does not execute) a job in J at t. A time interval is *busy on processor* M_k (respectively, *non-busy on processor* M_k) for J if each instant within it is busy (respectively, non-busy) on M_k for J.

The following claim follows from the definition of LAG (the concepts of lag and LAG are presented in Section 2.2) .

Claim 5. If $LAG(\mathbf{d}, t_2, S) > LAG(\mathbf{d}, t_1, S)$, where $t_2 > t_1$, then $[t_1, t_2)$ is non-busy for **d**. In other words, *LAG* for **d** can increase only throughout a non-busy interval for **d** .

3.4.2 New $O(m)$ Analysis Technique

By Claim 5 and the above discussion, the pessimism of analyzing sporadic self-suspending task systems is due to the worst-case scenario where all sporadic self-suspending tasks might have enabled jobs that suspend at a time instant t, making t non-busy; this can result in non-busy intervals in which LAG for **d** increases. Specifically, the worst-case scenario happens when *at least m suspending tasks have enabled tardy jobs with deadlines at or before a non-busy time instant t where such jobs*



M_k
Last trans. interval w.r.t. M_k
Second trans. interval w.r.t. M_k
First trans. interval w.r.t. M_k
0
$A_k^{N_k}$
$B_k^{N_k}$
A_k^2
B_k^2
A_k^1
B_k^1
t_f
The transformation intervals w.r.t. each processor are identified one by one from right to the left in the schedule with respect to time

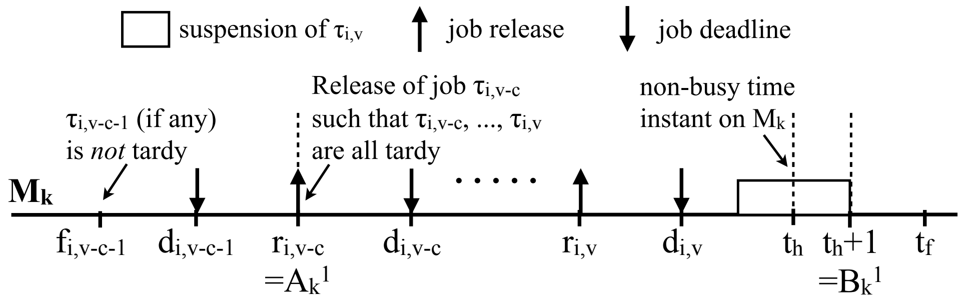
Figure 3.32: Transformation intervals with respect to M_k.

suspend at t. (As seen in the analysis in Sections 3.4.3 and 3.4.4, suspensions of non-tardy jobs are not problematic.)

Thus, our goal is to avoid such a worst case. The key idea behind our new technique is the following: *At any non-busy time* t*, if* k *processors* ($1 \leq k \leq m$) *are idle at* t *while at least* k *suspending tasks have enabled tardy jobs with deadlines at or before* t *that suspend simultaneously at* t*, then, by treating suspensions of* k *jobs of* k *such tasks to be computation at* t*,* t *becomes busy. Treating the suspensions of all such tasks to be computation is needlessly pessimistic.*

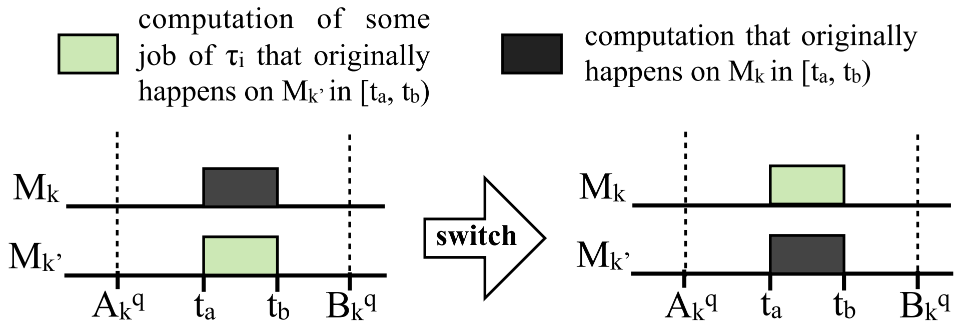
Let t_f denote the end time of the schedule S. Our new technique involves transforming the entire schedule S within $[0, t_f)$ from right to left (i.e., from time t_f to time 0) to obtain a new schedule $\overline{S}$ as described below. The goal of this transformation is to convert certain tardy jobs' suspensions into computation in non-busy time intervals to eliminate idleness as discussed above. For any job $\tau_{i,v}$, if its suspensions are converted into computation in a time interval $[t_1, t_2)$, then $\tau_{i,v}$ is considered to execute in $[t_1, t_2)$. We transform S to $\overline{S}$ by applying m *transformation steps*, where in the k^{th} step, the schedule is transformed with respect to processor M_k. Let $S^0 = S$ denote the original schedule, S^k denote the transformed schedule after performing the k^{th} transformation step, and $S^m = \overline{S}$ denote the final transformed schedule. The k^{th} transformation step works as follows.

Transformation method. By analyzing the schedule S^{k-1} on M_k, we first define $N_k \geq 0$ *transformation intervals* denoted $[A_k^1, B_k^1)$, $[A_k^2, B_k^2)$, ..., $[A_k^{N_k}, B_k^{N_k})$ ordered from right to left with respect to time, as illustrated in Figure 3.32. These transformation intervals are the only intervals that are affected by the k^{th} transformation step. We identify these transformation intervals by moving from



suspension of τi,v
job release
job deadline
τi,v-c-1 (if any) is *not* tardy
Release of job τi,v-c such that τi,v-c, ..., τi,v are all tardy
non-busy time instant on Mk
Mk
fi,v-c-1
di,v-c-1
ri,v-c =Ak1
di,v-c
ri,v
di,v
th
th+1 =Bk1
tf

Figure 3.33: Defining t_h and $r_{i,v-c}$ in the transformation method.



computation of some job of τi that originally happens on Mk' in [ta, tb)
computation that originally happens on Mk in [ta, tb)
Mk
Mk'
Akq
ta
tb
Bkq
switch

Figure 3.34: *Switch:* switch the computation of τ_i originally executed on $M_{k'}$ to M_k.

right to left with respect to time in the schedule S^{k-1} considering allocations on processor M_k. (An extended example illustrating the entire transformation method will be given later.)

Moving from time t_f to the left in S^{k-1}, let t_h denote the first encountered non-busy time instant on M_k where at least one task τ_i has an enabled job $\tau_{i,v}$ suspending at t_h where

$$d_{i,v} \leq t_h. \tag{3.22}$$

(If t_h does not exist, then we have $S^k = S^{k-1}$.) Let $v - c$ ($0 \leq c \leq v - 1$) denote the minimum job index of τ_i such that all jobs $\tau_{i,v-c}, \tau_{i,v-c+1}, ..., \tau_{i,v}$ are tardy, as illustrated in Figure 3.33. Then, $[r_{i,v-c}, t_h + 1)$ is the first transformation interval with respect to M_k, i.e., $[A_k^1, B_k^1) = [r_{i,v-c}, t_h + 1)$.

To find the next transformation interval on M_k, further moving from A_k^1 to the left in S^{k-1}, find the next t_h, $\tau_{i,v}$, and $\tau_{i,v-c}$ applying the same definitions given above. If they exist, then the newly founded interval $[r_{i,v-c}, t_h + 1)$ is the second transformation interval $[A_k^2, B_k^2)$. Such a process

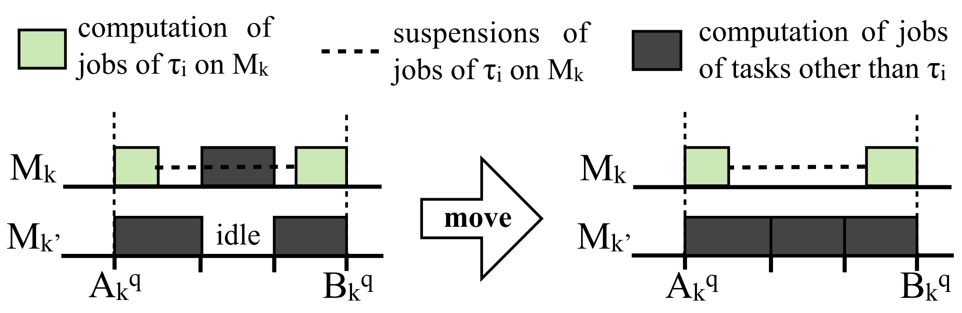

computation of jobs of τi on Mk
suspensions of jobs of τi on Mk
computation of jobs of tasks other than τi
Mk
Mk'
idle
Ak^q
Bk^q
move
Mk
Mk'
Ak^q
Bk^q

Figure 3.35: *Move:* move the computation of tasks other than τ_i from M_k to some idle processor $M_{k'}$.

continues, moving from right to the left in S^{k-1}, until time 0 is reached before any such t_h is found. If task τ_i is selected to define the transformation interval $[A_k^q, B_k^q)$ in the manner just described, then we say that τ_i is *associated* with this transformation interval; each transformation interval has exactly one associated task. (As seen below, when performing transformation steps after the k^{th} step, τ_i cannot be selected again for defining transformation intervals within $[A_k^q, B_k^q)$.) According to the way we identify transformation intervals as described above, the following property holds.

(G1) Successive transformation intervals with respect to processor M_k do not overlap, i.e., $B_k^q \leq A_k^{q-1}$ holds where $2 \leq q \leq N_k$.

After identifying all transformation intervals with respect to M_k, we perform the following three operations on each of these N_k transformation intervals, starting with $[A_k^1, B_k^1)$. In the following, let $[A_k^q, B_k^q)$ $(1 \leq q \leq N_k)$ denote the currently considered transformation interval and let τ_i be the associated task.

1. Switch: We assume that all computations of jobs of τ_i occurring within $[A_k^q, B_k^q)$ happen on M_k. This is achieved by switching any computation of τ_i in any interval $[t_a, t_b) \subseteq [A_k^q, B_k^q)$ originally executed on some processor $M_{k'}$ other than M_k with the computation (if any) occurring in $[t_a, t_b)$ on M_k, as illustrated in Figure 3.34.

2. Move: Then for all intervals in $[A_k^q, B_k^q)$ on M_k where jobs not belonging to τ_i execute while some job of τ_i suspends, if any of such interval is non-busy (at least one processor is idle in this interval), then we also move the computation occurring within this interval on M_k to some processor

$M_{k'}$ that is idle in the same interval, as illustrated in Figure 3.35. This guarantees that all intervals in $[A_k^q, B_k^q)$ on M_k where jobs not belonging to τ_i execute are busy on all processors.

Due to the fact that all jobs of τ_i enabled in $[A_k^q, B_k^q)$ (i.e., $\tau_{i,v-c}, \tau_{i,v-c+1}, ..., \tau_{i,v})$[6] are tardy, interval $[A_k^q, B_k^q)$ on M_k consists of three types of subintervals: (*i*) those in which jobs of τ_i enabled within $[A_k^q, B_k^q)$ are executing, (*ii*) those in which jobs of τ_i enabled within $[A_k^q, B_k^q)$ are suspending (note that jobs of tasks other than τ_i may also execute on M_k in such subintervals; if this is the case, then the move operation ensures that any such subinterval is busy on all processors), and (*iii*) those in which jobs of τ_i enabled within $[A_k^q, B_k^q)$ are preempted. Thus, within any non-busy interval on M_k in $[A_k^q, B_k^q)$, jobs of τ_i must be suspending (for otherwise this interval would be busy on M_k). Therefore, we perform the third transformation operation as follows.

3. Convert: Within all time intervals that are non-busy on M_k in $[A_k^q, B_k^q)$, convert the suspensions of all jobs of τ_i enabled within $[A_k^q, B_k^q)$ into computation, as illustrated in Figure 3.36. This guarantees that M_k is busy within $[A_k^q, B_k^q)$. Since all such jobs belong to the associated task τ_i of $[A_k^q, B_k^q)$, by the definitions of A_k^q and B_k^q as described above, the following property holds.

(G2) For any transformation interval $[A_k^q, B_k^q)$, jobs whose suspensions are converted into computation in this interval in the k^{th} transformation step have releases and deadlines within this interval.

When performing any later transformation step $k' > k$ (i.e., with respect to processor $M_{k'}$), τ_i clearly cannot be selected again for its suspensions to be converted into computation in idle intervals on $M_{k'}$ within $[A_k^q, B_k^q)$. Moreover, since all intervals within $[A_k^q, B_k^q)$ on M_k where jobs not belonging to τ_i execute are busy on all processors, any switch or move operation performed in later transformation steps does not change the fact that $[A_k^q, B_k^q)$ is busy on M_k in the final transformed schedule $\overline{S}$. Note that the above switch, move, and convert operations do not affect the start and completion times of any job.

After performing the switch, move, and convert operations on $[A_k^q, B_k^q)$ as described above, the transformation within $[A_k^q, B_k^q)$ is complete. We then consider the next transformation interval

[6]Note that, according to the way we select $v - c$, job $\tau_{i,v-c-1}$ is not enabled within $[A_k^q, B_k^q)$ because it is not tardy and thus completes at or before $d_{i,v-c-1} \leq r_{i,v-c} = A_k^q$.

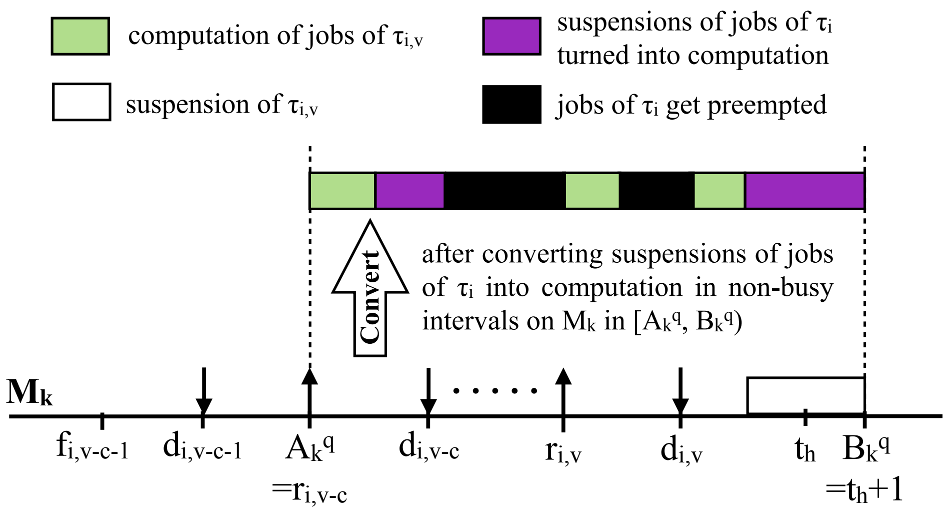

computation of jobs of τi,v
suspensions of jobs of τi turned into computation
suspension of τi,v
jobs of τi get preempted
after converting suspensions of jobs of τi into computation in non-busy intervals on Mk in [Akq, Bkq)
Convert
Mk
fi,v-c-1
di,v-c-1
Akq =ri,v-c
di,v-c
ri,v
di,v
th
Bkq =th+1

Figure 3.36: *Convert:* convert the suspensions of all jobs of τ_i that are enabled within $[A_k^q, B_k^q)$ (i.e., $\tau_{i,v-c}$, ..., $\tau_{i,v}$) into computation within all non-busy time intervals on M_k in $[A_k^q, B_k^q)$.

$[A_k^{q+1}, B_k^{q+1})$, and so on. The k^{th} transformation step is complete when all such transformation intervals have been considered, from which we obtain S^k.

Repeating this entire process, we similarly obtain S^{k+1}, S^{k+2}, ..., $S^m = \overline{S}$.

Analysis. The transformation method above ensures the following.

Claim 6. At any non-busy time instant $t \in [0, t_f)$ in the transformed schedule $\overline{S}$, at most $m - 1$ tasks can have enabled tardy jobs with deadlines at or before t.

Proof. Suppose that $t \in [0, t_f)$ is non-busy in $\overline{S}$, and there are z idle processors at t. Assume that m or more tasks have enabled tardy jobs at t with deadlines at or before t. Then at least z such tasks have enabled tardy jobs suspending at t in order for t to be non-busy. However, our transformation method would have converted the suspension time at t of z such jobs into computation, which makes t busy on M_k, a contradiction. □

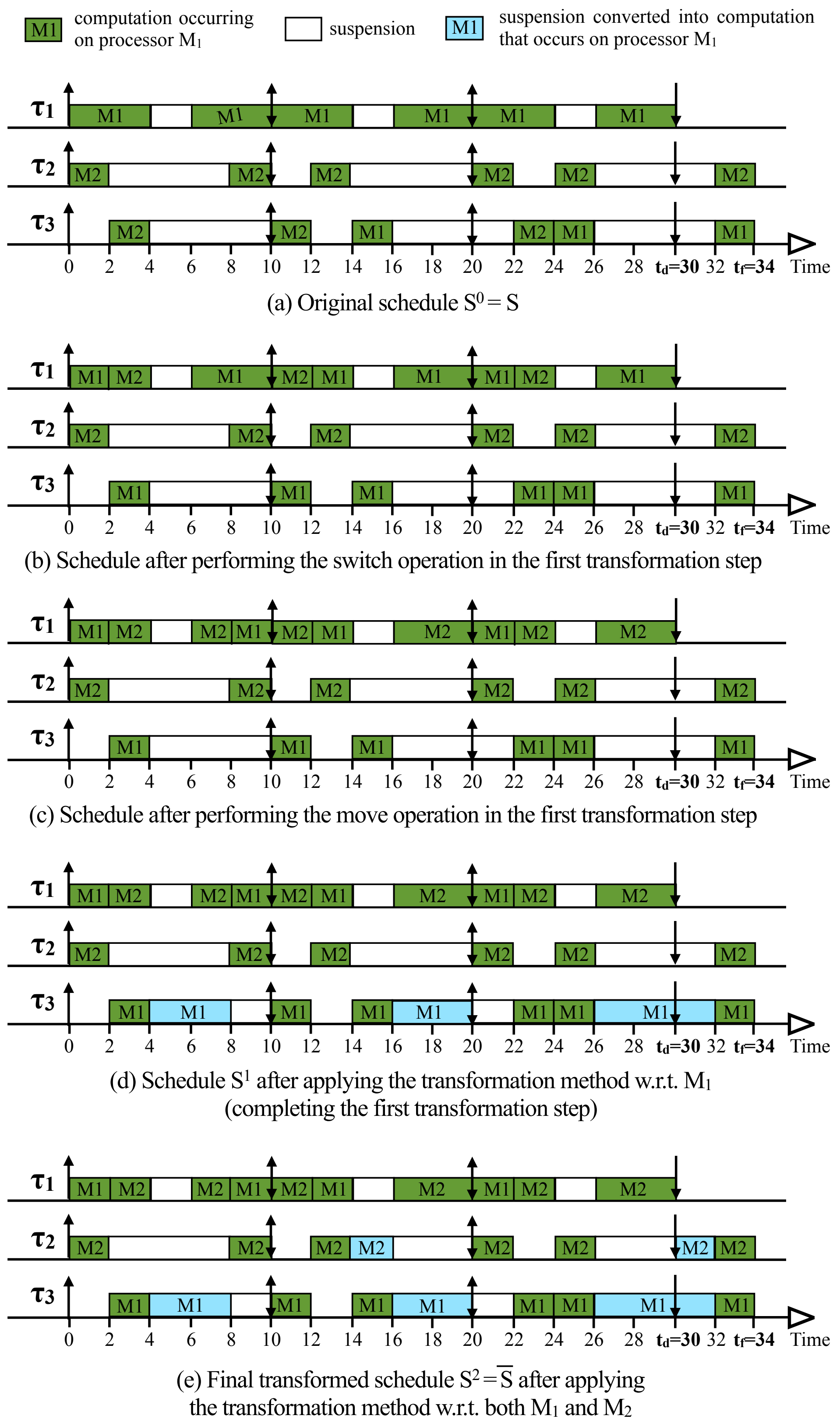
Example 1. Consider a two-processor task set τ that consists of three tasks: τ_1 = (e4, s2, e4, 10), and τ_2 = τ_3 = (e2, s6, e2, 10). Figure 3.37(a) shows the original GEDF schedule S for the time interval [0,34). Assume $\tau_{3,3}$ is the analyzed job so that $t_d = 30$ and $t_f = 34$. By the transformation method, we first transform the schedule in $[0, 34)$ with respect to processor M_1 (i.e., the first transformation step). Moving from $t_f = 34$ to the left in S, the first idle time instant on M_1 is time 31. At time 31, two jobs $\tau_{2,3}$ and $\tau_{3,3}$ are suspending and both jobs satisfy condition (3.22) since $d_{2,3} = d_{3,3} = 30$.

We arbitrarily choose τ_3 for this transformation step. Since job $\tau_{3,1}$ has the minimum job index of τ_3 such that all jobs $\tau_{3,1}, \tau_{3,2}, \tau_{3,3}$ are tardy, we use τ_3 for the transformation with respect to M_1 up to the release time of $\tau_{3,1}$, which is time 0. Thus, M_1 has only one transformation interval, i.e., $[0, 32)$. By the transformation method, we first perform the switch operation, which switches any computation of jobs of τ_3 in $[0, 32)$ that is not occurring on M_1 to M_1; this includes the computation in $[2, 4)$, $[10, 12)$, and $[22, 24)$. Accordingly, the computations that originally occur in these three intervals on M_1, which are due to jobs of τ_1, are switched to M_2. The resulting schedule after this switching is shown in Figure 3.37(b). After this switching, we apply the move operation, which affects non-busy intervals in $[0, 32)$ in which jobs of τ_3 are suspending while jobs of tasks other than τ_3 are executing on M_1. For this example schedule, $[6, 8)$, $[16, 20)$, and $[26, 30)$ must be considered, and we move the computation of τ_1 in these three intervals from M_1 to M_2. (Note that since intervals $[4, 6)$ and $[30, 32)$ are non-busy on both processors, they are not considered.) The resulting schedule after this moving is shown in Figure 3.37(c). Finally, within all non-busy intervals on M_1 in $[0, 32)$, which include $[4, 8)$, $[16, 20)$, and $[26, 32)$, we convert the suspensions of all enabled jobs of τ_3 in $[0, 32)$ into computation. We thus complete the first transformation step and obtain the schedule S^1, which is shown in Figure 3.37(d). As seen in Figure 3.37(d), after applying the transformation method with respect to M_1, M_1 is fully occupied in interval $[0, 32)$ by the computation of jobs of τ_3, suspensions of jobs of τ_3 that are converted into computation, the computations of jobs not belonging to τ_3 that preempt jobs of τ_3, and the computations of jobs not belonging to τ_3 that occur on M_1 while jobs of τ_3 are suspending. Next, we perform the second transformation step and transform S^1 in $[0, 34)$ with respect to M_2. This yields the final transformed schedule $S^2 = \overline{S}$, as shown in Figure 3.37(e). Notice that M_2 is idle in $[4, 6)$ in $\overline{S}$ because jobs $\tau_{1,1}$ and $\tau_{2,1}$, which suspend in $[4, 6)$, have deadlines (at time 10) after time 6; thus, by the transformation method (see (3.22)) these jobs' suspensions are not turned into computation.

Definition 3.14. Let $\overline{u_i} = \dfrac{e_i + s_i}{p_i} = u_i + \dfrac{s_i}{p_i}$.

Note that in the above definition, $\overline{u_i} \leq 1$ holds for any task $\tau_i \in \tau$ because $e_i + s_i \leq p_i$, as discussed in Section 3.1.

Definition 3.15. The interval $[r_{i,j}, d_{i,j})$ is called the *job execution window* for job $\tau_{i,j}$.



M1 computation occurring on processor M_1
suspension
M1 suspension converted into computation that occurs on processor M_1
τ_1
τ_2
τ_3
M1
M2
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 t_d=30 32 t_f=34 Time

(a) Original schedule $S^0 = S$

(b) Schedule after performing the switch operation in the first transformation step

(c) Schedule after performing the move operation in the first transformation step

(d) Schedule S^1 after applying the transformation method w.r.t. M_1 (completing the first transformation step)

(e) Final transformed schedule $S^2 = \overline{S}$ after applying the transformation method w.r.t. both M_1 and M_2

Figure 3.37: Example schedule transformation.

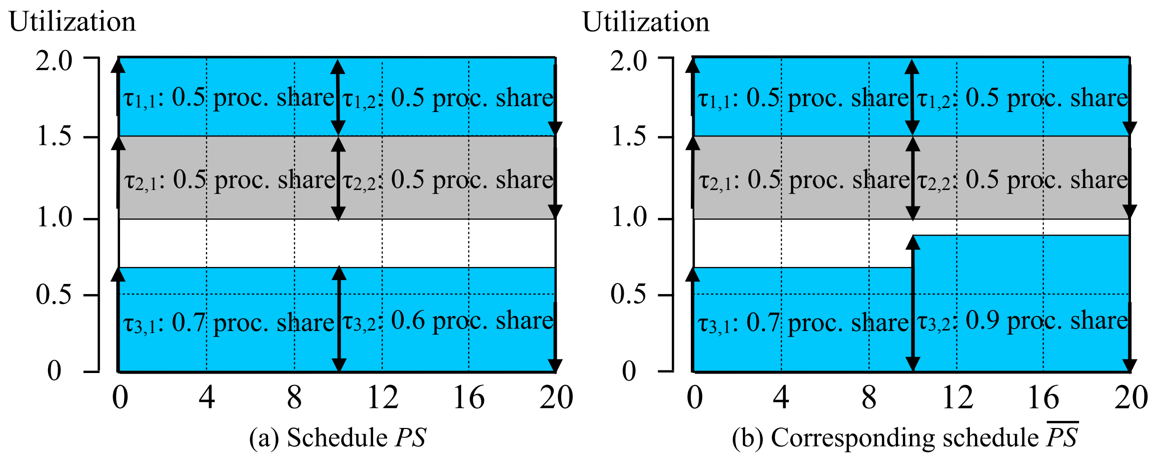


Figure 3.38: An example task system containing three tasks, τ_1 and τ_2 of utilization 0.5, and τ_3 of utilization 0.6. All three tasks have a period of 10 time units. (a) shows the PS schedule PS for this system where each task executes according to its utilization rate when it is active. Assume that in the transformed schedule $\overline{S}$ for this system, three time units of suspensions of $\tau_{3,2}$ are turned into computation, which causes the utilization of τ_3 to increase to 0.9 in $\tau_{3,2}$'s job execution window $[r_{3,2}, d_{3,2}) = [10, 20)$. (b) shows the corresponding schedule $\overline{PS}$ after this transformation. As seen, in $\overline{PS}$, task τ_3 executes with a rate of 0.9 in $[10, 20)$.

Defining $\overline{PS}$. Now we define the PS schedule $\overline{PS}$ corresponding to the transformed schedule $\overline{S}$. Without applying the transformation method, any task τ_i executes in a PS schedule with the rate u_i in any of its job execution windows (i.e., when τ_i is active). Thus, if τ_i is active throughout $[t_1, t_2)$ in PS, then $A(\tau_{i,j}, t_1, t_2, PS) = (t_2 - t_1)u_i$ holds.

On the other hand, after applying the transformation method, since we may convert a certain portion of the suspension time of any job of any task into computation, a task may have different utilizations within different job execution windows, but within the range of $[u_i, \overline{u_i}]$. The upper bound of this range is $\overline{u_i}$ because all suspensions of a job could be turned into computation. Thus, for any task τ_i that is active throughout $[t_1, t_2)$ in $\overline{PS}$, we have

$$(t_2 - t_1) \cdot u_i \le A(\tau_{i,j}, t_1, t_2, \overline{PS}) \le (t_2 - t_1) \cdot \overline{u_i}. \tag{3.23}$$

An example is given in Figure 3.38.

Note that our analysis does not require $\overline{PS}$ to be specifically defined; rather, only (3.23) is needed in our analysis.

Definition 3.16. Let $v_i = \dfrac{s_i}{p_i}$ denote the *suspension ratio* for task $\tau_i \in \tau$. Let v^j denote the j^{th} maximum suspension ratio among tasks in τ.

In order for our analysis to be correct, we have to ensure that such a valid $\overline{PS}$ schedule exists. That is, we have to guarantee that the total utilization of τ is at most m at any time instant t in $\overline{PS}$. The following lemma gives a sufficient condition that can provide such a guarantee.

Lemma 3.5. *If $U_{sum} + \sum_{j=1}^{m} v^j \leq m$, then a valid PS schedule $\overline{PS}$ corresponding to $\overline{S}$ exists.*

Proof. By the transformation method, for any processor M_k, jobs whose suspensions are converted into computation with respect to M_k do not have overlapping job execution windows. This is because such jobs either belong to the same task (in which case such jobs clearly do not have overlapping job execution windows), or belong to different tasks each of which is associated with a different transformation interval with respect to M_k. In the latter case, non-overlap follows by Properties (G1) and (G2) stated earlier. Since there are m processors and the jobs used for the transformation with respect to each processor do not have overlapping job execution windows, at any time t in $\overline{PS}$, there exist at most m jobs with overlapping job execution windows whose suspensions are converted into computation, which can increase total utilization by at most $\sum_{j=1}^{m} v^j$. This implies that at any time t in $\overline{PS}$, the total utilization of τ is at most $U_{sum} + \sum_{j=1}^{m} v^j$, which is at most m according to the claim statement. $\square$

We next derive a lower bound (Section 3.4.3) and an upper bound (Section 3.4.4) on the pending work that can compete with $\tau_{l,j}$ after t_d in schedule $\overline{S}$, which is given by $LAG(\mathbf{d}, t_d, \overline{S})$, as **d** includes all jobs of higher priority than $\tau_{l,j}$.

3.4.3 Lower Bound

Lemma 3.6 below establishes the desired lower bound on $LAG(\mathbf{d}, t_d, \overline{S})$.

Lemma 3.6. *If the tardiness of $\tau_{l,j}$ exceeds $x + e_l + s_l$, then $LAG(\mathbf{d}, t_d, \overline{S}) > m \cdot x + e_l + s_l$.*

Proof. We prove the contrapositive: we assume that

$$LAG(\mathbf{d}, t_d, \overline{S}) \quad \leq \quad m \cdot x + e_l + s_l \tag{3.24}$$

holds and show that the tardiness of $\tau_{l,j}$ cannot exceed $x + e_l + s_l$. Let η_l be the amount of work $\tau_{l,j}$ performs by time t_d in $\overline{S}$. Note that by the transformation method, $0 \le \eta_l < e_l + s_l$. Define y as follows.

$$y = x + \frac{\eta_l}{m} \tag{3.25}$$

We consider two cases.

Case 1. $[t_d, t_d + y)$ *is a busy interval for* **d**. In this case, the amount of work completed in $[t_d, t_d + y)$ is exactly my. Hence, the amount of work pending at $t_d + y$ is at most $LAG(\mathbf{d}, t_d, \overline{S}) - my \overset{\{\text{by (3.24) and (3.25)}\}}{\le} mx + e_l + s_l - mx - \eta_l = e_l + s_l - \eta_l$. This remaining work will be completed no later than $t_d + y + e_l + s_l - \eta_l \overset{\{\text{by (3.25)}\}}{=} t_d + x + \frac{\eta_l}{m} + e_l + s_l - \eta_l \le t_d + x + e_l + s_l$. Since this remaining work includes the work due for $\tau_{l,j}$, $\tau_{l,j}$ thus completes by $t_d + x + e_l + s_l$.

Case 2. $[t_d, t_d + y)$ *is a non-busy interval for* **d**. Let t_s be the earliest non-busy instant in $[t_d, t_d + y)$. If more than $m - 1$ tasks have enabled tardy jobs in **d** at t_s, then since all such enabled tardy jobs in **d** at t_s have deadlines at or before $t_d \le t_s$, by Claim 6, t_s cannot be non-busy. Thus, at most $m - 1$ tasks can have enabled tardy jobs in **d** at t_s. Moreover, since the number of tasks that have enabled jobs in **d** does not increase after t_d, we have

(Z) At most $m - 1$ tasks have enabled tardy jobs in **d** at or after t_s.

If $\tau_{l,j}$ completes by t_s, then $f_{l,j} \le t_s < t_d + y \overset{\{\text{by (3.25)}\}}{=} t_d + x + \frac{\eta_l}{m} < t_d + x + e_l + s_l$. In the rest of the proof, assume that $\tau_{l,j}$ completes after t_s. Let t_p be the completion time of $\tau_{l,j}$'s predecessor (i.e., $\tau_{l,j-1}$). If $t_p \le t_s$, then $\tau_{l,j}$ is enabled at t_s and will execute or suspend at t_s because t_s is non-busy. Furthermore, by (Z), $\tau_{l,j}$ is not preempted after t_s. Thus, by the definition of η_l and t_s, we have $f_{l,j} \le t_s + e_l + s_l - \eta_l < t_d + y + e_l + s_l - \eta_l \overset{\{\text{by (3.25)}\}}{=} t_d + x + \frac{\eta_l}{m} + e_l + s_l - \eta_l \le t_d + x + e_l + s_l$.

The remaining possiblity is that $t_p > t_s$. In this case, $\tau_{l,j}$ will begin its first phase at t_p and by (Z) finish by time $t_p + e_l + s_l$. By Property (P1) (applied to $\tau_{l,j}$'s predecessor), $t_p \le t_d - p_l + x + e_l + s_l \le t_d + x$. Thus, the tardiness of $\tau_{l,j}$ is $f_{l,j} - t_d \le t_p + e_l + s_l - t_d \le x + e_l + s_l$. $\square$

3.4.4 Upper Bound

In this section, we determine an upper bound on $LAG(\mathbf{d}, t_d, \overline{S})$.

Definition 3.17. Let t_n be the end of the latest non-busy interval for **d** before t_d, if any; otherwise, $t_n = 0$.

By the above definition and Claim 5, we have

$$LAG(\mathbf{d}, t_d, \overline{S}) \leq LAG(\mathbf{d}, t_n, \overline{S}). \tag{3.26}$$

Lemma 3.7. *For any task τ_i and $t \in [0, t_d]$, if τ_i has pending jobs at t in the schedule $\overline{S}$, then we have*

$$lag(\tau_i, t, \overline{S}) \leq \begin{cases} e_i + s_i & \textit{if } d_{i,k} \geq t \\ \overline{u_i} \cdot x + e_i + s_i + \overline{u_i} \cdot s_i & \textit{if } d_{i,k} < t \end{cases}$$

where $d_{i,k}$ is the deadline of the earliest pending job of τ_i, $\tau_{i,k}$, at time t in $\overline{S}$. If such a job does not exist, then $lag(\tau_i, t, \overline{S}) \leq 0$.

Proof. If τ_i does not have a pending job at t in $\overline{S}$, then by Definition 2.3 and (2.4), $lag(\tau_i, t, \overline{S}) \leq 0$ holds. So assume such a job exists. Let γ_i be the amount of work $\tau_{i,k}$ performs before t. By the transformation method, $\gamma_i < e_i + s_i$ holds.

By the selection of $\tau_{i,k}$, we have $lag(\tau_i, t, \overline{S}) = \sum_{h \geq k} lag(\tau_{i,h}, t, \overline{S}) = \sum_{h \geq k} \big(A(\tau_{i,h}, 0, t, PS) - A(\tau_{i,h}, 0, t, \overline{S})\big)$. Given that no job executes before its release time, $A(\tau_{i,h}, 0, t, \overline{S}) = A(\tau_{i,h}, r_{i,h}, t, \overline{S})$. Thus,

$$\begin{aligned} lag(\tau_i, t, \overline{S}) \quad = \quad & A(\tau_{i,k}, r_{i,k}, t, \overline{PS}) - A(\tau_{i,k}, r_{i,k}, t, \overline{S}) \\ & + \sum_{h > k} \big(A(\tau_{i,h}, r_{i,h}, t, \overline{PS}) \\ & - A(\tau_{i,h}, r_{i,h}, t, \overline{S})\big). \end{aligned} \tag{3.27}$$

By the definition of $\overline{PS}$, (3.23), and Definition 3.14, $A(\tau_{i,k}, r_{i,k}, t, \overline{PS}) \leq e_i + s_i$, and $\sum_{h>k} A(\tau_{i,h}, r_{i,h}, t, \overline{PS}) \leq \overline{u_i} \cdot \max(0, t - d_{i,k})$. By the selection of $\tau_{i,k}$, $A(\tau_{i,k}, r_{i,k}, t, \overline{S}) = \gamma_i$, and $\sum_{h>k} A(\tau_{i,h}, r_{i,h}, t, \overline{S}) = 0$. By setting these values into (7.3), we have

$$lag(\tau_i, t, \overline{S}) \leq e_i + s_i - \gamma_i + \overline{u_i} \cdot \max(0, t - d_{i,k}). \tag{3.28}$$

There are two cases to consider.

Case 1. $d_{i,k} \geq t$. In this case, (3.28) implies $lag(\tau_i, t, \overline{S}) \leq e_i + s_i - \gamma_i \leq e_i + s_i$.

Case 2. $d_{i,k} < t$. In this case, because $t \leq t_d$ and $d_{l,j} = t_d$, $\tau_{i,k}$ is not the job $\tau_{l,j}$. Thus, by Property (P1), $\tau_{i,k}$ has a tardiness of at most $x + e_i + s_i$. Since $\tau_{i,k}$ is the earliest pending job of τ_i at time t, the earliest possible completion time of $\tau_{i,k}$ is at $t + e_i - \gamma_i$ ($\tau_{i,k}$ may suspend for zero time at run-time). Thus, we have $t + e_i - \gamma_i \leq d_{i,k} + x + e_i + s_i$, which gives $t - d_{i,k} \leq x + \gamma_i + s_i$. Setting this value into (3.28), we have $lag(\tau_i, t, \overline{S}) \leq e_i + s_i - \gamma_i + \overline{u_i} \cdot (x + \gamma_i + s_i) \leq \overline{u_i} \cdot x + e_i + s_i + \overline{u_i} \cdot s_i$. □

Definition 3.18. Let $\overline{U}_{m-1}$ be the sum of the $m-1$ largest $\overline{u_i}$ values among tasks in τ. Let $\overline{E}$ be the largest value of the expression $\sum_{\tau_i \in \tau}(e_i + s_i) + \sum_{\tau_i \in \psi} \overline{u_i} \cdot s_i$, where ψ denotes any set of $m-1$ tasks in τ.

Lemma 3.8 below upper bounds $LAG(\mathbf{d}, t_d, \overline{S})$.

Lemma 3.8. *$LAG(\boldsymbol{d}, t_d, \overline{S}) \leq \overline{U}_{m-1} \cdot x + \overline{E}$.*

Proof. By (3.26), we have $LAG(\mathbf{d}, t_d, \overline{S}) \leq LAG(\mathbf{d}, t_n, \overline{S})$. By summing individual task lags at t_n, we can bound $LAG(\mathbf{d}, t_n, \overline{S})$. If $t_n = 0$, then $LAG(\mathbf{d}, t_n, \overline{S}) = 0$, so assume $t_n > 0$.

Given that the instant $t_n - 1$ is non-busy, by Claim 6, at most $m-1$ tasks can have enabled tardy jobs at $t_n - 1$ with deadlines at or before $t_n - 1$. Let θ denote the set of such tasks. Therefore, we have

$$\begin{aligned}
& LAG(\mathbf{d}, t_d, \overline{S}) \\
& \quad \{\text{by } (3.26)\} \\
\leq & LAG(\mathbf{d}, t_n, \overline{S}) \\
& \quad \{\text{by } (2.4)\} \\
= & \sum_{\tau_i : \tau_{i,v} \in \mathbf{d}} lag(\tau_i, t_n, \overline{S}) \\
& \quad \{\text{by Lemma 3.7}\} \\
\leq & \sum_{\tau_i \in \theta} (\overline{u_i} \cdot x + e_i + s_i + \overline{u_i} \cdot s_i) + \sum_{\tau_j \in \tau - \theta} (e_j + s_j)
\end{aligned}$$

$\{$by Definition 3.18$\}$

$\leq \overline{U}_{m-1} \cdot x + \overline{E}.$ □

3.4.5 Determining x

Setting the upper bound on $LAG(\mathbf{d}, t_d, \overline{S})$ in Lemma 3.8 to be at most the lower bound in Lemma 3.6 will ensure that the tardiness of $\tau_{l,j}$ is at most $x + e_l + s_l$. The resulting inequality can be used to determine a value for x. By Lemmas 3.6 and 3.8, this inequality is $m \cdot x + e_l + s_l \geq \overline{U}_{m-1} \cdot x + \overline{E}$. Solving for x, we have

$$x \geq \frac{\overline{E} - e_l - s_l}{m - \overline{U}_{m-1}}. \tag{3.29}$$

By Definitions 3.14 and 3.18, $\overline{U}_{m-1} < m$ clearly holds. Thus, if x equals the right-hand side of (3.29), then the tardiness of $\tau_{l,j}$ will not exceed $x + e_l + s_l$ in $\overline{S}$. A value for x that is independent of the parameters of τ_l can be obtained by replacing $-e_l - s_l$ with $max_l(-e_l - s_l)$. Moreover, in order for our analysis to be valid, the condition $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ as stated in Lemma 3.5 must hold.

Since the transformation method used to obtain $\overline{S}$ does not alter the tardiness of any job, the claim below follows.

Claim 7. The tardiness of $\tau_{l,j}$ in the original schedule S is the same as that in the transformed schedule $\overline{S}$.

By the above discussion, the theorem below follows.

Theorem 3.3. *With x as defined in (3.29), the tardiness of any task τ_l scheduled under GEDF is at most $x + e_l + s_l$, provided $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ where v^j is defined in Definition 3.16.*

3.4.6 Theoretical Dominance over Prior Tests

We now show that our derived schedulability test theoretically dominates the suspension-oblivious approach [86] and the suspension-aware analysis presented in Section 3.2 with respect to schedulability. Moreover, we show via a counterexample that task systems that violate the utilization constraint stated in Theorem 3.3 may have unbounded tardiness.

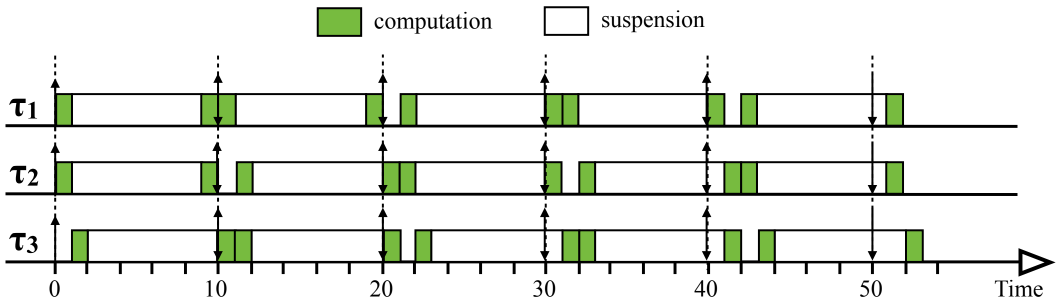


Figure 3.39: GEDF schedule of the counterexample.

When using the suspension-oblivious approach of treating all suspensions as computation, which transforms all sporadic self-suspending tasks into ordinary sporadic tasks, and then applying prior SRT schedulability analysis [41], the resulting utilization constraint is given by $U_{sum}+\sum_{i=1}^{n}\frac{s_i}{p_i} \leq m$. Clearly the constraint $U_{sum}+\sum_{i=1}^{m} v^j \leq m$ from Theorem 3.3 is less restrictive than this prior approach. Moreover, the resulting utilization constraint when using the technique presented in Section 3.2 is given by $U_{sum} < \left(1 - max_i\left(\frac{s_i}{e_i+s_i}\right)\right) \cdot m$. Since $\sum_{i=1}^{m} v^j \leq max_i\left(\frac{s_i}{e_i+s_i}\right) \cdot m$, our schedulability test is also less restrictive than this prior approach. Note that a major reason the prior approach in Section 3.2 causes significant capacity loss for some task systems is the fact that the term $max_i\left(\frac{s_i}{e_i+s_i}\right) \cdot m$ is not associated with any task period parameter. As a result, many task systems with even small utilizations may be deemed to be unschedulable.

Counterexample. Consider a two-processor task set τ that consists of three identical self-suspending tasks: $\tau_1 = \tau_2 = \tau_3$ = (e1, s8, e1, 10). For this system, $U_{sum}+\sum_{i=1}^{m} v^j = 0.6+1.6 = 2.2 > m = 2$, which violates the condition stated in Theorem 3.3. Figure 3.39 shows the GEDF schedule of this task system. As seen, the tardiness of each task in this system grows with increasing job index.

3.4.7 Experiments

In this section, we describe experiments conducted using randomly-generated task sets to evaluate the applicability of Theorem 3.3. Our goal is to examine how restrictive the derived schedulability test's utilization cap is, and how large the magnitude of tardiness is, and to compare it with prior methods. In the following, we denote our $O(m)$ schedulability test, the test presented in Section 3.2, and the suspension-oblivious approach, as "O(m)," "LA," and "SC," respectively.

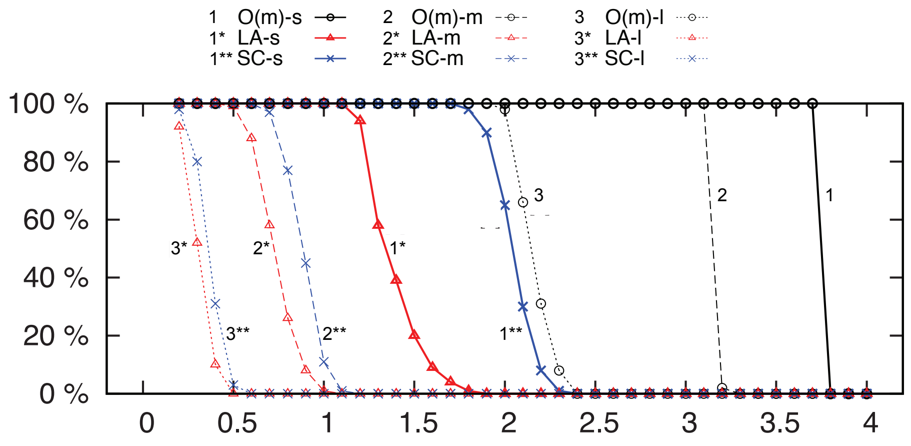


Figure 3.40: $m = 4$ and light per-task utilizations are assumed.

Experimental setup. In our experiments, task sets were generated as follows. Task periods were uniformly distributed over [50*ms*,200*ms*]. Task utilizations were distributed differently for each experiment using three uniform distributions. The ranges for the uniform distributions were [0.005,0.1] (light), [0.1,0.3] (medium), and [0.3,0.8] (heavy). Task execution costs were calculated from periods and utilizations. Suspension lengths of tasks were also distributed using three uniform distributions: $[0.005 \cdot (1 - u_i) \cdot p_i, 0.1 \cdot (1 - u_i) \cdot p_i]$ (suspensions are short), $[0.1 \cdot (1 - u_i) \cdot p_i, 0.3 \cdot (1 - u_i) \cdot p_i]$ (suspensions are moderate), and $[0.3 \cdot (1 - u_i) \cdot p_i, 0.8 \cdot (1 - u_i) \cdot p_i]$ (suspensions are long).[7] We varied the total system utilization U_{sum} within $\{0.1, 0.2, ..., m\}$. For each combination of task utilization distribution, suspension length distribution, and U_{sum}, 1,000 task sets were generated for systems with four and eight processors. Each such task set was generated by creating tasks until total utilization exceeded the corresponding utilization cap, and by then reducing the last task's utilization so that the total utilization equalled the utilization cap. For each generated system, SRT schedulability (i.e., the ability to ensure bounded tardiness) was checked for O(m), LA, and SC.

Results. The obtained SRT schedulability results for GEDF are shown in Figures 3.40-3.45. In these figures, labels "O(m)-s", "O(m)-m", "O(m)-l" ("LA-s", "LA-m", "LA-l" and "SC-s", "SC-m", "SC-l", respectively) represent the approach of O(m) (LA and SC, respectively) assuming short, moderate, and long suspensions, respectively. In all figures, the x-axis denotes the task set utilization cap and the y-axis denotes the fraction of generated task sets that were schedulable with bounded deadline

[7]Note that any s_i is upper-bounded by $(1 - u_i) \cdot p_i$.

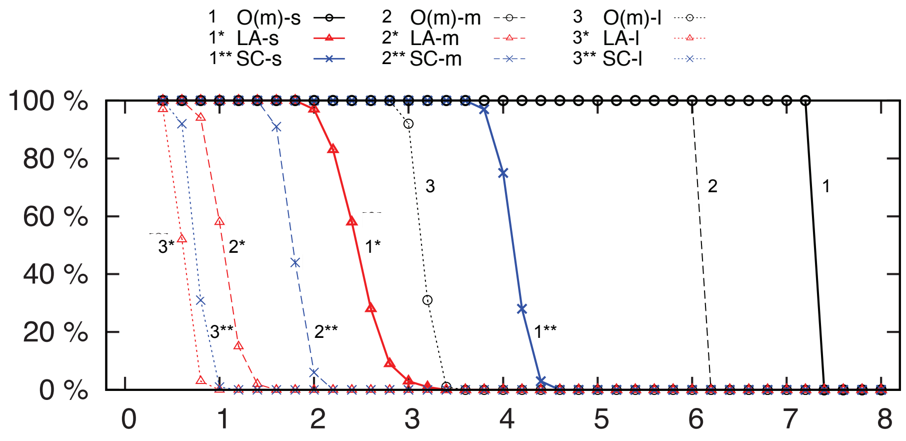


Figure 3.41: $m = 8$ and light per-task utilizations are assumed.

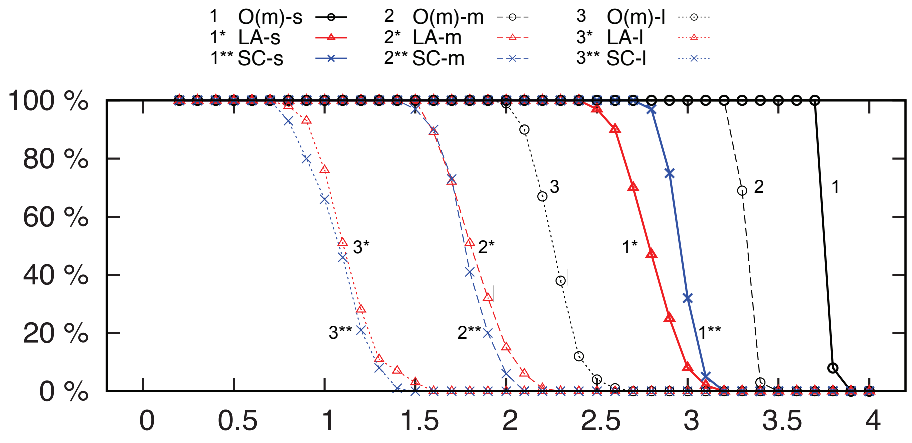


Figure 3.42: $m = 4$ and medium per-task utilizations are assumed.

tardiness. Each figure gives three curves per tested approach for the cases of short, moderate, and long suspensions, respectively. Each curve plots the fraction of the generated task sets the corresponding approach successfully scheduled, as a function of total utilization. As seen, in all tested scenarios, O(m) significantly improves upon LA and SC by a substantial margin. For example, as seen in Figure 3.40, when task utilizations are light and $m = 4$, O(m) can achieve 100% schedulability when U_{sum} equals 3.7, 3.1, and 2.1 when suspension lengths are short, moderate, and long, respectively, while LA and SC fail to do so when U_{sum} merely exceeds 1.8, 0.7, and 0.3, respectively. Note that when task utilizations are lighter, the improvement margin by O(m) over LA and SC increases. This is because in this case, the $s_i/(e_i + s_i)$ term that cause capacity loss in LA becomes large since

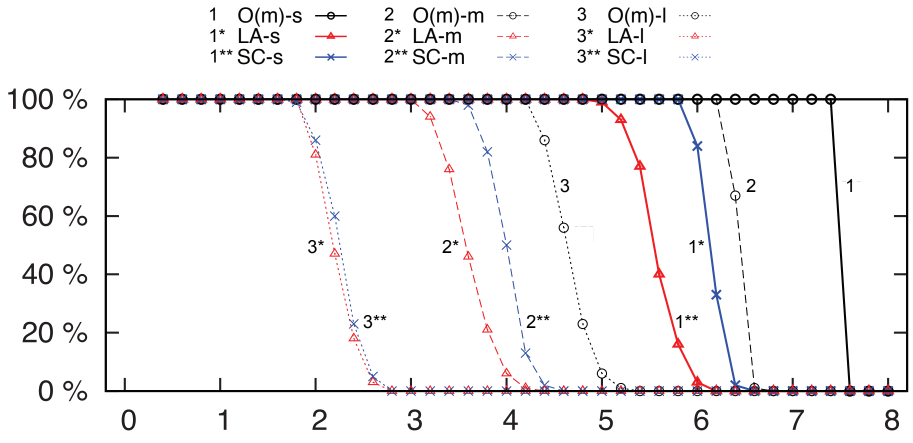


Figure 3.43: $m = 8$ and medium per-task utilizations are assumed.

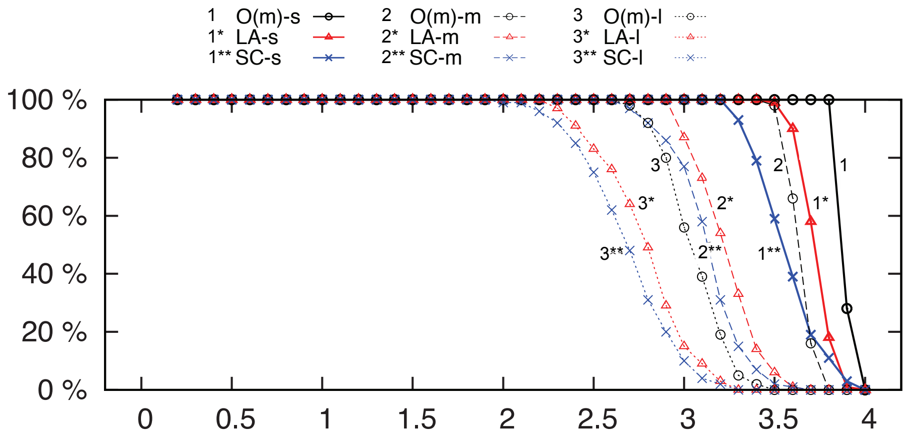


Figure 3.44: $m = 4$ and heavy per-task utilizations are assumed.

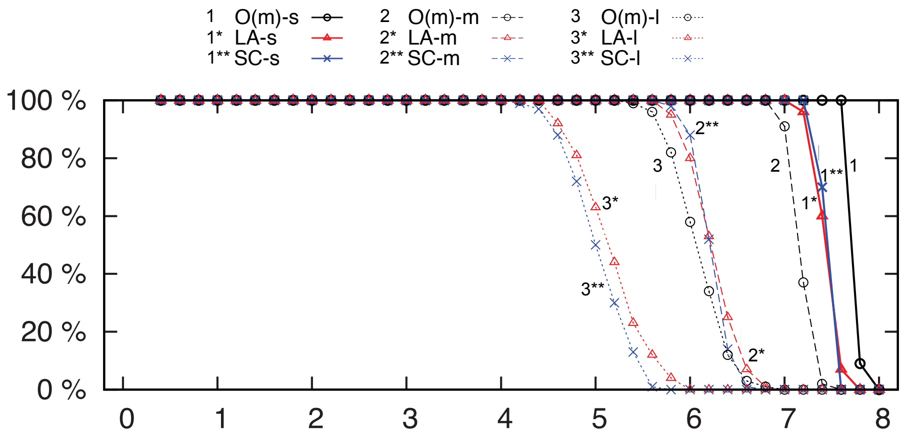


Figure 3.45: $m = 8$ and heavy per-task utilizations are assumed.

Table 3.3: Average tardiness bounds under O(m), SC, and LA when $m = 4$. The labels "**u-L**" /"**u-M**"/"**u-H**" indicate light/medium/ heavy task utilizations, respectively. Within the row "**O(m)**," the three sub-rows "**n-O(m)**" /"**n-SC**"/"**n-LA**" represent the average tardiness bound achieved by O(m) as computed for all task sets that can be successfully scheduled under O(m)/SC/LA, respectively. The rows "**SC**"/"**LA**" represent the average tardiness bound achieved by SC/LA as computed for all task sets that can be successfully scheduled under SC/LA, respectively. All time units are in *ms*.

Method		Short susp.			Moderate susp.			Long susp.		
		u-L	u-M	U-H	u-L	u-M	U-H	u-L	u-M	U-H
O(m)	n-O(m)	140	125	191	301	173	286	667	286	377
	n-SC	59	83	178	82	113	247	181	167	289
	n-LA	22	34	158	50	64	197	106	172	203
SC		36	58	153	63	85	213	126	148	243
LA		97	144	316	149	287	375	231	626	892

e_i is small; similarly, for SC, more tasks are generated under light utilizations, which causes more capacity loss since all generated tasks' suspensions must be converted into computation. On the other hand, O(m)'s capacity loss is determined by the m largest suspension ratios, and thus is not similarly affected. In general, when suspension lengths are short and moderate, O(m) achieves little or no capacity loss in all scenarios. Even when suspension lengths become long, capacity loss under O(m) is still reasonable, especially when compared to the loss under LA and SC. Observe that all three approaches perform better when using heavier per-task utilization distributions. This is because when u_i is larger, s_i becomes smaller since $e_i + s_i \leq p_i$ must hold, which helps all three approaches yield smaller capacity loss.

In addition to schedulability, the magnitude of tardiness, as computed using the bound in Theorem 3.3, is of importance. Table 3.3 (the organization of which is explained in the table's caption) depicts the average of the computed bounds for each of the nine tested scenarios (three utilization distributions combined with three suspension length distributions) when $m = 4$ under O(m), SC, and LA, respectively. In all tested scenarios, O(m) provides reasonable predicted tardiness, which is comparable to SC and improves upon LA. For cases where suspensions are short, the predicted tardiness under O(m) is low. Moreover, as Figures 3.40-3.45 imply, O(m) can often ensure bounded tardiness when SC or LA cannot. To conclude, our proposed analysis technique not only often guarantees schedulability with little or no capacity loss, but can provide such a guarantee with reasonable predicted tardiness.

3.5 Chapter Summary

In this chapter, we presented multiprocessor schedulability tests for SRT sporadic self-suspending task systems. For the past 20 years, the unsolved problem of supporting real-time systems with suspensions has impeded research progress on many related research topics such as analyzing and implementing I/O-intensive applications in multiprocessor systems. The impact of my work is demonstrated by the fact that it provides a first set of practically efficient solutions that can fundamentally solve this problem.

CHAPTER 4

Scheduling HRT Self-Suspending Tasks[1]

In the previous chapter, we presented multiprocessor schedulability tests for SRT self-suspending task systems. In this chapter, we consider the HRT case. It has been shown that precisely analyzing HRT systems with suspensions is difficult, even for very restricted self-suspending task models on uniprocessors [106]. However, uniprocessor analysis that is correct in a sufficiency sense (although pessimistic) has been proposed (see Section 2.3.2). Such analysis can be applied on a per-processor basis to deal with suspensions under partitioning approaches where tasks are statically bound to processors. In contrast, for global scheduling where tasks are scheduled from a single run queue and may migrate across processors, other than the suspension-oblivious approach, which simply integrates suspensions into per-task WCET requirements, no known global HRT schedulability analysis exists for self-suspending task systems. In this chapter, we present the first suspension-aware analysis for globally scheduled HRT self-suspending tasks for multiprocessor systems. We focus specifically on two widely used global schedulers: GTFP and GEDF scheduling. We analyze these schedulers assuming the scheduled workload is an HRT arbitrary-deadline sporadic self-suspending task system.

We consider the sporadic self-suspending task model described in Section 2.1.4, except that we consider the HRT case (defined in Section 1.2.2) in this chapter. Specifically, We establish HRT schedulability tests for arbitrary-deadline sporadic self-suspending task systems under both GTFP (Section 4.1) and GEDF (Section 4.2).

To enable more general results, for any task τ_i, we allow a predefined tardiness threshold to be set, denoted λ_i. (Task τ_i is an ordinary HRT task if $\lambda_i = 0$.) Throughout this chapter, we assume that

[1]Contents of this chapter previously appeared in preliminary form in the following papers:
Cong Liu and James Anderson. Suspension-Aware Analysis for Hard Real-Time Multiprocessor Scheduling, Proceedings of the 25th EuroMicro Conference on Real-Time Systems, to appear, 2013.

e_i, s_i, d_i, p_i, and λ_i for any task $\tau_i \in \tau$ are non-negative integers, and as before, all time values are integral. Thus, a job that executes at time point t executes during the entire time interval $[t, t+1)$.

For simplicity, we henceforth assume that each job of any task τ_i executes for *exactly* e_i time units. By Claim 1, any response-time bound derived for a sporadic self-suspending task system by considering only schedules meeting this assumption applies to other schedules as well. For any job $\tau_{i,k}$, we let $s_{i,k}$ denote its total suspension time, where $s_{i,k} \leq s_i$.

Real-time workloads often have both self-suspending tasks and computational tasks (which do not suspend) co-exist. To reflect this, we let τ^s (τ^e) denote the set of self-suspending (computational) tasks in τ, as we did in Chapter 3. Also, we let n_s (n_e) denote the number of self-suspending (computational) tasks in τ.

Next, in Sections 4.1 and 4.2, we present the aforementioned suspension-aware GTFP and GEDF schedulability tests, respectively. In Section 4.2.4, we experimentally compare them with other methods.

4.1 GTFP

In this section, based upon RTA, we derive a fixed-priority multiprocessor schedulability test for HRT and SRT (i.e., each task τ_i can have a predefined tardiness threshold λ_i) arbitrary-deadline sporadic self-suspending task systems.

Under GTFP, a task cannot be interfered with by tasks with lower priorities. Assume that tasks are ordered by decreasing priority, i.e., $i < k$ iff τ_i has a higher priority than τ_k.

Definition 4.1. Let $\tau_{l,j}$ be the *maximal* job of τ_l, i.e., $\tau_{l,j}$ either has the largest response time among all jobs of τ_l or it is the first job of τ_l that has a response time exceeding $d_l + \lambda_l$.

We assume $l > m$ since under GTFP, any task τ_i where $i \leq m$ has a response time bound of $e_i + s_i$. We further assume that for any task τ_i where $i < l$, its largest response time does not exceed $d_i + \lambda_i$. Our analysis focuses on the job $\tau_{l,j}$, as defined above. To avoid distracting "boundary cases," we also assume that the schedule being analyzed is prepended with a schedule in which no deadlines are missed that is long enough to ensure that all predecessor jobs referenced in the proof exist (this applies to Section 4.2 as well).

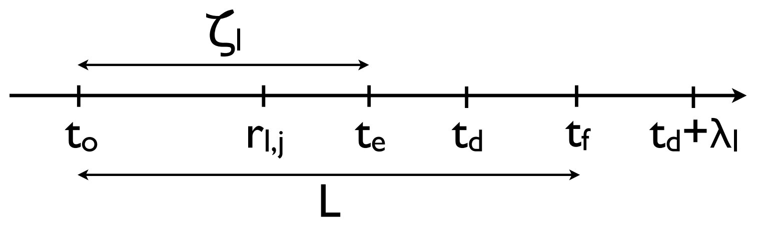


Figure 4.1: The maximal job $\tau_{l,j}$ of task τ_l becomes eligible at t_e. t_o is the earliest time instant before t_e such that at any time instant $t \in [t_o, t_e)$ all processors are occupied by tasks with equal or higher priority than $\tau_{l,j}$.

Definition 4.2. Let $f_{i,k}$ denote the completion time of job $\tau_{i,k}$. The *eligible time* of job $\tau_{i,k}$ is defined to be $max(r_{i,k}, f_{i,k-1})$. Let t_f denote the completion time of our job of interest $\tau_{l,j}$, t_e denote its eligible time (i.e., $t_e = max(r_{l,j}, f_{l,j-1})$), and t_d denote its deadline.

As in [10], we extend the analyzed interval from t_e to an earlier time instant t_o as defined below.

Definition 4.3. t_o denotes the earliest time instant at or before t_e such that at any time instant $t \in [t_o, t_e)$ all processors are occupied by tasks with equal[2] or higher priority than τ_l, as illustrated in Figure 4.1.

For conciseness, let $\tau_{hp} \subseteq \tau$ denote the set of tasks that have equal or higher priority than the analyzed task τ_l, and let

$$L = t_f - t_o \tag{4.1}$$

and

$$\zeta_l = t_e - t_o. \tag{4.2}$$

Definition 4.4. A task τ_i has a *carry-in* job if there is a job of τ_i that is released before t_o that has not completed by t_o.

Two parameters are important to RTA: the *workload* and the *interference*, as defined below.

Workload. The workload of an sporadic self-suspending task τ_i in the interval $[t_o, t_f)$ is the amount of computation that τ_i requires to execute in $[t_o, t_f)$. Note that suspensions do not contribute to the workload since they do not occupy any processor. Let $\omega(\tau_i, L)$ denote an upper bound of the

[2] Note that any job $\tau_{l,k}$ of τ_l where $k < j$ may delay $\tau_{l,j}$ from executing and thus can be considered to have higher priority than $\tau_{l,j}$.

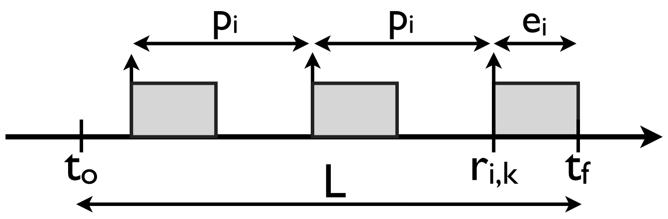


Figure 4.2: Computing $\omega^{nc}(\tau_i, L)$.

workload of each task $\tau_i \in \tau_{hp}$ in the interval $[t_o, t_f)$ of length L. Let $\omega^{nc}(\tau_i, L)$ denote the workload bound if τ_i does not have a carry-in job (see Definition 4.4), and let $\omega^{c}(\tau_i, L)$ denote the workload bound if τ_i has a carry-in job. $\omega^{nc}(\tau_i, L)$ and $\omega^{c}(\tau_i, L)$ can be computed as shown in the following lemmas.

Lemma 4.1.

$$\omega^{nc}(\tau_i, L) = \left(\left\lfloor \frac{L - e_i}{p_i} \right\rfloor + 1 \right) \cdot e_i. \tag{4.3}$$

Proof. Since τ_i does not have a carry-in job, only jobs that are released within $[t_o, t_f)$ can contribute to $\omega^{nc}(\tau_i, L)$. The scenario for the worst-case workload to happen is shown in Figure 4.2, where job $\tau_{i,k}$, which is the last job of τ_i that is released before t_f, executes continuously within $[r_{i,k}, r_{i,k} + e_i)$ such that $r_{i,k} + e_i = t_f$ (according to our task model, each suspension of $\tau_{i,k}$ within $[r_{i,k}, t_f)$ may be of length 0), and jobs of τ_i are released periodically. (Note that if $i = l$, then this worst-case scenario still gives a safe upper bound on the workload since in this case $\tau_{l,j}$ could be the job $\tau_{i,k}$.) Besides $\tau_{i,k}$, there are at most $\left\lfloor \frac{L - e_i}{p_i} \right\rfloor$ jobs of τ_i released within $[t_o, t_f)$. $\square$

Definition 4.5. For any interval of length t, let $\Delta(\tau_i, t) = \left(\left\lceil \frac{t}{p_i} \right\rceil - 1 \right) \cdot e_i + min \left(e_i, t - \left\lceil \frac{t}{p_i} \right\rceil \cdot p_i + p_i \right)$.

$\Delta(\tau_i, t)$ is defined for computing carry-in workloads. Definition 4.5 improves upon a similar definition for computing carry-in workloads proposed in [77] by deriving a more precise upper bound of the workload.

The following lemma, which computes $\omega^{c}(\tau_i, L)$, is proved similarly to Lemma 4.1.

Lemma 4.2.

$$\omega^{c}(\tau_i, L) \quad = \Delta(\tau_i, L - e_i + d_i + \lambda_i) \tag{4.4}$$

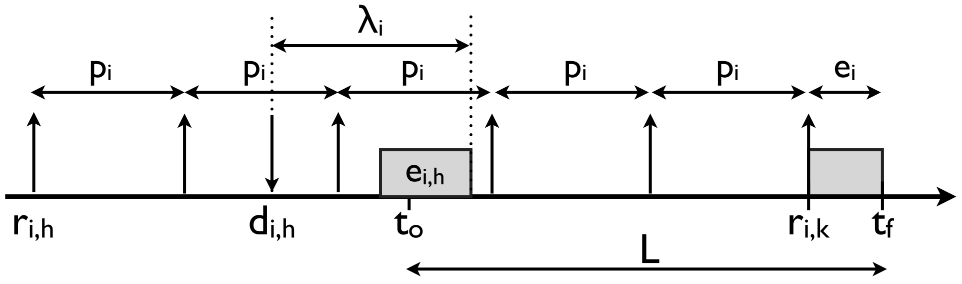


Figure 4.3: Computing $\omega^c(\tau_i, L)$.

Proof. The scenario for the worst-case workload to happen is shown in Figure 4.3, where job $\tau_{i,k}$, which is the last job of τ_i that is released before t_f, executes continuously within $[r_{i,k}, r_{i,k} + e_i)$ such that

$$r_{i,k} + e_i = t_f \tag{4.5}$$

(recall that $\tau_{i,k}$ may suspend for zero time within $[r_{i,k}, t_f)$), and jobs are released periodically. (Note that if $i = l$, then this worst-case scenario still gives a safe upper bound on the workload since $\tau_{l,j}$ could be the job $\tau_{i,k}$.)

Let $\tau_{i,h}$ be the first job such that $r_{i,h} < t_o$ and

$$d_{i,h} + \lambda_i > t_o, \tag{4.6}$$

i.e., $\tau_{i,h}$ is the first job of τ_i (potentially tardy) that may execute during $[t_o, t_f)$ and is released before t_o (note that if $\tau_{i,h}$ does not exist, then τ_i would not have a carry-in job). Besides $\tau_{i,k}$ and $\tau_{i,h}$, jobs of τ_i that are released within $[r_{i,h}+p_i, r_{i,k})$ can contribute to $\omega^c(\tau_i, L)$. Let y denote the number of jobs of τ_i released in $[r_{i,h}, r_{i,k})$. There are thus $y - 1$ jobs of τ_i that are released within $[r_{i,h} + p_i, r_{i,k})$. Since jobs of τ_i are released periodically, we have

$$r_{i,k} - r_{i,h} = y \cdot p_i. \tag{4.7}$$

Moreover, work contributed by $\tau_{i,h}$ cannot exceed the smaller of e_i and the length of the interval $[t_o, d_{i,h} + \lambda_i)$. The length of the interval $[t_o, d_{i,h} + \lambda_i)$ is given by

$$\begin{aligned}
& d_{i,h} + \lambda_i - t_o \\
=& r_{i,h} + d_i + \lambda_i - t_o \\
& \{\text{by } (4.7)\} \\
=& r_{i,k} - y \cdot p_i + d_i + \lambda_i - t_o \\
& \{\text{by } (4.1) \text{ and } (4.5)\} \\
=& (L - e_i + d_i + \lambda_i) - y \cdot p_i.
\end{aligned}$$

Thus, the work contributed by $\tau_{i,h}$ is given by $min(e_i, (L - e_i + d_i + \lambda_i) - y \cdot p_i)$.

By summing the contributions of $\tau_{i,h}$, $\tau_{i,k}$, and jobs of τ_i that are released within $[r_{i,h} + p_i, r_{i,k})$, we have

$$\begin{aligned}
& \omega^c(\tau_i, L) \\
= & \ min(e_i, (L - e_i + d_i + \lambda_i) - y \cdot p_i) \\
& + e_i + (y - 1) \cdot e_i \\
= & \ min(e_i, (L - e_i + d_i + \lambda_i) - y \cdot p_i) \\
& + y \cdot e_i
\end{aligned} \tag{4.8}$$

To find y, by (4.7), we have

$$\begin{aligned}
& y \\
=& \frac{r_{i,k} - r_{i,h}}{p_i} \\
=& \frac{r_{i,k} - d_{i,h} + d_i}{p_i} \\
& \{\text{by } (4.6)\} \\
<& \frac{r_{i,k} - t_o + \lambda_i + d_i}{p_i} \\
& \{\text{by } (4.1) \text{ and } (4.5)\}
\end{aligned}$$

$$= \frac{L - e_i + \lambda_i + d_i}{p_i}.$$

For conciseness, let $\sigma = L - e_i + \lambda_i + d_i$. Thus, $y < \frac{\sigma}{p_i}$ holds. If $\sigma \bmod p_i = 0$, then $y \leq \frac{\sigma}{p_i} - 1 = \left\lceil \frac{\sigma}{p_i} \right\rceil - 1$, otherwise, $y \leq \left\lfloor \frac{\sigma}{p_i} \right\rfloor = \left\lceil \frac{\sigma}{p_i} \right\rceil - 1$. Thus, a general expression for y can be given by $y \leq \left\lceil \frac{\sigma}{p_i} \right\rceil - 1$.

By (4.8), the maximum value for $\omega^c(\tau_i, L)$ can be obtained when $y = \left\lceil \frac{\sigma}{p_i} \right\rceil - 1$. Setting this expression for y into (4.8), we get

$$\begin{aligned}
&\omega^c(\tau_i, L) \\
=& min\left(e_i, \sigma - \left\lceil \frac{\sigma}{p_i} \right\rceil \cdot p_i + p_i\right) + \left(\left\lceil \frac{\sigma}{p_i} \right\rceil - 1\right) \cdot e_i \\
&\quad \{\text{by Definition 4.5}\} \\
=& \Delta(\tau_i, \sigma) \\
&\quad \{\text{by the definition of } \sigma\} \\
=& \Delta(\tau_i, L - e_i + d_i + \lambda_i).
\end{aligned}$$
□

It is important to point out that neither $\omega^{nc}(\tau_i, L)$ nor $\omega^c(\tau_i, L)$ depends on ζ_l (as defined in (4.2)). For any given interval $[t_o, t_f)$ of length L, we get the same result of $\omega^{nc}(\tau_i, L)$ and $\omega^c(\tau_i, L)$, regardless of the value of ζ_l. This observation enables us to greatly reduce the time complexity to derive the response time bound, as shown later.

Interference. The interference $I_l(\tau_i, L)$ of a specific task τ_i on τ_l over $[t_o, t_f)$ is the part of the workload of τ_i that has higher priority than $\tau_{l,j}$ and can delay $\tau_{l,j}$ from executing its computation phases. Note that if $i \neq l$, then τ_i cannot interfere with τ_l while τ_i or τ_l is suspending. If $i = l$, then suspensions of job $\tau_{l,k}$ where $k < j$, may delay $\tau_{l,j}$ from executing. However, by Definition 4.3, all processors are occupied by tasks with equal or higher priority than τ_l at any time instant $t \in [t_o, t_e)$. Thus, whenever suspensions of any such job $\tau_{l,k}$ delay $\tau_{l,j}$ from executing within $[t_o, t_e)$, such suspensions must be overlapped with computation from some other task with higher priority than τ_l. Therefore, it suffices for us to compute the interference using workload as derived in (4.3) and (4.4).

(Intuitively, this portion of the schedule, i.e., the schedule within $[t_o, t_e)$, would be the same even if τ_l did not suspend, since τ_l has the lowest priority among the tasks being considered.)

As we did for the workload, we also define two expressions for $I_l(\tau_i, L)$. We use $I_l^{nc}(\tau_i, L)$ to denote a bound on the interference of τ_i to τ_l during $[t_o, t_f)$ if τ_i does not have a carry-in job, and use $I_l^c(\tau_i, L)$ if τ_i has a carry-in job.

By the definitions of workload and interference, within $[t_o, t_f)$, if $i \neq l$, then task τ_i cannot interfere with τ_l by more than τ_i's workload in this interval. Thus, we have $I_l^{nc}(\tau_i, L) \leq \omega^{nc}(\tau_i, L)$ and $I_l^c(\tau_i, L) \leq \omega^c(\tau_i, L)$. The other case is $i = l$. In this case, since $\tau_{l,j}$ cannot interfere with itself, we have $I_l^{nc}(\tau_i, L) \leq \omega^{nc}(\tau_l, L) - e_l$ and $I_l^c(\tau_i, L) \leq \omega^c(\tau_l, L) - e_l$. Moreover, because τ_i cannot interfere with τ_l while $\tau_{l,j}$ is executing and suspending for a total of $e_l + s_{l,j}$ time units in $[t_o, t_f)$, $I_l(\tau_i, L)$ cannot exceed $L - e_l - s_{l,j}$. Therefore, we have[3]

$$I_l^{nc}(\tau_i, L) = \begin{cases} min(\omega^{nc}(\tau_i, L), L - e_l - s_{l,j} + 1), & \text{if } i \neq l \\ min(\omega^{nc}(\tau_l, L) - e_l, L - e_l - s_{l,j} + 1), & \text{if } i = l \end{cases} \tag{4.9}$$

and

$$I_l^{c}(\tau_i, L) = \begin{cases} min(\omega^{c}(\tau_i, L), L - e_l - s_{l,j} + 1), & \text{if } i \neq l \\ min(\omega^{c}(\tau_l, L) - e_l, L - e_l - s_{l,j} + 1), & \text{if } i = l. \end{cases} \tag{4.10}$$

Now we define the *total interference bound* on τ_l within any interval $[t_o, t_o + Z)$ of arbitrary length Z, denoted $\Omega_l(Z)$, which is given by $\sum_{\tau_i \in \tau_{hp}} I_l(\tau_i, Z)$. The total interference bound on τ_l within the interval $[t_o, t_f)$ is thus given by $\Omega_l(L)$.

Upper-bounding $\Omega_l(L)$. By Definition 4.3, either $t_o = 0$, in which case no task has a carry-in job, or some processor is idle in $[t_o - 1, t_o)$, in which at most $m - 1$ computational tasks are active at $t_o - 1$. Thus, at most $min(m - 1, n_{hp}^e)$ computational tasks in τ_{hp} have carry-in jobs, where n_{hp}^e

[3] The upper bounds of $I_l^{nc}(\tau_i, L)$ and $I_l^c(\tau_i, L)$ (as shown next) are set to be $L - e_l - s_{l,j} + 1$ instead of $L - e_l - s_{l,j}$ in order to guarantee that the response time bound we get from the schedulability test presented later is valid. A formal explanation of this issue can be found in Section 4 of [19].

denotes the number of computational tasks in τ_{hp}. Due to suspensions, however, all self-suspending tasks in τ_{hp} may have carry-in jobs that suspend at t_o. Let τ^s_{hp} denote the set of self-suspending tasks in τ_{hp}. Thus, self-suspending tasks can contribute at most $\sum_{\tau_i \in \tau^s_{hp}} max(I^c_l(\tau_i, L), I^{nc}_l(\tau_i, L))$ work to $\Omega_l(L)$. Let τ^e_{hp} denote the set of computational tasks in τ_{hp} and $\beta^{min(m-1, n^e_{hp})}_{\tau_i \in \tau^e_{hp}}$ denote the $min(m-1, n^e_{hp})$ greatest values of $max(0, I^c_l(\tau_i, L) - I^{nc}_l(\tau_i, L))$ for any computational task $\tau_i \in \tau^e_{hp}$. Then computational tasks can contribute at most $\sum_{\tau_i \in \tau^e_{hp}} I^{nc}_l(\tau_i, L) + \beta^{min(m-1, n^e_{hp})}_{\tau_i \in \tau^e_{hp}}$ work to $\Omega_l(L)$. Therefore, by summing up the work contributed by both self-suspending tasks and computational tasks, we can bound $\Omega_l(L)$ by

$$\begin{aligned} \Omega_l(L) \quad = & \sum_{\tau_i \in \tau^s_{hp}} max(I^c_l(\tau_i, L), I^{nc}_l(\tau_i, L)) \\ & + \sum_{\tau_i \in \tau^e_{hp}} I^{nc}_l(\tau_i, L) + \beta^{min(m-1, n^e_{hp})}_{\tau_i \in \tau^e_{hp}}. \end{aligned} \quad (4.11)$$

The time complexity for computing $\sum_{\tau_i \in \tau^s_{hp}} max(I^c_l(\tau_i, L), I^{nc}_l(\tau_i, L))$ and $\sum_{\tau_i \in \tau^e_{hp}} I^{nc}_l(\tau_i, L)$ is $O(n)$. Also, as noted in [10], by using a linear-time selection technique from [24], the time complexity for computing $\beta^{min(m-1, n^e_{hp})}_{\tau_i \in \tau^e_{hp}}$ is $O(n)$. Thus, the time complexity to upper-bound $\Omega_l(L)$ as above is $O(n)$.

Schedulability test. We now derive an upper bound on the response time of task τ_l in an sporadic self-suspending task system τ scheduled using fixed priorities, as stated in Theorem 4.1. Before stating the theorem, we first present two lemmas, which are used to prove the theorem. Lemma 4.3 is intuitive since it states that the total interference of tasks with equal or higher priority than τ_l must be large enough to prevent $\tau_{l,j}$ from being finished at $t_o + H$ if $t_o + H < t_f$ holds (recall that t_f is defined to be the completion time of $\tau_{l,j}$).

Lemma 4.3. *For job $\tau_{l,j}$ and any interval $[t_o, t_o + H)$ of length H, if $H < t_f - t_o$, then*

$$\left\lfloor \frac{\Omega_l(H)}{m} \right\rfloor > H - e_l - s_{l,j}. \quad (4.12)$$

Proof. $\Omega_l(H)$ denotes the total interference bound on τ_l within the interval $[t_o, t_o + H)$.

If $t_e \geq t_o + H$, then by Definition 4.3, all processors must be occupied by tasks in τ_{hp} during the interval $[t_o, t_o + H)$, which implies that tasks in τ_{hp} generate a total workload of at least $m \cdot H$ within $[t_o, t_o + H)$ that can interfere with τ_l. Thus, (4.12) holds since $\Omega_l(H) \geq m \cdot H \geq m \cdot (H - e_l - s_{l,j} + 1)$.

The other possibility is $t_e < t_o + H$. In this case, given (from the statement of the lemma) that $H < t_f - t_o$, job $\tau_{l,j}$ is not yet completed at time $t_o + H$. Thus, only at strictly fewer than $e_l + s_{l,j}$ time points within the interval $[t_o, t_o + H)$ was $\tau_{l,j}$ able to execute its computation and suspension phases (for otherwise it would have completed by $t_o + H$). In order for $\tau_{l,j}$ to execute its computation and suspension phases for strictly fewer than $e_l + s_{l,j}$ time points within $[t_o, t_o + H)$, tasks in τ_{hp} must generate a total workload of at least $m \cdot (H - e_l - s_{l,j} + 1)$ within $[t_o, t_o + H)$ that can interfere with τ_l. Thus, $\Omega_l(H) \geq m \cdot (H - e_l - s_{l,j} + 1)$ holds. □

Lemma 4.4. *$t_e - r_{l,j} \leq \kappa_l$, where $\kappa_l = \lambda_l - p_l + d_l$ if $\lambda_l > p_l - d_l$, and $\kappa_l = 0$, otherwise.*

Proof. By Definition 4.1, we have

$$f_{l,j-1} \leq d_{l,j-1} + \lambda_l. \tag{4.13}$$

If $\lambda_l > p_l - d_l$, then we have

$$\begin{aligned}
& t_e - r_{l,j} \\
& \quad \{\text{by Definition 4.2}\} \\
=& max(r_{l,j}, f_{l,j-1}) - r_{l,j} \\
=& max(0, f_{l,j-1} - r_{l,j}) \\
& \quad \{\text{by (4.13)}\} \\
\leq& max(0, d_{l,j-1} + \lambda_l - r_{l,j}) \\
=& max(0, r_{l,j-1} + d_l + \lambda_l - r_{l,j}) \\
\leq& max(0, d_l + \lambda_l - p_l) \\
=& \lambda_l - p_l + d_l.
\end{aligned}$$

If $\lambda_l \le p_l - d_l$, then

$$\begin{aligned}
&f_{l,j-1}\\
&\{\text{by } (4.13)\}\\
\le &d_{l,j-1} + \lambda_l\\
= &r_{l,j-1} + d_l + \lambda_l\\
\le &r_{l,j} - p_l + d_l + \lambda_l\\
\le &r_{l,j},
\end{aligned}$$

which implies that job $\tau_{l,j-1}$ completes by $r_{l,j}$. Thus, by Definition 4.2, we have $t_e - r_{l,j} = 0$. □

Theorem 4.1. *Let ψ_l be the set of minimum solutions of (4.14) for L below for each value of $s_{l,j} \in \{0, 1, 2, ..., s_l\}$ by performing a fixed-point iteration on the RHS of (4.14) starting with $L = e_l + s_{l,j}$:*

$$L = \left\lfloor \frac{\Omega_l(L)}{m} \right\rfloor + e_l + s_{l,j}. \tag{4.14}$$

Then $\psi_l^{max} + \kappa_l$ upper-bounds τ_l's response time, where ψ_l^{max} is the maximum value in ψ_l.

Proof. We first prove by contradiction that $\psi_l^{max} + \kappa_l - \zeta_l$ is an upper bound of τ_l's response time. Assume that the actual worst-case response time of τ_l is given by R, where

$$R > \psi_l^{max} + \kappa_l - \zeta_l. \tag{4.15}$$

By Definitions 4.1 and 4.2, we have

$$R = t_f - r_{l,j}. \tag{4.16}$$

Thus, we have

$$\begin{aligned}
&\psi_l^{max}\\
&\{\text{by } (4.15)\}\\
< &R + \zeta_l - \kappa_l
\end{aligned}$$

$$
\begin{aligned}
&\quad\{\text{by (4.16)}\}\\
&= t_f - r_{l,j} + \zeta_l - \kappa_l\\
&\quad\{\text{by (4.2)}\}\\
&= t_f - t_o + t_e - r_{l,j} - \kappa_l\\
&\quad\{\text{by Lemma 4.4}\}\\
&\le t_f - t_o + \kappa_l - \kappa_l\\
&= t_f - t_o.
\end{aligned}
$$

Hence, by Lemma 4.3, (4.12) holds with $H = \psi_l^{max}$, which contradicts the assumption of ψ_l^{max} being a solution of (4.14). Therefore, $\psi_l^{max} + \kappa_l - \zeta_l$ is an upper bound of τ_l's response time.

By (4.2) and Definition 4.3, $\zeta_l \geq 0$ holds. Moreover, by (4.3) and (4.4)-(4.11), $\Omega_l(L)$ is *independent* of ζ_l, which implies that ψ_l^{max} is independent of ζ_l. Thus, the maximum value for the term $\psi_l^{max} + \kappa_l - \zeta_l$, which is given by $\psi_l^{max} + \kappa_l$ when setting $\zeta_l = 0$, is an upper bound of τ_l's response time. □

Note that (4.14) depends on $s_{l,j}$. Thus, it is necessary to test each possible value of $s_{l,j} \in \{0, 1, 2, ..., s_l\}$ to find a corresponding minimum solution of (4.14). By the definition of ψ_l^{max}, ψ_l^{max} can then be safely used to upper-bound τ_l's response time. Moreover, for every task $\tau_i \in \tau$, $\psi_i^{max} \leq d_i + \lambda_i - \kappa_i$ must hold in order for τ to be schedulable; otherwise, some jobs of τ_i may have missed their deadlines by more than the corresponding tardiness thresholds. The following corollary immediately follows.

Corollary 3. Task system τ is GTFP-schedulable upon m processors if, by repeating the iteration stated in Theorem 4.1 for all tasks $\tau_i \in \tau$, $\psi_i^{max} \leq d_i + \lambda_i - \kappa_i$ holds.

Comparing with [53]. In [53], an RTA technique, which we refer to as "GY" for short, was proposed to handle ordinary arbitrary-deadline sporadic task systems (without suspensions). (Note that GY is the only prior work that considers multiprocessor RTA techniques for arbitrary-deadline task systems.) In GY, the methodology used for the constrained-deadline case is extended for dealing with the arbitrary-deadline case by recursively solving an RTA equation. This recursive process could iterate many times depending on task parameters, and may not terminate in some rare cases.

On the other hand, due to the fact that our analysis used a more suitable interval analysis framework for arbitrary-deadline task systems (which applies to constrained-deadline task systems as well), for any task in an ordinary sporadic task systems (without suspensions), its response-time bound can be found by solving the RTA equation (4.14) only once and our RTA process always terminates. Specifically, we analyzed the eligible time t_e of our job of interest $\tau_{l,j}$, instead of its release time $r_{l,j}$ as done in [53]. In this way, when computing workload and interference, we already considered the case where job $\tau_{l,k}$ where $k < j$ might complete beyond $r_{l,j}$ if $d_l > p_l$ holds. On the contrary, in GY, the analyzed interval is extended to include all prior jobs that may complete beyond $r_{l,j}$ and an RTA equation is recursively solved to find the response-time bound. As shown by experiments presented in Section 4.2.4, our analysis has better runtime performance than GY.

4.2 GEDF

In this section, we present a GEDF schedulability test for sporadic self-suspending task systems. Our goal is to identify sufficient conditions for ensuring that each task τ_i cannot miss any deadlines by more than its predefined tardiness threshold, λ_i. These conditions must be checked for each of the n tasks in τ.

Let S be a GEDF schedule of τ such that a job $\tau_{l,j}$ of task τ_l is the first job in S to miss its deadline at $t_d = d_{l,j}$ by more than its predefined tardiness threshold λ_l, as shown in Figure 4.4. Under GEDF, jobs with lower priorities than $\tau_{l,j}$ do not affect the scheduling of $\tau_{l,j}$ and jobs with higher priorities than $\tau_{l,j}$, so we will henceforth discard from S all jobs with priorities lower than $\tau_{l,j}$.

Similar to Section 4.1, we extend the analyzed interval from $\tau_{l,j}$'s eligible time t_e (see Definition 4.2) to an earlier time instant t_o as defined below.

Definition 4.6. t_o denotes the earliest time instant at or before t_e such that there is no idleness in $[t_o, t_e)$.

Our goal now is to identify conditions necessary for $\tau_{l,j}$ to miss its deadline by more than λ_l; i.e., for $\tau_{l,j}$ to execute its computation and suspension phases for strictly fewer than $e_l + s_{l,j}$ time units over $[t_e, t_d + \lambda_l)$. This can happen only if all m processors execute jobs other than $\tau_{l,j}$ for strictly more than $(t_d + \lambda_l - t_e) - (e_l + s_{l,j})$ time units (i.e., at least $t_d + \lambda_l - t_e - e_l - s_{l,j} + 1$ time units)

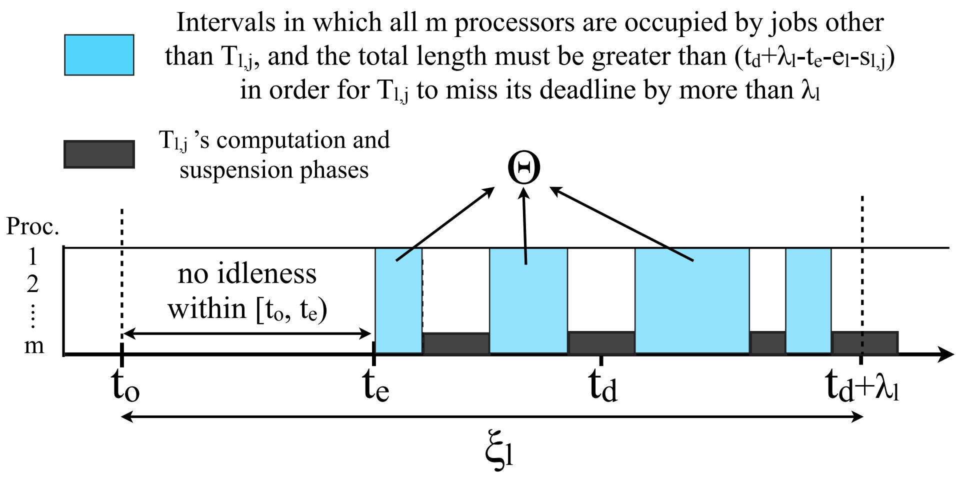

Intervals in which all m processors are occupied by jobs other than T_{l,j}, and the total length must be greater than (td+λl-te-el-sl,j) in order for T_{l,j} to miss its deadline by more than λl
T_{l,j} 's computation and suspension phases
Θ
Proc.
1
2
⋮
m
no idleness within [to, te)
to
te
td
td+λl
ξl

Figure 4.4: A job $\tau_{l,j}$ of task τ_l becomes eligible at t_e and misses its deadline at t_d by more than λ_l. t_o is the earliest time instant at or before t_e such that there is no idleness in $[t_o, t_e)$.

over $[t_e, t_d + \lambda_l)$ (for otherwise, $\tau_{l,j}$ would complete by $t_d + \lambda_l$), as illustrated in Figure 4.4. For conciseness, let

$$\xi_l = t_d + \lambda_l - t_o. \tag{4.17}$$

Definition 4.7. Let Θ denote a subset of the set of intervals within $[t_e, t_d + \lambda_l)$, where $\tau_{l,j}$ does not execute or suspend, such that the cumulative length of Θ is exactly $t_d + \lambda_l - t_e - e_l - s_{l,j} + 1$ over $[t_e, t_d + \lambda_l)$. As seen in Figure 4.4, Θ may not be contiguous.

By Definition 4.7, the length of the intervals in $[t_o, t_e) \cup \Theta$ is given by $t_e - t_o + t_d + \lambda_l - t_e - e_l - s_{l,j} + 1 = t_d + \lambda_l - t_o - e_l - s_{l,j} + 1 \overset{\{\text{by (4.17)}\}}{=} \xi_l - e_l - s_{l,j} + 1$.

For each task τ_i, let $W(\tau_i)$ denote the contribution of τ_i to the work done in S during $[t_o, t_e) \cup \Theta$. In order for $\tau_{l,j}$ to miss its deadline, it is necessary that the total amount of work that executes over $[t_o, t_e) \cup \Theta$ satisfies

$$\sum_{\tau_i \in \tau} W(\tau_i) > m \cdot (\xi_l - e_l - s_{l,j}). \tag{4.18}$$

This follows from the observation that all m processors are, by Definitions 4.6 and 4.7, completely busy executing work over the $\xi_l - e_l - s_{l,j} + 1$ time units in the interval $[t_o, t_e) \cup \Theta$.

Condition (4.18) is a necessary condition for $\tau_{l,j}$ to miss its deadline by more than λ_l. Thus, in order to show that τ is GEDF-schedulable, it suffices to demonstrate that Condition (4.18) cannot be satisfied for any task τ_l for any possible values of ξ_l and $s_{l,j}$.

We now construct a schedulability test using Condition (4.18) as follows. In Section 4.2.1, we first derive an upper bound for the term $\sum_{\tau_i \in \tau} W(\tau_i)$ in the LHS of Condition (4.18). Then, in Section 4.2.2, we compute possible values of the term $m \cdot (\xi_l - e_l - s_{l,j})$ in the RHS of Condition (4.18). Later, in Section 4.2.3, a schedulability test is derived based on these results.

4.2.1 Upper-Bounding $\sum_{\tau_i \in \tau} W(\tau_i)$

In this section, we derive an upper bound on $\sum_{\tau_i \in \tau} W(\tau_i)$, by first upper-bounding $W(\tau_i)$ for each task τ_i and then summing these per-task upper bounds.

In the following, we compute upper bounds on $W(\tau_i)$. If τ_i has no carry-in job (defined in Definition 4.4), then let $W_{nc}(\tau_i)$ denote this upper bound; otherwise, let $W_c(\tau_i)$ denote the upper bound. Since $\tau_{l,j}$ is the first job that misses its deadline at t_d by more than its corresponding tardiness threshold, we have

$$f_{l,j-1} \leq d_{l,j-1} + \lambda_l \leq t_d - p_l + \lambda_l. \tag{4.19}$$

The following lemma bounds the length of time interval $[t_o, t_e)$.

Lemma 4.5. $t_e - t_o \leq max(\xi_l - \lambda_l - d_l, \xi_l - p_l)$.

Proof. $t_e - t_o \overset{\{\text{by Definition 4.2}\}}{=} max(r_{l,j}, f_{l,j-1}) - t_o \overset{\{\text{by (4.19)}\}}{\leq} max(t_d - d_l, t_d - p_l + \lambda_l) - t_o = max(t_d - d_l - t_o, t_d - p_l + \lambda_l - t_o) \overset{\{\text{by (4.17)}\}}{=} max(\xi_l - \lambda_l - d_l, \xi_l - p_l)$. □

If a task τ_i has no carry-in job, then the total amount of work that must execute over $[t_o, t_e) \cup \Theta$ is generated by jobs of τ_i arriving in, and having deadlines within, the interval $[t_o, t_d]$. The following lemma, which was originally proved for ordinary sporadic task systems in [16], applies to sporadic self-suspending task systems as well.

Lemma 4.6. *The maximum cumulative execution requirement by jobs of an sporadic self-suspending task τ_i that both arrive in, and have deadlines within, any interval of length t is given by demand bound function*

$$DBF(\tau_i, t) = max(0, (\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1) \cdot e_i).$$

Proof. Because we restrict attention to jobs of τ_i that have releases and deadlines within the considered interval of length t, and suspensions do not occupy any processor, the required total work

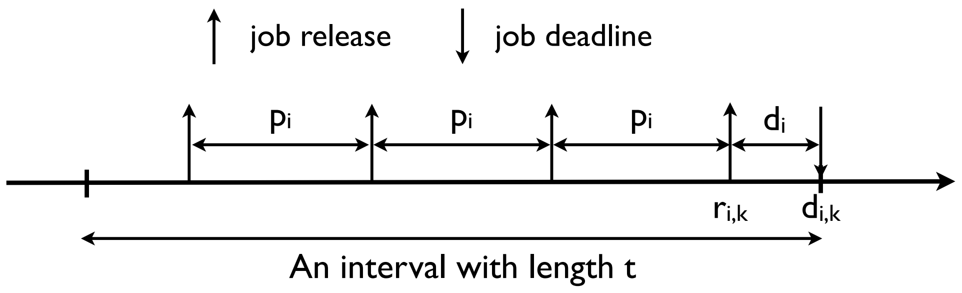


Figure 4.5: DBF for self-suspending tasks.

of τ_i can be bounded by considering the scenario in which some job $\tau_{i,k}$ of τ_i has a deadline at the end of the interval and jobs are released periodically. This scenario is illustrated in Figure 4.5. There are at most $\left\lfloor \frac{t-d_i}{p_i} \right\rfloor$ jobs that are released and have deadlines within the interval other than $\tau_{i,k}$. Thus, the maximum cumulative execution requirement by jobs of τ_i is given by $DBF(\tau_i, t) = max(0, (\left\lfloor \frac{t-d_i}{p_i} \right\rfloor + 1) \cdot e_i)$, which is the same as the DBF for ordinary sporadic tasks (i.e., without suspensions). This is due to the fact that suspensions do not contribute to the execution requirement. □

The lemma below computes $W_{nc}(\tau_i)$ using DBF.

Lemma 4.7.

$$W_{nc}(\tau_i) = \begin{cases} min\big(DBF(\tau_i, \xi_l - \lambda_l), \\ \quad \xi_l - e_l - s_{l,j} + 1\big) & if\, i \neq l \\ min\big(DBF(\tau_l, \xi_l - \lambda_l) - e_l, \\ \quad max(\xi_l - \lambda_l - d_l, \xi_l - p_l)\big) & if\, i = l \end{cases}$$

Proof. Depending on the relationship between i and l, there are two cases to consider.

Case $i \neq l$**.** The total amount of work contributed by τ_i that must execute over $[t_o, t_e) \cup \Theta$ cannot exceed the total length of the intervals in $[t_o, t_e) \cup \Theta$, which is $\xi_l - e_l - s_{l,j} + 1$. Furthermore, the total work that needs to be bounded must have releases and deadlines within the interval $[t_o, t_d]$, which by (4.17) is of length $\xi_l - \lambda_l$. By Lemma 4.6, this total work is at most $DBF(\tau_i, \xi_l - \lambda_l)$.

Case $i = l$**.** As in the previous case, the total work is at most $DBF(\tau_l, \xi_l - \lambda_l)$. However, in this case, since $\tau_{l,j}$ does not execute within $[t_o, t_e) \cup \Theta$, we can subtract its execution requirement, which is e_l, from $DBF(\tau_l, \xi_l - \lambda_l)$. Also, this contribution cannot exceed the length of the interval $[t_o, t_e)$, which by Lemma 4.5 is at most $max(\xi_l - \lambda_l - d_l, \xi_l - p_l)$. $\square$

We now consider the case where τ_i has a carry-in job. The following lemma, which computes $W_c(\tau_i)$, is proved similarly to Lemma 4.7.

Lemma 4.8.

$$W_c(\tau_i) = \begin{cases} min(\Delta(\tau_i, \xi_l - \lambda_l + \lambda_i), & \\ \quad \xi_l - e_l - s_{l,j} + 1) & if\, i \neq l \\ min(\Delta(\tau_l, \xi_l) - e_l, & \\ \quad max(\xi_l - \lambda_l - d_l, \xi_l - p_l). & if\, i = l \end{cases}$$

Proof. The total work of τ_i in this case can be upper-bounded by considering the scenario in which some job of τ_i has a deadline at t_d and jobs of τ_i are released periodically, as illustrated in Figure 4.6. Depending on the relationship between i and l, we have two cases to consider.

Case $i \neq l$**.** Let $\tau_{i,k}$ be the first job such that $r_{i,k} < t_o$ and

$$d_{i,k} + \lambda_i > t_o, \tag{4.20}$$

i.e., $\tau_{i,k}$ is the first job of τ_i (potentially tardy) that may execute during $[t_o, t_d)$ and is released before t_o (note that if $\tau_{i,k}$ does not exist then τ_i would have no carry-in job). Since jobs are released periodically,

$$t_d - d_{i,k} = x \cdot p_i \tag{4.21}$$

holds for some integer x.

The demand for jobs of τ_i in this case is thus bounded by the demand due to x jobs that have deadlines at or before t_d and are released at or after $r_{i,k} + p_i$, plus the demand imposed by the job $\tau_{i,k}$, which cannot exceed the smaller of e_i and the length of the interval $[t_o, d_{i,k} + \lambda_i)$, which by (4.21) is $t_d - x \cdot p_i + \lambda_i - t_o \overset{\{\text{by (4.17)}\}}{=} \xi_l - \lambda_l + \lambda_i - x \cdot p_i$. Thus, we have

$$W_c(\tau_i) = x \cdot e_i + min(e_i, \xi_l - \lambda_l + \lambda_i - x \cdot p_i). \tag{4.22}$$

To find x, by (4.21), we have

$$\begin{aligned}
&x\\
=&\frac{t_d - d_{i,k}}{p_i}\\
&\{\text{by } (4.20)\}\\
<&\frac{t_d - t_o + \lambda_i}{p_i}\\
&\{\text{by } (4.17)\}\\
=&\frac{\xi_l - \lambda_l + \lambda_i}{p_i}.
\end{aligned}$$

For conciseness, let $\pi = \xi_l - \lambda_l + \lambda_i$. Thus, $x < \dfrac{\pi}{p_i}$ holds. If $\pi \bmod p_i = 0$, then $x \le \dfrac{\pi}{p_i} - 1 = \left\lceil \dfrac{\pi}{p_i} \right\rceil - 1$, otherwise, $x \le \left\lfloor \dfrac{\pi}{p_i} \right\rfloor = \left\lceil \dfrac{\pi}{p_i} \right\rceil - 1$. Thus, a general expression for x can be given by $x \le \left\lceil \dfrac{\pi}{p_i} \right\rceil - 1$.

By (4.22), the maximum value for $W_c(\tau_i)$ can be obtained when $x = \left\lceil \dfrac{\pi}{p_i} \right\rceil - 1$. Setting this expression for x into (4.22), we have

$$\begin{aligned}
&W_c(\tau_i)\\
=&(\left\lceil \frac{\pi}{p_i} \right\rceil - 1) \cdot e_i + min\left(e_i, \pi - \left\lceil \frac{\pi}{p_i} \right\rceil \cdot p_i + p_i\right)\\
&\{\text{by Definition 4.5}\}\\
=&\Delta(\tau_i, \pi)\\
&\{\text{by the definition of } \pi\}\\
=&\Delta(\tau_i, \xi_l - \lambda_l + \lambda_i).
\end{aligned}$$

Moreover, this total demand cannot exceed the total length of the intervals in $[t_o, t_e) \cup \Theta$, which is $\xi_l - e_l - s_{l,j} + 1$.

Case $i = l$**.** Repeating the reasoning from the previous case, we find that the total demand of jobs of τ_l with deadlines at most t_d is at most $\Delta(i, \xi_l - \lambda_l + \lambda_i) = \Delta(\tau_i, \xi_l)$. Since $\tau_{l,j}$ does not execute within $[t_o, t_e) \cup \Theta$, we subtract its execution requirement, which is e_l, from $\Delta(\tau_i, \xi_l)$.

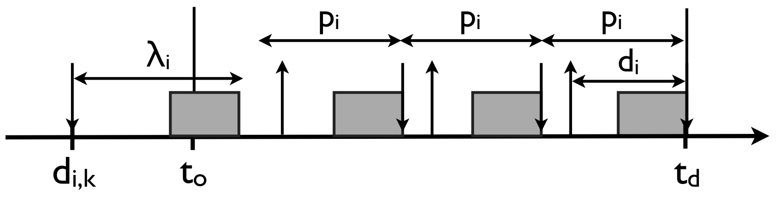


Figure 4.6: Computing $W_c(\tau_i)$.

Also, this contribution cannot exceed the length of the interval $[t_o, t_e)$, which by Lemma 4.5 is $max(\xi_l - \lambda_l - d_l, \xi_l - p_l)$. □

Upper-bounding $\sum_{\tau_i \in \tau} W(\tau_i)$. Similar to the discussion in Section 4.1, by Definition 4.6, either $t_o = 0$, in which case no task has a carry-in job, or some processor is idle in $[t_o - 1, t_o)$, in which at most $m - 1$ computational tasks are active at $t_o - 1$. Thus, at most $min(m - 1, n_e)$ computational tasks can have a carry-in job. However, since suspensions do not occupy any processor, each self-suspending task may be active at $t_o - 1$ and have a job that is suspended at t_o. Thus, in the worst case, all n_s self-suspending tasks can have carry-in jobs. Consequently, there are at most n_s self-suspending tasks and $min(m - 1, n_e)$ computational tasks that contribute $W_c(\tau_i)$ work, and the remaining $max(0, n_e - m + 1)$ computational tasks must contribute to $W_{nc}(\tau_i)$. Thus, self-suspending tasks can contribute at most $\sum_{\tau_i \in \tau^s} max\big(W_{nc}(\tau_i), W_c(\tau_i)\big)$ work to $\sum_{\tau_i \in \tau} W(\tau_i)$. Let $\delta_{\tau_i \in \tau^e}^{min(m-1,n_e)}$ denote the $min(m - 1, n_e)$ greatest values of $max(0, W_c(\tau_i) - W_{nc}(\tau_i))$ for any computational task τ_i. Then computational tasks can contribute at most $\sum_{\tau_j \in \tau^e} W_{nc}(\tau_j) + \delta_{\tau_k \in \tau^e}^{min(m-1,n_e)}$ work to $\sum_{\tau_i \in \tau} W(\tau_i)$. Therefore, by summing up the work contributed by both self-suspending tasks and computational tasks, we can bound $\sum_{\tau_i \in \tau} W(\tau_i)$ by $\sum_{\tau_i \in \tau^s} max(W_{nc}(\tau_i), W_c(\tau_i)) + \sum_{\tau_j \in \tau^e} W_{nc}(\tau_j) + \delta_{\tau_k \in \tau^e}^{min(m-1,n_e)}$.

Similar to the discussion in Section 4.1, the time complexity for computing $W_c(\tau_i)$, $W_{nc}(\tau_i)$, and $W_c(\tau_i) - W_{nc}(\tau_i)$ is $O(n)$. Also, by using a linear-time selection technique from [24], the time complexity for computing $\delta_{\tau_k \in \tau^e}^{min(m-1,n_e)}$ is $O(n)$. Thus, the time complexity to upper-bound $\sum_{\tau_i \in \tau} W(\tau_i)$ as above is $O(n)$.

4.2.2 Finding Values of ξ_l and $s_{l,j}$

So far we have upper-bounded the LHS of Condition (4.18). Recall that our goal is to test Condition (4.18) for a violation for all possible values of ξ_l and $s_{l,j}$. The following theorem shows that the range of possible values of ξ_l that need to be tested can be limited. Let e_{sum} be the sum of the execution costs for all tasks in τ. For conciseness, let $\phi = m \cdot (e_l + s_{l,j}) - \lambda_l \cdot u_{sum} + \sum_{\tau_i \in \tau} \lambda_i \cdot u_i + e_{sum}$.

Theorem 4.2. *If Condition (4.18) is satisfied for τ_l, then it is satisfied for some ξ_l satisfying*

$$min(d_l + \lambda_l, p_l) \leq \xi_l < \frac{\phi}{m - u_{sum}}, \tag{4.23}$$

provided $u_{sum} < m$

Proof. By Lemmas 4.6 and 4.7, $W_{nc}(\tau_i) \leq \left\lfloor \frac{\xi_l - \lambda_l}{p_i} \right\rfloor \cdot e_i + e_i$ holds. By Lemma 4.8 and Definition 4.5, $W_c(\tau_i) \leq \left\lfloor \frac{\xi_l - \lambda_l + \lambda_i}{p_i} \right\rfloor \cdot e_i + e_i$ holds. Thus, the LHS of Condition (4.18) is no greater than $\sum_{\tau_i \in \tau} \left(\left\lfloor \frac{\xi_l - \lambda_l + \lambda_i}{p_i} \right\rfloor \cdot e_i + e_i \right)$. Assuming Condition (4.18) is satisfied, we have

$$\begin{aligned}
& \textstyle\sum_{\tau_i \in \tau} W(\tau_i) > m \cdot (\xi_l - e_l - s_{l,j}) \\
\Rightarrow \quad & \{\text{upper-bounding } \textstyle\sum_{\tau_i \in \tau} W(\tau_i) \text{ as above}\} \\
& \textstyle\sum_{\tau_i \in \tau} \left(\left\lfloor \frac{\xi_l - \lambda_l + \lambda_i}{p_i} \right\rfloor \cdot e_i + e_i \right) > m \cdot (\xi_l - e_l - s_{l,j}) \\
\Rightarrow \quad & \{\text{removing the floor}\} \\
& \textstyle\sum_{\tau_i \in \tau} \Big((\xi_l - \lambda_l + \lambda_i) \cdot u_i + e_i \Big) > m \cdot (\xi_l - e_l - s_{l,j}) \\
\Rightarrow \quad & \{\text{rearranging}\} \\
& \xi_l \cdot u_{sum} - \lambda_l \cdot u_{sum} + \textstyle\sum_{\tau_i \in \tau} \lambda_i \cdot u_i + e_{sum} \\
& \quad > m \cdot \xi_l - m \cdot e_l - m \cdot s_{l,j} \\
\Rightarrow \quad & \xi_l < \frac{\phi}{m - u_{sum}},
\end{aligned}$$

provided $u_{sum} < m$.

Moreover, we have

ξ_l

$\{\text{by (4.17)}\}$

$=t_d - t_o + \lambda_l$

$\{\text{by Definition 4.6}\}$

$\geq t_d - t_e + \lambda_l$

$\{\text{by Definition 4.2}\}$

$=t_d - max(r_{l,j}, f_{l,j-1}) + \lambda_l$

$\{\text{by (4.19)}\}$

$\geq t_d - max(t_d - d_l, t_d - p_l + \lambda_l) + \lambda_l$

$=min(d_l, p_l - \lambda_l) + \lambda_l$

$=min(d_l + \lambda_l, p_l).$ □

Possible values for $s_{l,j}$. By Lemmas 4.7 and 4.8, $\sum_{\tau_i \in \tau} W(\tau_i)$, which is the LHS of Condition (4.18), *depends on* the value of $s_{l,j}$ non-monotonically. Moreover, by Theorem 4.2, ξ_l also depends on the value of $s_{l,j}$ (recall ϕ). Thus, it is necessary to test all possible values of $s_{l,j}$, which are $\{0, 1, 2, ..., s_l\}$.

4.2.3 Schedulability Test

Theorem 4.3. *Task system τ is GEDF-schedulable on m processors if for all tasks τ_l and all values of ξ_l satisfying (4.23),*

$$\begin{aligned}\sum_{\tau_i \in \tau} max\Big(W_{nc}\left((\tau_i), W_c(\tau_i)\right) + \sum_{\tau_j \in \tau^e} W_{nc}(\tau_j) \\ +\delta_{\tau_k \in \tau^e}^{min(m-1, n_e)}\Big) \leq m \cdot (\xi_l - e_l - s_{l,j})\end{aligned} \tag{4.24}$$

holds for every value of $s_{l,j} \in \{0, 1, 2, ..., s_l\}$.

By Theorem 4.2, we can test Condition (4.24) in time pseudo-polynomial in the task parameters.

4.2.4 Experiments

We now describe experiments conducted using randomly-generated task sets to evaluate the performance of the proposed schedulability tests. In these experiments, several aspects of our analysis were investigated. In the following, we denote our GEDF and GTFP schedulability tests as "Our-EDF" and "Our-FP," respectively.

HRT effectiveness. We evaluated the effectiveness of the proposed techniques for HRT sporadic self-suspending task systems by comparing Our-EDF and Our-FP to the suspension-oblivious approach denoted "SC" combined with the tests in [10] and [53], which we denote "Bar" (for "Baruah") and (as noted earlier) "GY," respectively. That is, after transforming all sporadic self-suspending tasks into ordinary sporadic tasks (no suspensions) using SC, we applied Bar and GY, which are the best known schedulability tests for GEDF and GTFP, respectively. In [10], Bar was shown to overcome a major deficiency (i.e., the $O(n)$ carry-in work) of prior GEDF analysis (as discussed in Section 2.3). In [53], GY was shown to be superior to all prior analysis for ordinary task systems available at that time. Moreover, since partitioning approaches have been shown to be generally superior to global approaches on multiprocessors [8], we compared our test to SC combined with the partitioning approach proposed in [12], which we denoted "FB-Par". FB-Par is considered to be the best partitioning approach for constrained-deadline sporadic task systems.

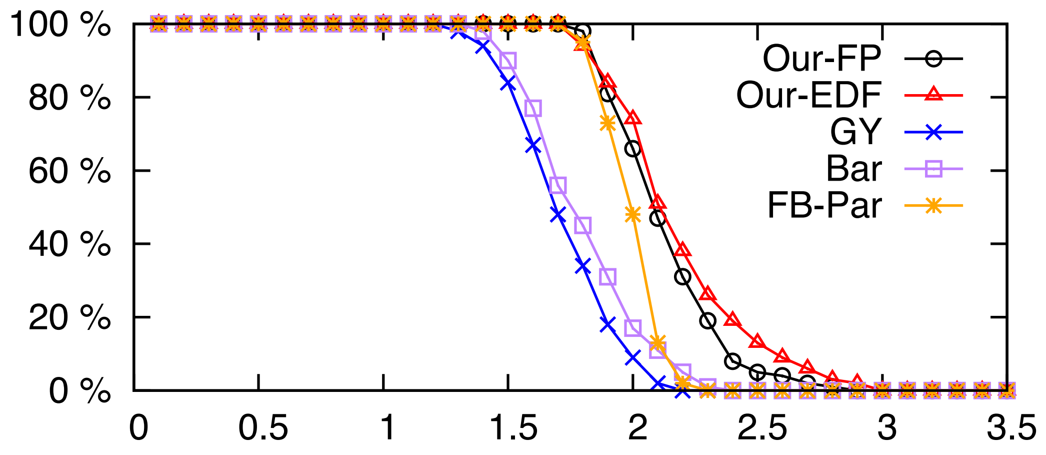
SRT effectiveness. We evaluated the effectiveness of the proposed techniques for SRT sporadic self-suspending task systems with predefined tardiness threshold by comparing them to SC combined with the test proposed in [77], which we denote "LA." LA is the only prior schedulability test for SRT ordinary task systems with predefined tardiness thresholds.

Impact of carry-in work. To evaluate the impact brought by $O(n)$ carry-in work on our analysis, we compared the HRT schedulability for sporadic self-suspending task systems using our analysis to that obtained by applying the analysis proposed in [19], which we denoted "BC," to an otherwise equivalent task system with no suspensions. In [19], BC was shown to be superior to all prior analysis assuming $O(n)$ carry-in work available at that time.

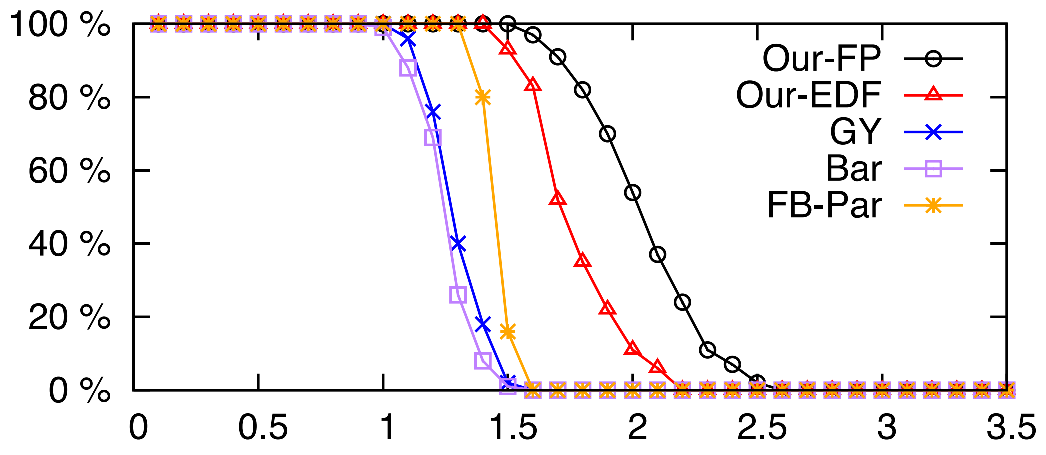
Runtime performance. Finally, we evaluated the effectiveness and the runtime performance of Our-FP for ordinary arbitrary-deadline sporadic task systems (with no suspensions) by comparing it to GY.

In our experiments, sporadic self-suspending task sets were generated based upon the methodology proposed by Baker in [8]. Integral task periods were distributed uniformly over [10ms,100ms]. Per-task utilizations were uniformly distributed in [0.01, 0.3]. Task execution costs were calculated from periods and utilizations. For any task τ_i in any generated task set, d_i/p_i was varied within [1,2] for the arbitrary-deadline case and within $[max(0.7, \frac{e_i + s_i}{p_i}), 1]$ for the constrained-deadline case, and the tardiness threshold λ_i was varied uniformly within $[0, 2 \cdot p_i]$ for SRT tasks. The suspension length for any task τ_i was generated by varying s_i/e_i as follows: 0.5 (short suspension length), 1 (moderate suspension length), and 1.5 (long suspension length). Task sets were generated for $m = 4$ processors, as follows. A cap on overall utilization was systematically varied within $[1, 1.1, 1.2, ..., 3.9, 4]$. For each combination of utilization cap and suspension length, we generated 1,000 sporadic self-suspending task sets. Each such sporadic self-suspending task set was generated by creating sporadic self-suspending tasks until total utilization exceeded the corresponding utilization cap, and by then reducing the last task's utilization so that the total utilization equalled the utilization cap. For GTFP scheduling, priorities were assigned on a global deadline-monotonic basis. In all figures presented in this section, the x-axis denotes the utilization cap and the y-axis denotes the fraction of generated task sets that were schedulable.

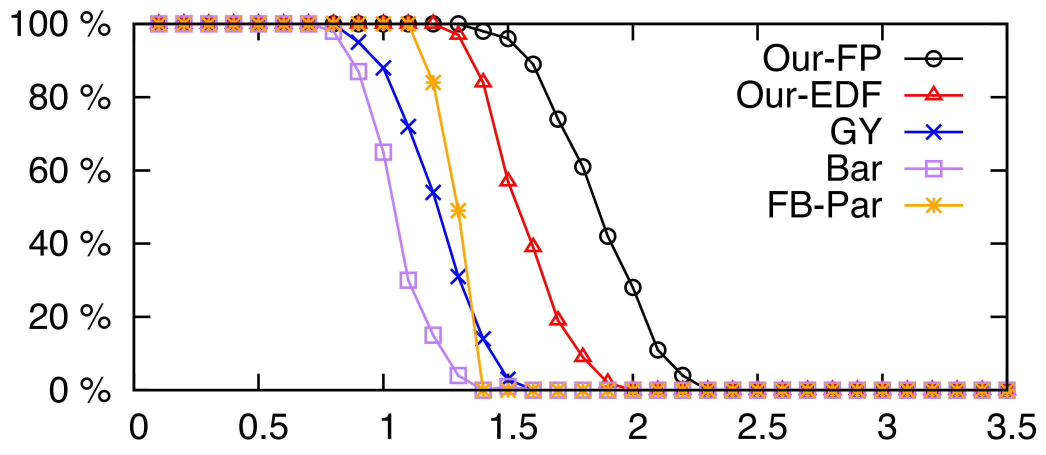
Figures 4.7 and 4.8 show HRT and SRT schedulability results for constrained-deadline sporadic self-suspending task sets achieved by using Our-EDF, Our-FP, Bar, GY, and FB-Par. As seen, for both the HRT and the SRT cases, Our-EDF and Our-FP improve upon the other tested alternatives. Notably, Our-EDF and Our-FP consistently yield better schedulability results than the partitioning approach FB-Par, as seen in Figures 4.7. This is due to the fact that, after treating suspensions as computation, FB-Par suffers from bin-packing-related capacity loss. Moreover, as the suspension length increases, such performance improvement also increases. This is because treating suspension as computation becomes more pessimistic as the suspension length increases. This result suggests that a task's suspensions do not negatively impact the schedulability of other tasks as much as computation does.



(a) HRT: short suspensions

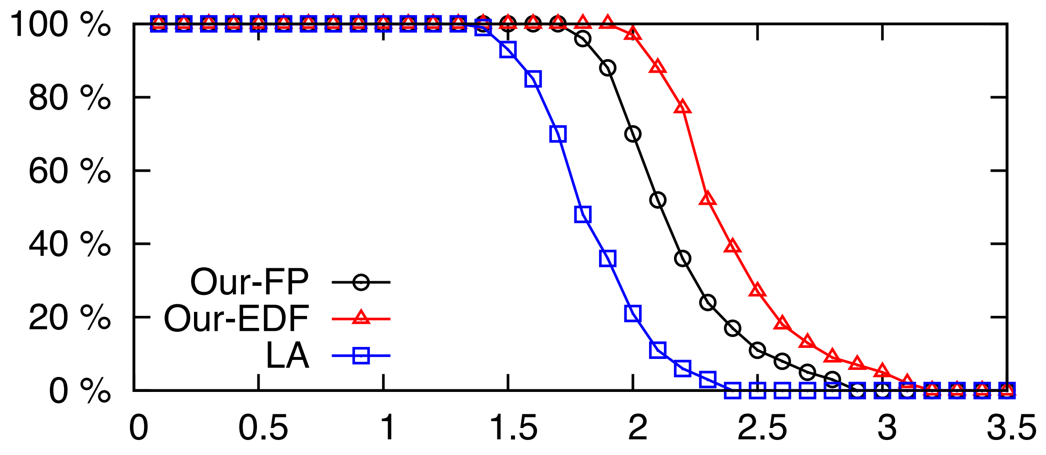


(b) HRT: moderate suspensions

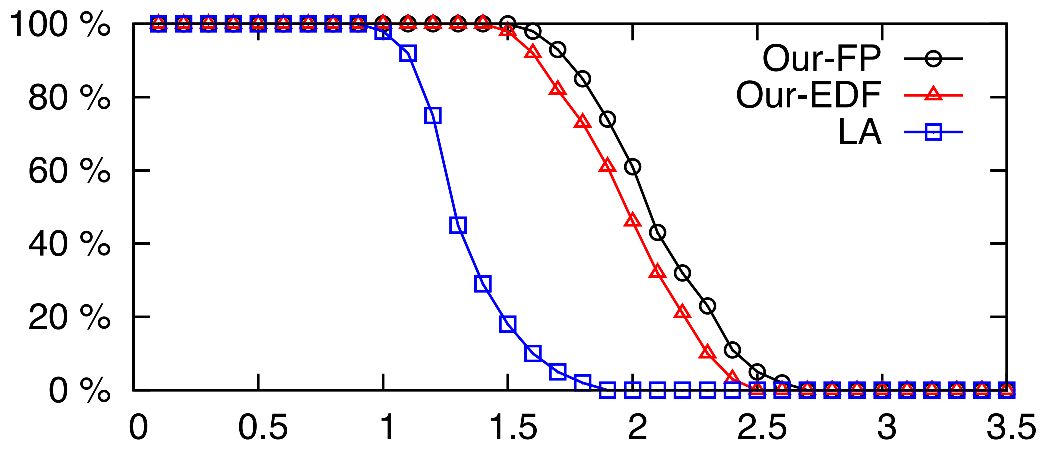


(c) HRT: long suspensions

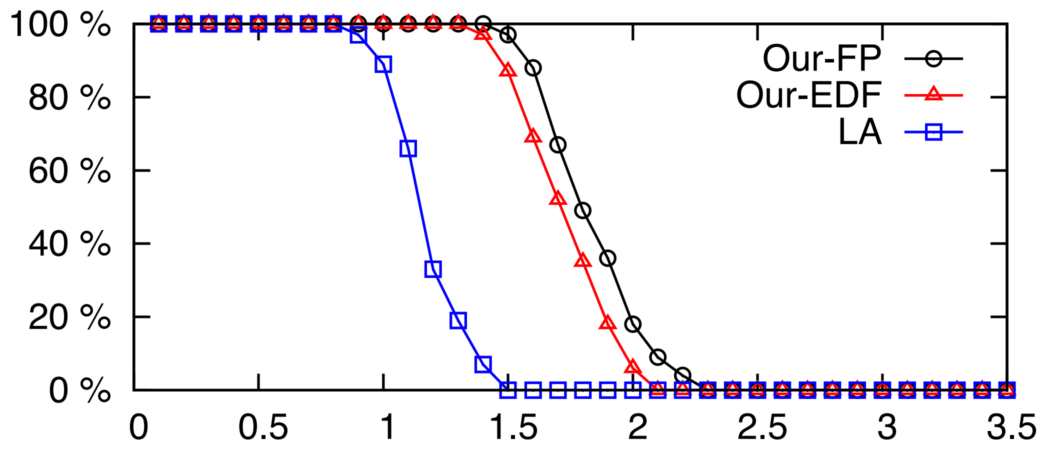
Figure 4.7: HRT results. $u_i \in [0.01, 0.3]$, $d_i \in [max(0.7 \cdot p_i, e_i + s_i), p_i]$.



(a) SRT: short suspensions



(b) SRT: moderate suspensions



(c) SRT: long suspensions

Figure 4.8: SRT results. $u_i \in [0.01, 0.3]$, $d_i \in [max(0.7 \cdot p_i, e_i + s_i), p_i]$.

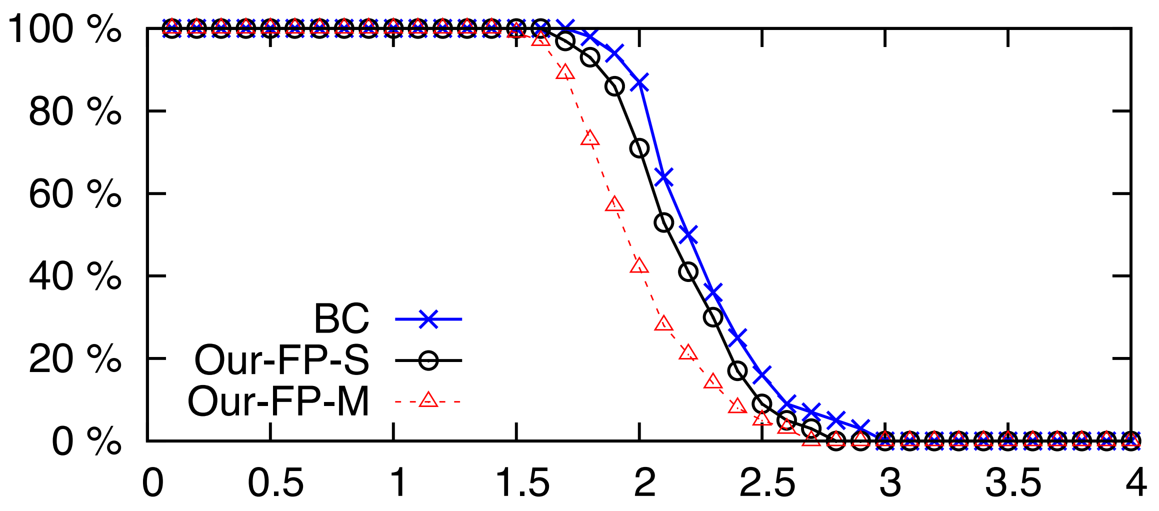


Figure 4.9: HRT results compared with BC.

Figure 4.9 shows HRT schedulability results for constrained-deadline sporadic self-suspending task sets achieved by using Our-FP and by applying BC to otherwise equivalent task sets with no suspensions. In Figure 4.9, "Our-FP-S" (respectively, "Our-FP-M") represents schedulability results achieved by Our-FP for the task sets (which are originally generated for BC with no suspensions) after adding suspensions by setting $\frac{s_i}{e_i} = 0.2$ (respectively, $\frac{s_i}{e_i} = 0.5$). It can be seen that Our-FP yields schedulability results that are very close to that achieved by BC. For task sets with $\frac{s_i}{e_i} = 0.2$, Our-FP and BC achieved almost identical schedulability results. This shows that the negative impact brought by suspensions is mainly caused by forcing $O(n)$ carry-in work.

Figure 4.10 shows HRT schedulability results for arbitrary-deadline ordinary task systems (with no suspensions) achieved by using Our-FP and GY. In this experiment, for each choice of the utilization cap, 10,000 task sets are generated. As seen, Our-FP slightly improves upon GY. Moreover, Figure 4.10 also shows the total time for running this entire experiment for Our-FP and GY. As seen, Our-FP runs much faster ($> 10\times$) than GY, due to the fact that Our-FP can find any task's response time by solving the RTA equation only once.

4.3 Chapter Summary

In this chapter, we presented the first suspension-aware multiprocessor schedulability tests for globally-scheduled HRT sporadic self-suspending task systems under both GTFP and GEDF schedul-

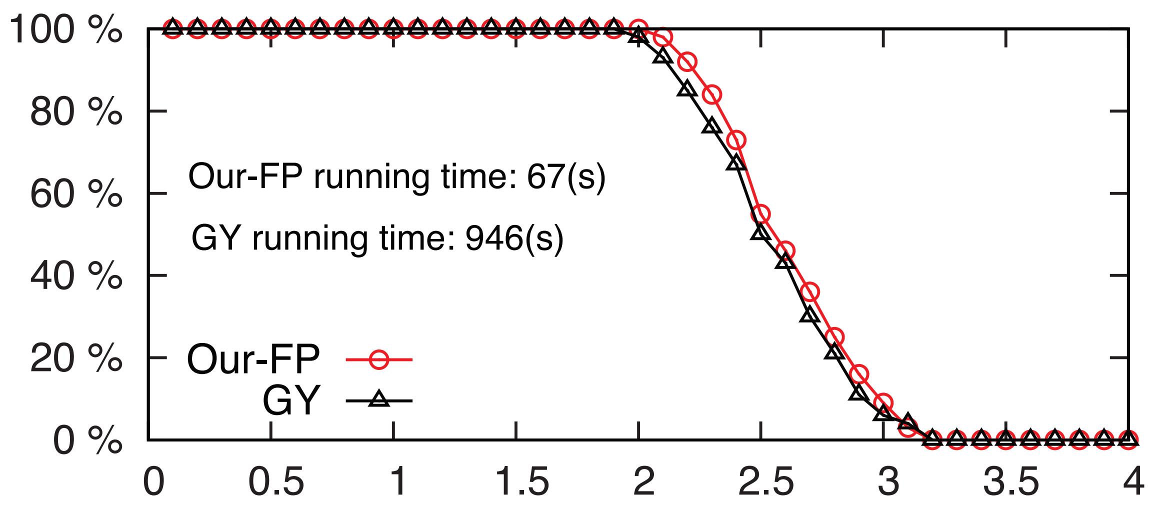


Figure 4.10: HRT results compared with GY.

ing. We also presented experimental results that show that our suspension-aware analysis is much superior to the prior suspension-oblivious approach. Also, when applied to ordinary arbitrary-deadline task systems with no suspensions, our fixed-priority analysis has a lower running time than that proposed in [53].

CHAPTER 5

Multiprocessor Scheduling of PGM Graphs[1]

In this chapter, we consider the problem of scheduling multiprocessor implementations of real-time systems specified as directed-acyclic-graphs (DAGs). If all deadlines in such a system are viewed as hard, and tasks execute sporadically (or periodically), then DAG-based systems can be easily supported by assigning a common period to all tasks in a DAG and by adjusting job releases so that successive tasks execute in sequence. Figure 5.1 shows an example of scheduling a DAG τ_1 on a two-processor system consisting of four sporadic tasks, τ_1^1, τ_1^2, τ_1^3, and τ_1^4. (DAG-based systems are formally defined in Chapter 2. It suffices to know here that the k^{th} job of τ_1^1, τ_1^2 (or τ_1^3), and τ_1^4, respectively, must execute in sequence.) As seen in this example, the timing guarantees provided by the sporadic model ensure that any DAG executes correctly as long as no deadlines are missed.

However, if all deadlines in a multiprocessor sporadic task system must be viewed as hard, then significant processing capacity must be sacrificed, due to either inherent schedulability-related capacity loss—which is unavoidable under most scheduling schemes—or high runtime overheads—which typically arise in optimal schemes that avoid schedulability-related loss. In systems where less stringent notions of real-time correctness suffice, such loss can be avoided by viewing deadlines as soft. Such systems are also our focus; the notion of soft real-time correctness we consider is that deadline tardiness is bounded.

Unfortunately, if deadlines can be missed, then DAGs are not as easy to support as ordinary sporadic tasks. For example, if the first job of τ_1^1 in Figure 5.1 were to miss its deadline, then its

[1]Contents of this chapter previously appeared in preliminary form in the following papers:
Cong Liu and James Anderson. Supporting Soft Real-Time DAG-based Systems on Multiprocessors with No Utilization Loss, Proceedings of the 31st IEEE Real-Time Systems Symposium, pages 3-13, 2010.
Cong Liu and James Anderson. Supporting Graph-Based Real-Time Applications in Distributed Systems, Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pages 143-152, 2011.

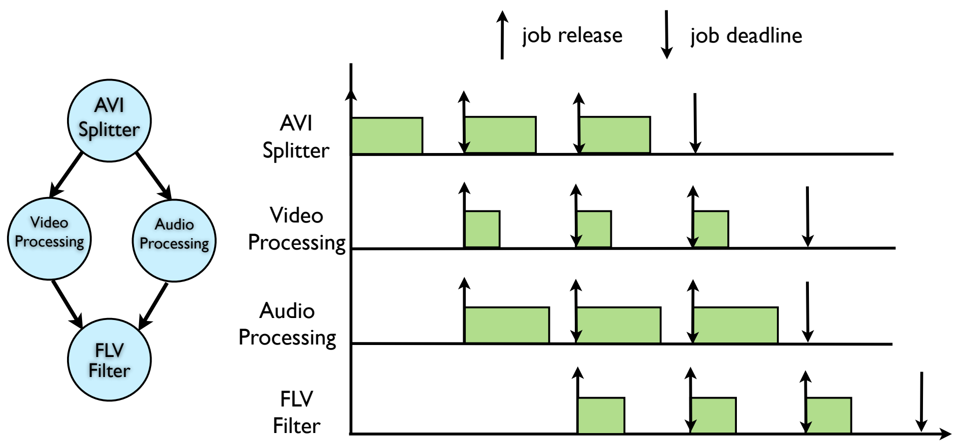


Figure 5.1: Example DAG.

execution might overlap that of the first jobs of τ_1^2 and τ_1^3. This violates the requirement that instances of successive DAG vertices must execute in sequence. In this chapter, we address this lack of support by presenting scheduling techniques and analysis that can be applied to support DAG-based systems on multiprocessors with no capacity loss, assuming that bounded deadline tardiness is the timing guarantee that must be ensured. Our results can be applied to systems with rather sophisticated precedence constraints. To illustrate this, we consider a particularly expressive DAG-based formalism, namely PGM (recall Section 2.3.3 of Chapter 2). We show that PGM-specified systems can be scheduled with bounded tardiness on multiprocessors with no capacity loss.

In this chapter, we show that sophisticated notions of acyclic precedence constraints can be supported under GEDF on multiprocessors and in distributed systems, provided bounded deadline tardiness is acceptable. The types of precedence constraints we consider are those allowed by PGM and studied previously in the uniprocessor case by Goddard [51]. Since any acyclic PGM graph has a natural representation as a DAG-based rate-based (RB) task system, such systems are our major focus. We specifically show that, when any such system is scheduled by GEDF, each task's maximum tardiness is bounded. We show this by transforming any such system into an ordinary sporadic system (with sporadic job releases and without precedence constraints) and by exploiting the fact the latter has bounded tardiness under GEDF. Note that, although we focus specifically on GEDF, our results can be applied to any global scheduling algorithm that can ensure bounded tardiness with no capacity loss for ordinary sporadic task systems. Moreover, due to the fact that our

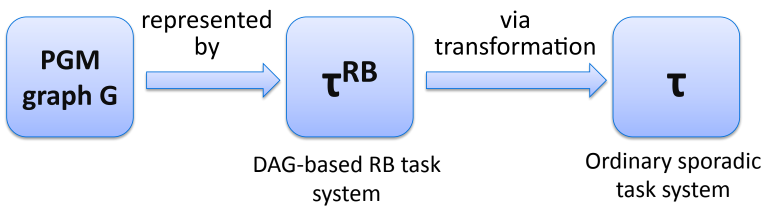


Figure 5.2: Roadmap.

RB task model is a generalization of the periodic and sporadic task models (see Section 2.1), our results are general enough to be applicable to periodic and sporadic DAG systems.

Roadmap. In this chapter, we primarily deal with PGM graphs, DAG-based RB task systems, and ordinary sporadic task systems (see Section 2.1 for definitions of these task models). For clarity, we let G denote a PGM graph, τ denote an ordinary sporadic task system, and τ^{RB} denote a DAG-based RB task system, as shown in Figure 5.2.

The rest of this chapter is organized as follows. In Section 5.1, we first derive our multiprocessor schedulability test for SRT PGM graphs. In Section 5.2, we show how to support SRT PGM graphs in a distributed system. In Section 5.3, we summarize our work.

5.1 Supporting PGM-Specified Systems on Multiprocessors

In this section, we present our proposed approach for supporting PGM-specified systems on multiprocessors. We first show that any PGM graph G can be represented by a DAG-based RB task system τ^{RB} by mapping PGM nodes to RB tasks. We then show that τ^{RB} can be transformed to an ordinary sporadic task system τ, for which tardiness bounds can be derived. A summary of the terms defined so far, as well as some additional terms defined later, is presented in Table 7.1.

5.1.1 Representing PGM Graphs by DAG-based RB Task Systems

In this section, our goal is to implement G by a task system τ^{RB}, where G consists of a set of n acyclic PGM graphs $\{G_1, G_2, ..., G_n\}$. By computing valid node execution rates (as defined below) based on the producer/consumer relationships that exist in any graph G_l in G, we can implement each node in G_l by an RB task in τ^{RB}.

m	Number of processors
n	Number of tasks
U_{sum}	Total utilization of τ^{RB}
(x_i, y_i)	Execution rate of an RB task τ_i, indicating that there are at most x_i job releases within any time interval $[j \cdot y_i, (j+1) \cdot y_i)$ $(j \geq 0)$
$\rho^{k \leftarrow j}$	Produce amount of the queue connecting any two nodes G^j and G^k in a PGM graph G
$\varphi^{k \leftarrow j}$	Threshold of the queue connecting any two nodes G^j and G^k in a PGM graph G
$c^{k \leftarrow j}$	Consume amount of the queue connecting any two nodes G^j and G^k in a PGM graph G
$pred(\tau_l^h)$	Set of predecessor tasks of task τ_l^h
$pred(\tau_{l,j}^h)$	Set of predecessor jobs of job $\tau_{l,j}^h$
$F_{max}(pred(\tau_{l,j}^h))$	Latest completion time among all predecessor jobs of $\tau_{l,j}^h$
$t_f(\tau_{l,j}^h)$	Completion time of $\tau_{l,j}^h$
$\varepsilon(\tau_{l,j}^h)$	Early-release time of $\tau_{l,j}^h$

Table 5.1: Summary of notation.

Definition 5.1. [51] An *execution rate* is a pair of non-negative integers (x, y). An execution rate specification for any node G_l^i in G_l, (x_l^i, y_l^i), is *valid* if there exists a time t such that G_l^j executes *exactly* x_l^i times in time intervals $[t + (k-1) \cdot y_l^i, t + k \cdot y_l^i)$ for all $k > 0$. An execution of a node in G_l is *valid* iff: (1) the task executes only when it is eligible for execution and no two executions of the same node overlap, (2) each input queue has its tokens atomically consumed after each output queue has its tokens atomically produced, and (3) tokens are produced at most once on an output queue during each node execution. An execution of G is *valid* iff all of the nodes in the execution sequence have valid executions and no data loss occurs.

We define τ^{RB} to have the same structure as G, i.e., each graph G_l in G is implemented by a DAG-based RB task τ_l in τ^{RB}. Moreover, each node G_l^i in G_l is implemented by an RB task τ_l^i in τ_l, and these tasks are connected via edges just like the corresponding nodes in G_l. We assume that the source node of G_l is governed by an RB specification and non-source nodes execute according to the corresponding specifications in G_l (i.e., produce, threshold, and consume attributes of queues in G_l).[2]

As shown in [51], it is possible to compute a valid execution rate for every task in τ^{RB}, provided the assumption that the source node of a PGM graph executes according to a valid rate-based rate (note that we also make this assumption in this dissertation). Although the focus of [51] is uniprocessor platforms, this result is independent of the hardware platform. Thus, we first apply the same method to compute a valid execution rate for every task in τ^{RB}, as stated in the following lemma (proved in Theorem 2.4.9 on Page 74 of [51]).

Lemma 5.1. *[51] For any task τ_i^k in τ^{RB} that has at least one incoming edge, let ν denote the set of predecessor tasks of τ_i^k. For any node τ_i^u in ν, let $R_i^u = (x_i^u, y_i^u)$ be a valid execution rate. The execution rate $R_i^k = (x_i^k, y_i^k)$ for τ_i^k is valid if*

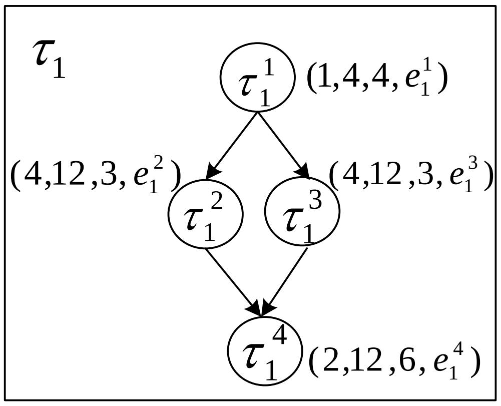
$$y_i^k = lcm\left\{\frac{c_i^{k\leftarrow v} \cdot y_i^v}{gcd(\rho_i^{k\leftarrow v} \cdot x_i^v, c_i^{k\leftarrow v})} | v \in \nu\right\},$$

$$x_i^k = y_i^k \cdot \frac{\rho_i^{k\leftarrow v}}{c_i^{k\leftarrow v}} \cdot \frac{x_i^v}{y_i^v}, \text{ where } \tau_i^v \in \nu.$$[3]

Proof. The proof of this lemma presented in [51] assumes the RBE task model while we use the RB task model. As discussed in Chapter 2, under the RBE task model, an execution rate of task τ_i^k is specified by parameters x_i^k and y_i^k, where x_i^k is the number of executions expected to be requested in any interval of length y_i^k; contrastingly, in our RB task model, x_i^k is the maximum number of executions in an interval of length y_i^k. However, as Goddard assumes that the source node of a PGM graph executes according to a valid rate-based pattern (i.e., exactly x executions in any interval y), this difference becomes immaterial. This lemma thus continues to hold in our context. □

[2]Given the close connection between τ^{RB} and G, we can henceforth associate tokens, queue attributes, etc., with edges in every DAG-based RB task in τ^{RB}, just like in G.

[3]It is shown in [51] that Definition 5.1 and Lemma 5.1 ensure that all tasks in ν with valid execution rates have the same x/y value.



τ_1
τ_1^1 $(1,4,4,e_1^1)$
$(4,12,3,e_1^2)$ τ_1^2
τ_1^3 $(4,12,3,e_1^3)$
τ_1^4 $(2,12,6,e_1^4)$

Figure 5.3: RB counterpart of the PGM graph in Figure 2.7.

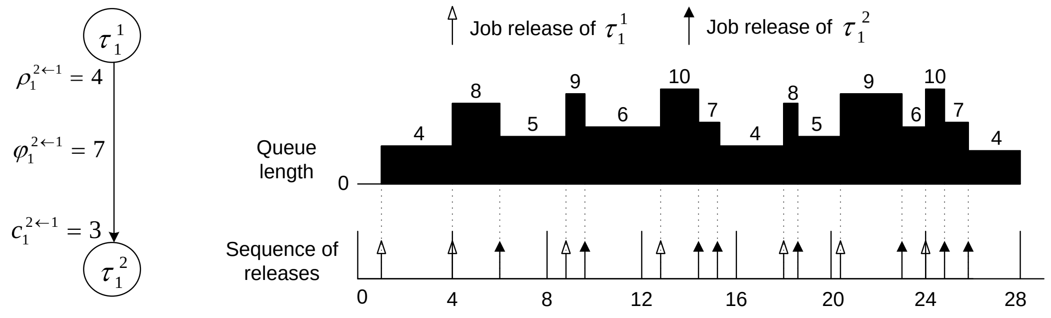


Figure 5.4: Extended snapshot sequence of releases.

By Lemma 5.1, we can compute an execution rate (x_i^k, y_i^k) for every task τ_i^k in τ^{RB}. Thus, each such task τ_i^k can be specified by parameters $(x_i^k, y_i^k, d_i^k, e_i^k)$, where

$$d_i^k = y_i^k / x_i^k. \tag{5.1}$$

Example. Consider again the example PGM graph shown in Figure 2.7. Figure 5.3 shows the corresponding DAG-based RB task system. The rate of each task is computed according to Lemma 5.1, assuming that the source node G_1^1 has an execution rate of $(1, 4)$. For instance, $y_1^2 = \dfrac{3 \cdot 4}{gcd(4 \cdot 1, 3)} = 12$ and $x_1^2 = 12 \cdot \dfrac{4 \cdot 1}{3 \cdot 4} = 4$. Also, $y_1^4 = lcm\left\{\dfrac{2 \cdot 12}{gcd(4,2)}, \dfrac{4 \cdot 12}{gcd(8,4)}\right\} = lcm\{12, 12\} = 12$ and $x_1^4 = 12 \cdot \dfrac{1}{2} \cdot \dfrac{4}{12} = 2$. Figure 5.4 shows an extended snapshot sequence showing releases of nodes τ_1^1 and τ_1^2. As seen, τ_1^2 releases at most four jobs within any interval $[j \cdot 12, (j+1) \cdot 12)$ $(j \geq 0)$.

In order to completely define τ^{RB}, we need to determine job precedence constraints in τ^{RB}. This is dealt with in the following lemma. In τ^{RB}, any job $\tau_{i,j}^{k}$'s predecessor jobs are those that need to complete execution in order for $\tau_{i,j}^{k}$ to be eligible to execute.

Lemma 5.2. *For any $\tau_i^w \in pred(\tau_i^k)$, $\tau_{i,v}^w$ is a predecessor job of $\tau_{i,j}^k$ iff $v = \left\lceil \dfrac{(j-1)\cdot c_i^{k\leftarrow w} + \varphi_i^{k\leftarrow w}}{\rho_i^{k\leftarrow w}} \right\rceil$.*

Proof. If $\tau_{i,v}^w$ is a predecessor job of $\tau_{i,j}^k$, then when $\tau_{i,v}^w$ completes, the number of tokens in the queue between τ_i^w and τ_i^k should be at least $\varphi_i^{k\leftarrow w}$ in order for $\tau_{i,j}^k$ to be able to execute. That is, $v \cdot \rho_i^{k\leftarrow w} - (j-1)\cdot c_i^{k\leftarrow w} \geq \varphi_i^{k\leftarrow w}$ must hold. By rearrangement, we have $v \geq \dfrac{(j-1)\cdot c_i^{k\leftarrow w} + \varphi_i^{k\leftarrow w}}{\rho_i^{k\leftarrow w}}$. Since v must be an integer, we have $v = \left\lceil \dfrac{(j-1)\cdot c_i^{k\leftarrow w} + \varphi_i^{k\leftarrow w}}{\rho_i^{k\leftarrow w}} \right\rceil$. □

Example. Consider the example shown in Figure 5.4. We can define job precedence constraints according to Lemma 5.2. For instance, the predecessor job of $\tau_{1,3}^2$ is $\tau_{1,v}^1$, where $v = \left\lceil \dfrac{2\cdot 3+7}{4} \right\rceil = 4$, and the predecessor job of $\tau_{1,4}^2$ is also $\tau_{1,4}^1$ because $\left\lceil \dfrac{3\cdot 3+7}{4} \right\rceil = 4$.

5.1.2 Transforming τ^{RB} to τ

We now show that τ^{RB} can be transformed to an ordinary sporadic system τ without capacity loss. The transformation process ensures that all precedence constraints in τ^{RB} are met. Later, in Section 5.1.3, we show that this process ensures that tardiness is bounded for τ^{RB} when GEDF is used.

We transform τ^{RB} to τ by redefining job releases appropriately. First, we must eliminate precedence constraints among tasks within the same DAG. We can do this by redefining job releases so that such constraints are automatically satisfied. By doing so, a DAG can be transformed into a set of independent RB tasks. Second, an RB task may release jobs arbitrarily close together, which is disallowed in the sporadic task model. Therefore, in order to transform τ^{RB} to τ, we also need to re-define job release times to enforce a minimum inter-arrival time.

For any job $\tau_{l,j}^h$ where $j > 1$ and $h > 1$, its original release time, $r^{RB}(\tau_{l,j}^h)$, is redefined to be

$$\begin{aligned} r(\tau_{l,j}^h) \quad = \quad & max\big(r^{RB}(\tau_{l,j}^h), F_{max}(pred(\tau_{l,j}^h)), \\ & r(\tau_{l,j-1}^h) + d_l^h\big), \end{aligned} \tag{5.2}$$

where $F_{max}(pred(\tau_{l,j}^h))$ denotes the latest completion time among all predecessor jobs of $\tau_{l,j}^h$.

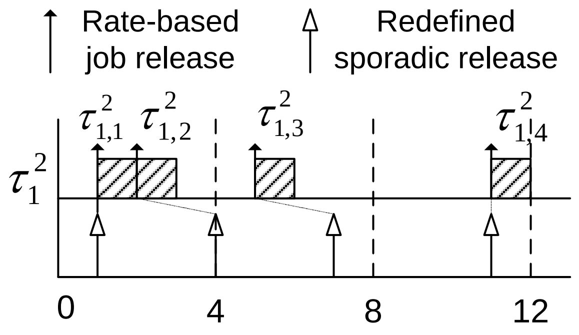


Figure 5.5: Redefining job releases according to (5.18) – (5.5).

Given that a source task has no predecessors, the release of any job $\tau_{l,j}^1$ ($j > 1$) of such a task is redefined to be

$$r(\tau_{l,j}^1) = max\big(r^{RB}(\tau_{l,j}^1), r(\tau_{l,j-1}^1) + d_l^1\big). \tag{5.3}$$

For the first job $\tau_{l,1}^h$ ($h > 1$) of any non-source task, its release time is redefined to be

$$r(\tau_{l,1}^h) = max\big(r^{RB}(\tau_{l,1}^h), F_{max}(pred(\tau_{l,1}^h))\big). \tag{5.4}$$

Finally, for the first job $\tau_{l,1}^1$ of any source task, its release time is redefined to be

$$r(\tau_{l,1}^1) = r^{RB}(\tau_{l,1}^1). \tag{5.5}$$

After redefining job releases according to (5.18)–(5.5), $\tau_{l,j}^h$'s redefined deadline, denoted $d(\tau_{l,j}^h)$, is given by

$$d(\tau_{l,j}^h) = r(\tau_{l,j}^h) + d_l^h. \tag{5.6}$$

Note that these definitions imply that each task's utilization remains unchanged.

Example. Consider the same example as shown in Figure 5.3. Figure 5.5 shows the redefined job releases of task τ_1^2. As seen, according to (5.18) and (5.20), $r(\tau_{1,1}^2) = r^{RB}(\tau_{1,1}^2)$, $r(\tau_{1,2}^2) = r(\tau_{1,1}^2) + d_1^2$, $r(\tau_{1,3}^2) = r(\tau_{1,2}^2) + d_1^2$, and $r(\tau_{1,4}^2) = r^{RB}(\tau_{1,4}^2)$.

Note that the release time of any job $\tau_{l,j}^h$ with predecessor jobs is redefined to be at least $F_{max}(pred(\tau_{l,j}^h))$. Thus, its start time is at least $F_{max}(pred(\tau_{l,j}^h))$. Hence, the GEDF schedule preserves the precedence constraints enforced by the DAG-based RB model. Note also that, since the release time of each $\tau_{l,j}^h$ ($j > 1$) is redefined to be at least that of $\tau_{l,j-1}^h$ plus d_l^h, τ_l executes as a sporadic task with $p_l^h = d_l^h$. By transforming every task in τ^{RB} to an independent sporadic task according to (5.18)–(5.5), we obtain τ. Note that (5.22) potentially causes job deadlines to move to later points in time.[4] The following lemma establishes an upper bound on the gap between the original deadline and the redefined deadline of any job of any source task. (Actually, it is not necessary to so aggressively shift *releases* to later points in time; this issue is dealt with in Section 5.1.4.)

Lemma 5.3. *For any job* $\tau_{i,j}^1$, $r(\tau_{i,j}^1) - r^{RB}(\tau_{i,j}^1) < 2 \cdot y_i^1$.

Proof. (All jobs considered in this proof are assumed to be jobs of τ_i^1.) It suffices to prove that for any job $\tau_{i,u}^1$ with $k \cdot y_i^1 \le r^{RB}(\tau_{i,u}^1) < (k+1) \cdot y_i^1$, where $k \ge -1$, we have $r(\tau_{i,u}^1) < k \cdot y_i^1 + 2 \cdot y_i^1$. We prove this by induction on k. For conciseness, we make the base case vacuous by starting with $k = -1$ (the base case then holds trivially since no job is released within time interval $[-y_i^1, 0)$).

For the induction step, let us assume that

$$r(\tau_{i,c}^1) < j \cdot y_i^1 + 2 \cdot y_i^1 \tag{5.7}$$

holds for any job $\tau_{i,c}^1$ with $j \cdot y_i^1 \le r^{RB}(\tau_{i,c}^1) < (j+1) \cdot y_i^1$ ($j \ge 0$). Then we want to prove that for any job $\tau_{i,v}^1$ with

$$(j+1) \cdot y_i^1 \le r^{RB}(\tau_{i,v}^1) < (j+2) \cdot y_i^1, \tag{5.8}$$

we have $r(\tau_{i,v}^1) < (j+1) \cdot y_i^1 + 2 \cdot y_i^1$.

Let χ denote the set of jobs with RB release times within $[(j+1) \cdot y_i^1, r^{RB}(\tau_{i,v}^1)]$. We consider three cases.

Case 1. χ is empty. In this case, $r^{RB}(\tau_{i,v}^1)$ is the first RB release within $[(j+1) \cdot y_i^1, (j+2) \cdot y_i^1)$. According to (5.19), either $r(\tau_{i,v}^1) = r^{RB}(\tau_{i,v}^1)$ or $r(\tau_{i,v}^1) = r(\tau_{i,v-1}^1) + d_i^1$ where $\tau_{i,v-1}^1$ is the last job of τ_i^1 with $j \cdot y_i^1 \le r^{RB}(\tau_{i,v-1}^1) < (j+1) \cdot y_i^1$. If $r(\tau_{i,v}^1) = r^{RB}(\tau_{i,v}^1)$, then $r(\tau_{i,v}^1) = r^{RB}(\tau_{i,v}^1) \overset{\{\text{by (5.8)}\}}{<}$

[4]Note that the method so far yields a non-work-conserving scheduler since the release time of some job may be delayed to a later point of time.

$j \cdot y_i^1 + 2 \cdot y_i^1$. If $r(\tau_{i,v}^1) = r(\tau_{i,v-1}^1) + d_i^1$, then by (5.7), we have $r(\tau_{i,v-1}^1) < j \cdot y_i^1 + 2 \cdot y_i^1$. Thus, $r(\tau_{i,v}^1) = r(\tau_{i,v-1}^1) + d_i^1 < j \cdot y_i^1 + 2 \cdot y_i^1 + d_i^1 \overset{\{\text{by } (5.1)\}}{\leq} (j+1) \cdot y_i^1 + 2 \cdot y_i^1$.

Case 2. χ is non-empty and there exists at least one job $\tau_{i,v'}^1$ in χ such that $r(\tau_{i,v'}^1) = r^{RB}(\tau_{i,v'}^1)$. According to (5.19) and the fact that at most x_i^1 jobs could be released within χ (since $[(j+1) \cdot y_i^1, r^{RB}(\tau_{i,v}^1)] \in [(j+1) \cdot y_i^1, (j+2) \cdot y_i^1)$), the release time of $\tau_{i,v}^1$ could be delayed by at most $(x_i^1 - 1) \cdot d_i^1$ time units. By (5.1), $(x_i^1 - 1) \cdot d_i^1 < y_i^1$. Thus, $r(\tau_{i,v}^1) < r^{RB}(\tau_{i,v}^1) + y_i^1 \overset{\{\text{by } (5.8)\}}{<} (j+2) \cdot y_i^1 + y_i^1 = (j+1) \cdot y_i^1 + 2 \cdot y_i^1$.

Case 3. χ is non-empty and there exists no job in χ such that $r(\tau_{i,v'}^1) = r^{RB}(\tau_{i,v'}^1)$. In this case, for the first-released job $\tau_{i,v'}^1$ in χ, we have $r(\tau_{i,v'}^1) = r(\tau_{i,v'-1}^1) + d_i^1$. Note that $\tau_{i,v'-1}^1$ exists, for otherwise, we would have $r(\tau_{i,v'}^1) = r^{RB}(\tau_{i,v'}^1)$. Due to the fact that at most $x_i^1 - 1$ jobs could be released within χ, we have $r(\tau_{i,v}^1) \leq r(\tau_{i,v'-1}^1) + x_i^1 \cdot d_i^1$. Thus, we have $r(\tau_{i,v}^1) \leq r(\tau_{i,v'-1}^1) + x_i^1 \cdot d_i^1 \overset{\{\text{by } (5.1) \text{ and } (5.7)\}}{<} j \cdot y_i^1 + 2 \cdot y_i^1 + y_i^1 = (j+1) \cdot y_i^1 + 2 \cdot y_i^1$. $\square$

5.1.3 Tardiness Bound for τ^{RB}

Given a DAG-based RB task system, τ^{RB}, by applying the strategy presented above, we obtain a task system τ containing only independent sporadic tasks. We can then apply the tardiness bound derived for ordinary sporadic task systems in [41] (or any other such bound), as stated below. Let $t_f(\tau_{l,j}^h)$ denote the completion time of $\tau_{l,j}^h$ in τ.

Definition 5.2. Let $\Delta = \dfrac{e_{sum} - e_{min}}{m - U_{m-1}} + e_l$, where e_{sum} is the sum of all tasks' WCET, e_{min} is the smallest WCET among all tasks, and U_{m-1} is the total utilization of $m-1$ tasks with the largest utilizations.

Theorem 5.1. *[41] In any GEDF schedule for the sporadic task system τ on m processors, if $U_{sum} \leq m$, then the tardiness of any job $\tau_{l,j}^h$, with respect to its redefined deadline $d(\tau_{l,j}^h)$, is at most δ, i.e.,*

$$t_f(\tau_{l,j}^h) - d(\tau_{l,j}^h) \leq \Delta. \tag{5.9}$$

However, Theorem 5.1 only gives a tardiness bound for any job $\tau_{l,j}^h$ with respect to its redefined deadline, $d(\tau_{l,j}^h)$. $\tau_{l,j}^h$ can have higher tardiness with respect to its original deadline, $d^{RB}(\tau_{l,j}^h)$.

Therefore, we must bound the actual tardiness any job $\tau_{l,j}^h$ may experience with respect to its original deadline. The following theorem gives such a bound. Let $y_l^{max} = max(y_l^1, y_l^2, ..., y_l^z)$, where z is the number of nodes in τ_l. Before stating the theorem, we first prove a lemma that is used in its proof.

Lemma 5.4. *For any two jobs $\tau_{l,j}^h$ and $\tau_{l,k}^h$ of τ_l^h in τ^{RB}, where $j < k$, $i \cdot y_l^h \leq r^{RB}(\tau_{l,j}^h) < (i+1) \cdot y_l^h$ ($i \geq 0$), and $(i+w) \cdot y_l^h \leq r^{RB}(\tau_{l,k}^h) < (i+w+1) \cdot y_l^h$ ($w \geq 0$), we have $r^{RB}(\tau_{l,k}^h) - r^{RB}(\tau_{l,j}^h) > (k-j) \cdot d_l^h - 2 \cdot y_l^h$.*

Proof. (All jobs considered in this proof are assumed to be jobs of τ_l^h.) Note that $k-j-1$ denotes the number of jobs other than $\tau_{l,j}^h$ and $\tau_{l,k}^h$ released in $[r^{RB}(\tau_{l,j}^h), r^{RB}(\tau_{l,k}^h)]$. Depending on the number of such jobs, we have two cases.

Case 1. $k-j-1 \leq 2 \cdot x_l^h - 2$. Given the case condition, $k-j \leq 2 \cdot x_l^h - 1$ holds. Thus, we have $(k-j) \cdot d_l^h - 2 \cdot y_l^h \leq (2 \cdot x_l^h - 1) \cdot d_l^h - 2 \cdot y_l^h \overset{\{\text{by } (5.1)\}}{<} 0$. Since $k > j$, $r^{RB}(\tau_{l,k}^h) - r^{RB}(\tau_{l,j}^h) \geq 0 > (k-j) \cdot d_l^h - 2 \cdot y_l^h$.

Case 2. $k-j-1 > 2 \cdot x_l^h - 2$. In this case, more than $2 \cdot (x_l^h - 1)$ jobs other than $\tau_{l,j}^h$ and $\tau_{l,k}^h$ are released in $[r^{RB}(\tau_{l,j}^h), r^{RB}(\tau_{l,k}^h)]$. By the statement of the lemma, at most x_l^h jobs can be released in $[r^{RB}(\tau_{l,j}^h), (i+1) \cdot y_l^h)$ or $[(i+w) \cdot y_l^h, r^{RB}(\tau_{l,k}^h)]$, respectively. Thus,

$$\lambda \geq (k-j-1) - 2 \cdot (x_l^h - 1), \tag{5.10}$$

where λ is the number of jobs other than $\tau_{l,j}^h$ and $\tau_{l,k}^h$ released in $[(i+1) \cdot y_l^h), (i+w) \cdot y_l^h)$.

Note that the length of the time interval $[(i+1) \cdot y_l^h), (i+w) \cdot y_l^h)$ really depends on the number of jobs released within this interval, due to that fact that at most x_l^h jobs are released within any interval $[(i+1) \cdot y_l^h), (i+2) \cdot y_l^h)$ of length y_l^h. For instance, if k jobs are released within $[(i+1) \cdot y_l^h), (i+w) \cdot y_l^h)$, where $1 \leq k < x_l^h$, then its length is at least y_l^h. If $x_l^h \leq k < 2x_l^h$ jobs are released within this interval, then its length is at least $2 \cdot y_l^h$. In general, by (5.10), the length of the time interval $[(i+1) \cdot y_l^h), (i+w) \cdot y_l^h)$ is $(i+w) \cdot y_l^h - (i+1) \cdot y_l^h \geq \left\lceil \frac{(k-j-1) - 2 \cdot (x_l^h - 1)}{x_l^h} \right\rceil \cdot y_l^h \geq \frac{(k-j-1) - 2 \cdot (x_l^h - 1)}{x_l^h} \cdot y_l^h = \frac{(k-j) - 2 \cdot x_l^h + 1}{x_l^h} \cdot y_l^h = (k-j) \cdot d_l^h - 2 \cdot y_l^h + d_l^h$.

Given (from the statement of the lemma) that $r^{RB}(\tau_{l,j}^h) < (i+1) \cdot y_l^h$ and $r^{RB}(\tau_{l,k}^h) \geq (i+w) \cdot y_l^h$, we have $r^{RB}(\tau_{l,k}^h) - r^{RB}(\tau_{l,j}^h) > (i+w) \cdot y_l^h - (i+1) \cdot y_l^h \geq (k-j) \cdot d_l^h - 2 \cdot y_l^h + d_l^h > (k-j) \cdot d_l^h - 2 \cdot y_l^h$. □

Theorem 5.2. *In any GEDF schedule for τ^{RB} on m processors, if $U_{sum} \leq m$, then the tardiness of any job $\tau_{l,j}^h$ of a task τ_l^h at depth k, with respect to its original deadline, $d^{RB}(\tau_{l,j}^h)$, is at most $(k+1) \cdot \Delta + 3(k+1) \cdot y_l^{max}$, i.e.,*

$$t_f(\tau_{l,j}^h) - d^{RB}(\tau_{l,j}^h) \leq (k+1) \cdot \Delta + 3(k+1) \cdot y_l^{max}. \tag{5.11}$$

Proof. This theorem can be proved by induction on task depth. In the base case, by Theorem 5.1 and the fact that τ_i^1 has no predecessors, its tardiness with respect to its newly-defined deadline, $d(\tau_{i,j}^1)$, is at most Δ. By Lemma 5.3, $r(\tau_{i,j}^1) - r^{RB}(\tau_{i,j}^1) < 2 \cdot y_i^1$. Thus, with respect to its original deadline, $d^{RB}(\tau_{i,j}^1)$, $\tau_{i,j}^1$ has a tardiness bound of $\Delta + 2 \cdot y_i^1 < \Delta + 3 \cdot y_i^{max}$.

For the induction step, let us assume (5.11) holds for any task τ_i^w at depth at most $k-1$, $k \geq 1$. Then, the tardiness of any job $\tau_{i,v}^w$ of τ_i^w is at most $k \cdot \Delta + 3k \cdot y_i^{max}$, i.e.,

$$t_f(\tau_{i,v}^w) - d^{RB}(\tau_{i,v}^w) \leq k \cdot \Delta + 3k \cdot y_i^{max}. \tag{5.12}$$

We want to prove that for any job $\tau_{i,j}^h$ of any task τ_i^h at depth k, $t_f(\tau_{i,j}^h) - d^{RB}(\tau_{i,j}^h) \leq (k+1) \cdot \Delta + 3(k+1) \cdot y_i^{max}$. According to (5.18) and (5.20), there are three cases to consider regarding $\tau_{i,j}^h$'s newly-defined release time $r(\tau_{l,j}^h)$, as illustrated in Figure 5.6.

Case 1. $r(\tau_{i,j}^h) = r^{RB}(\tau_{i,j}^h)$. By Theorem 5.1, we know that $t_f(\tau_{i,j}^h) - d(\tau_{i,j}^h) \leq \Delta$. Given that $d(\tau_{i,j}^h) = d^{RB}(\tau_{i,j}^h)$, we have $t_f(\tau_{i,j}^h) - d^{RB}(\tau_{i,j}^h) \leq \Delta < (k+1) \cdot \Delta + 3(k+1) \cdot y_i^{max}$.

Case 2. $r(\tau_{i,j}^h) = F_{max}(pred(\tau_{i,j}^h))$. Let $\tau_{i,v}^w$ be the predecessor of $\tau_{l,j}^h$ that has the latest completion time among all predecessors of $\tau_{i,j}^h$ ($\tau_{i,v}^w$ exists because the depth of τ_l^h is at least one). Thus, we have

$$r(\tau_{i,j}^h) = F_{max}(pred(\tau_{i,j}^h)) = t_f(\tau_{i,v}^w). \tag{5.13}$$

Therefore,

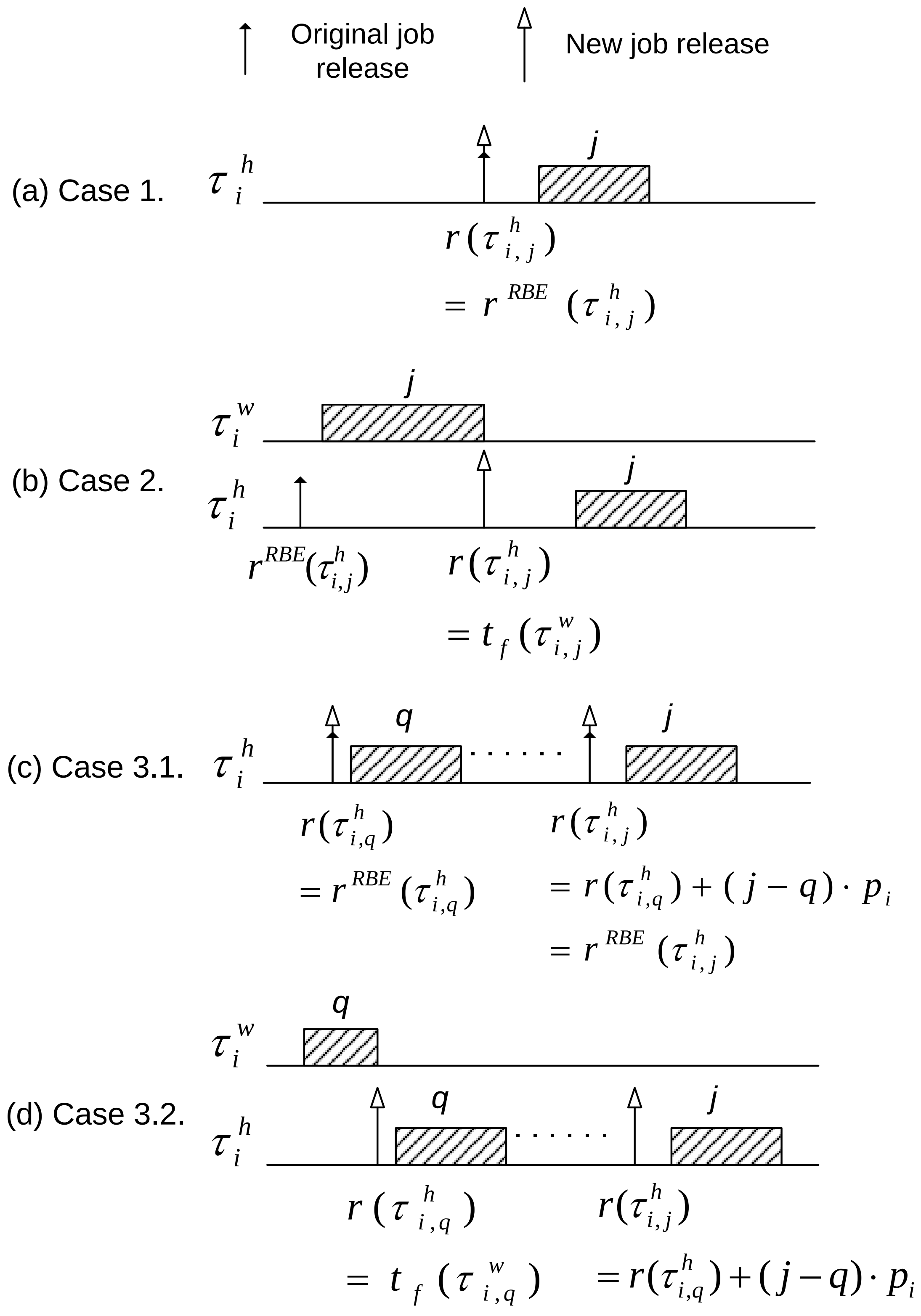


Figure 5.6: Three cases in Theorem 5.5.

$$
\begin{aligned}
& t_f(\tau^h_{i,j}) - d^{RB}(\tau^h_{i,j}) \\
& \quad \{\text{by } (2.1)\} \\
= \; & t_f(\tau^h_{i,j}) - r^{RB}(\tau^h_{i,j}) - d^h_i \\
= \; & t_f(\tau^h_{i,j}) - r(\tau^h_{i,j}) + r(\tau^h_{i,j}) - r^{RB}(\tau^h_{i,j}) - d^h_i \\
& \quad \{\text{by } (5.22)\} \\
= \; & t_f(\tau^h_{i,j}) - d(\tau^h_{i,j}) + d^h_i + r(\tau^h_{i,j}) - r^{RB}(\tau^h_{i,j}) - d^h_i \\
& \quad \{\text{by } (5.9)\} \\
\leq \; & \Delta + d^h_i + r(\tau^h_{i,j}) - r^{RB}(\tau^h_{i,j}) - d^h_i \\
= \; & \Delta + r(\tau^h_{i,j}) - r^{RB}(\tau^h_{i,j}) \\
& \quad \{\text{by } (2.2) \text{ and } (5.26)\} \\
\leq \; & \Delta + t_f(\tau^w_{i,v}) - r^{RB}(\tau^w_{i,v}) \\
& \quad \{\text{by } (2.1)\} \\
= \; & \Delta + t_f(\tau^w_{i,v}) - d^{RB}(\tau^w_{i,v}) + d^w_i \\
& \quad \{\text{by } (5.25)\} \\
\leq \; & \Delta + k \cdot \Delta + 3k \cdot y^{max}_i + d^w_i \\
& \quad \{\text{by } (5.1)\} \\
< \; & (k+1) \cdot \Delta + 3(k+1) \cdot y^{max}_i.
\end{aligned}
$$

Case 3. $j > 1 \wedge r(\tau^h_{i,j}) = r(\tau^h_{i,j-1}) + d^h_i$. Let $\tau^h_{i,q}$ ($q < j$) denote the last job of τ^h_i released before $\tau^h_{i,j}$ such that $r(\tau^h_{i,q}) = r^{RB}(\tau^h_{i,q})$ or $r(\tau^h_{i,q}) = F_{max}(pred(\tau^h_{i,q}))$. $\tau^h_{i,q}$ exists because according to (5.20) and (5.5), there exists at least one job, $\tau^h_{i,1}$, such that $r(\tau^h_{i,1}) = r^{RB}(\tau^h_{i,1})$ or $r(\tau^h_{i,1}) = F_{max}(pred(\tau^h_{i,1}))$. Depending on the value of $r(\tau^h_{i,q})$, we have two subcases.

Case 3.1. $r(\tau^h_{i,q}) = r^{RB}(\tau^h_{i,q})$. By the definition of $\tau^h_{i,q}$, the release time of any job $\tau^h_{i,k}$, where $q < k \leq j$, is redefined to be $r(\tau^h_{i,k}) = r(\tau^h_{i,k-1}) + d^h_i$. Thus, we have

$$r(\tau^h_{i,j}) = r(\tau^h_{i,q}) + (j - q) \cdot d^h_i. \tag{5.14}$$

Therefore, we have

$$t_f(\tau^h_{i,j}) - d^{RB}(\tau^h_{i,j})$$

$$
\begin{aligned}
& \{\text{by (2.1)}\} \\
= \quad & t_f(\tau_{i,j}^h) - r^{RB}(\tau_{i,j}^h) - d_i^h \\
= \quad & t_f(\tau_{i,j}^h) - r(\tau_{i,j}^h) + r(\tau_{i,j}^h) - r^{RB}(\tau_{i,j}^h) - d_i^h \\
& \{\text{by (5.22)}\} \\
= \quad & t_f(\tau_{i,j}^h) - d(\tau_{i,j}^h) + d_i^h + r(\tau_{i,j}^h) - r^{RB}(\tau_{i,j}^h) - d_i^h \\
& \{\text{by (5.9)}\} \\
\leq \quad & \Delta + d_i^h + r(\tau_{i,j}^h) - r^{RB}(\tau_{i,j}^h) - d_i^h \\
= \quad & \Delta + r(\tau_{i,j}^h) - r^{RB}(\tau_{i,j}^h) \\
& \{\text{by (5.27) and Lemma 5.4}\} \\
< \quad & \Delta + (r(\tau_{i,q}^h) + (j-q) \cdot d_i^h) - (r^{RB}(\tau_{i,q}^h) \\
& +(j-q) \cdot d_i^h - 2 \cdot y_i^h) \\
& \{\text{by the case condition}\} \\
= \quad & \Delta + 2 \cdot y_i^h \\
< \quad & (k+1) \cdot \Delta + 3(k+1) \cdot y_i^{max}.
\end{aligned}
$$

Case 3.2. $r(\tau_{i,q}^h) = F_{max}(pred(\tau_{i,q}^h))$. Let $\tau_{i,v}^w$ denote a predecessor job of $\tau_{i,q}^h$ with $t_f(\tau_{i,v}^w) = F_{max}(pred(\tau_{i,q}^h)) = r(\tau_{i,q}^h)$. We have

$$
\begin{aligned}
& t_f(\tau_{i,j}^h) - d^{RB}(\tau_{i,j}^h) \\
& \{\text{similarly to the derivation in Case 3.1}\} \\
< \quad & \Delta + (r(\tau_{i,q}^h) + (j-q) \cdot d_i^h) - (r^{RB}(\tau_{i,q}^h) \\
& +(j-q) \cdot d_i^h - 2 \cdot y_i^h) \\
= \quad & \Delta + r(\tau_{i,q}^h) - r^{RB}(\tau_{i,q}^h) + 2 \cdot y_i^h \\
& \{\text{by the case condition and (2.2)}\} \\
\leq \quad & \Delta + t_f(\tau_{i,v}^w) - r^{RB}(\tau_{i,v}^w) + 2 \cdot y_i^h \\
& \{\text{by (2.1)}\} \\
= \quad & \Delta + t_f(\tau_{i,v}^w) - d^{RB}(\tau_{i,v}^w) + d_i^w + 2 \cdot y_i^h \\
& \{\text{by (5.25)}\} \\
\leq \quad & \Delta + k \cdot \Delta + 3k \cdot y_i^{max} + d_i^w + 2 \cdot y_i^h \\
& \{\text{by (5.1)}\} \\
\leq \quad & (k+1) \cdot \Delta + 3(k+1) \cdot y_i^{max}. \qquad \square
\end{aligned}
$$

5.1.4 Improving Job Response Times by Early-Releasing

By forcing RB releases to be sporadic, we essentially delay job releases. However, excessive release delays are actually unnecessary and actual response times can be improved by applying a technique called "early-releasing," which allows jobs to execute before their specified release times. The earliest time at which job $\tau_{l,j}^{h}$ may execute is defined by its *early-release time* $\varepsilon(\tau_{l,j}^{h})$, where $\varepsilon(\tau_{l,j}^{h}) \le r(\tau_{l,j}^{h})$. For any job $\tau_{l,j}^{h}$, its early-releasing time can be defined as

$$\varepsilon(\tau_{l,j}^{h}) = \begin{cases} r^{RB}(\tau_{l,j}^{h}) & \text{if } h = 1 \\ F_{max}(pred(\tau_{l,j}^{h})) & \text{if } h > 1. \end{cases}$$

An unfinished job $\tau_{l,j}^{h}$ is *eligible* for execution at time t if $\tau_{l,j-1}^{h}$ has completed by t (if $j > 1$) and $t \ge \varepsilon(\tau_{l,j}^{h})$. The tardiness bound given in Theorem 5.1 continues to hold if early-releasing is allowed [39]. Intuitively, this is reflective of the fact that schedulability mainly hinges on the proper spacing of consecutive job *deadlines* of a task, instead of its *releases*. This same intuition underlies the development of the uniprocessor RBE model [61].

Example. Consider a DAG-based RB task system scheduled on two processors under GEDF consisting of two tasks: $\tau_1^1(1,4,4,2)$ and $\tau_1^2(1,4,4,2)$, where $pred(\tau_1^2) = \tau_1^1$ and $pred(\tau_{2,j}^2) = \tau_{1,j}^1$ for any $j > 0$. Figure 5.7(a) shows the original RB releases before time 12. Figure 5.7(b) shows the redefined releases according to (5.18)–(5.5), as well as the GEDF schedule. As seen in the schedule, $\tau_{1,2}^2$ completes at time 10 and misses its original deadline, which is at time 8. Figure 5.7(c) shows early releases as defined above and the corresponding GEDF schedule. As seen, most jobs' response times are improved. For instance, $\tau_{1,2}^2$ now completes at time 8 and meets its original deadline.

5.1.5 Case Study

In this section, we present a case study that demonstrates the utility of our results. We study a DAG-based system with two sporadic DAGs, as shown in Fig 5.8. Each sporadic DAG contains a number of sporadic tasks, with a common period of 10*ms*. Node labels in Figure 5.8 give the execution cost (in *ms*) of each task. The total utilization of this system is 3.0. Prior to our work,

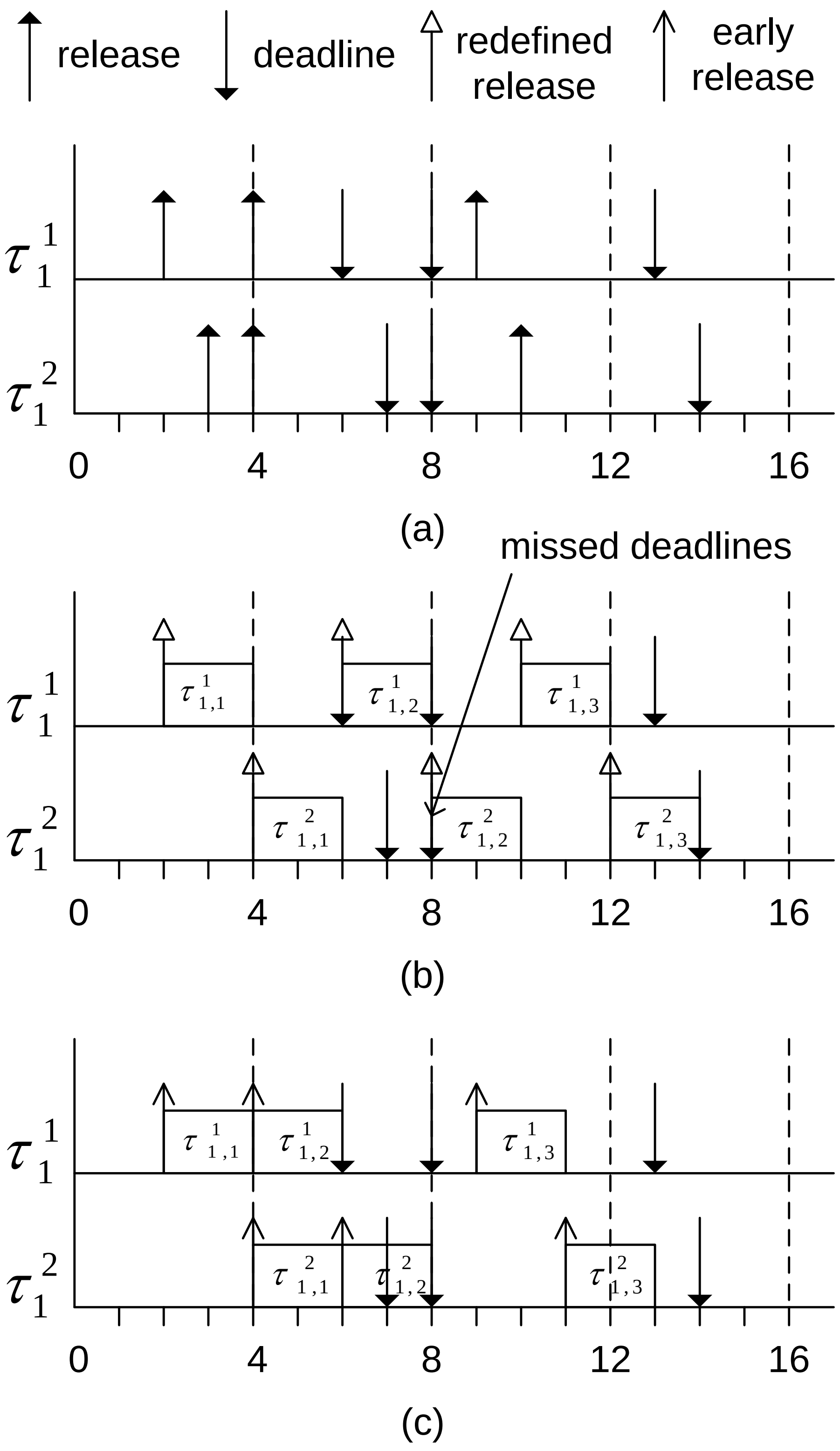


Figure 5.7: Early-releasing example.

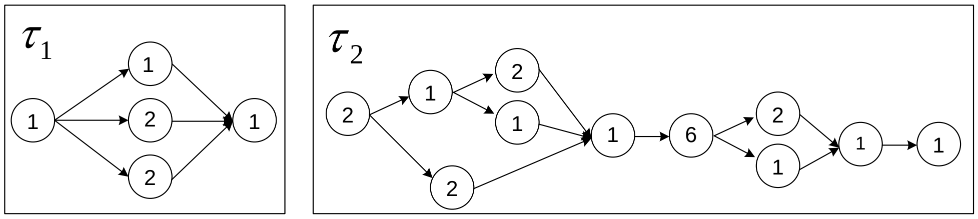


Figure 5.8: Case study.

	Previous approach	Our approach
Processors needed	**6**	**3**
Max. bound of τ_1	**0**	**102ms**
Max. bound of τ_2	**0**	**272 ms**
Max. observed tardiness	**0**	**10 ms**

Table 5.2: Case study results.

the only existing approach that could be applied to successfully schedule a DAG-based system under GEDF on multiprocessors is to ensure that every job meets its deadline, so that DAG-based precedence constraints can automatically be satisfied (as seen in Figure 5.1).

Table 5.2 shows a comparison of our approach and this previous approach, where in the latter case, a GEDF schedulability test by Baker [7] was used to ensure that deadlines are not missed. In this table, the number of required processors, theoretical tardiness bounds (computed using Theorems 5.1 and 5.5), and the maximum observed tardiness are shown. The latter was determined by simulating the schedule until it repeats. As seen, although our approach requires a higher tardiness bound, we only need three processors to correctly schedule this system under GEDF with bounded tardiness, which is only half of the processors required by the other approach. Moreover, the maximum observed tardiness arising under our approach is low.

5.1.6 Summary

We have shown in this section that DAG-based systems with sophisticated notions of acyclic precedence constraints can be supported under GEDF on multiprocessors with no capacity loss provided bounded deadline tardiness is acceptable. Our results also imply that any global scheduling algorithm that can ensure bounded tardiness with no capacity loss for ordinary sporadic task systems can ensure the same for any DAG-based task system. Our results are general enough to be applicable to periodic/sporadic DAG-based systems as well.

Our overall scheduling strategy is similar to that presented previously by Goddard for the uniprocessor case [51]. However, several differences do exist. First, under the RBE task model used in [51], a rate is specified by parameters x and y, where x is the number of executions expected to be requested in any interval of length y; contrastingly, in our RB task model, x is the maximum number of executions in an interval of length y. However, Goddard assumes that the source node of a PGM graph executes according to a deterministic rate-based pattern (i.e., exactly x executions in any interval y), so this difference becomes immaterial. Second, the RBE task model specifies a minimum separation between consecutive job deadlines of the same task. Although our RB task model does not require such a minimum separation, we ultimately redefine releases and deadlines to enforce a minimum separation (Section 5.1.2), and early-release jobs to obtain a work-conserving scheduler (Section 5.1.4). Third, in the uniprocessor case, DAG-based precedence constraints can be more easily eliminated than in the multiprocessor case. In [51], DAG-based precedence constraints are met by keeping the job ready queue in EDF order and breaking deadline ties on a FIFO basis. We instead redefine job releases to eliminate all DAG-based precedence constraints (see (5.18)–(5.5)). Finally, after mapping PGM graphs to rate-based tasks, Goddard derives a schedulability condition for the resulting system. We instead further transform rate-based tasks into sporadic tasks and derive a schedulability condition for the sporadic task system. Despite these differences, our approach can be seen as a multiprocessor counterpart of that in [51] for scheduling acyclic PGM graphs.

5.2 Scheduling SRT PGM in a Distributed System

In the previous section, we showed that SRT PGM task systems can be supported on a globally-scheduled multiprocessor with no capacity loss. In this section, this result is extended to be applicable to distributed systems, which consist a collection of clusters of processors.

In work on real-time scheduling in distributed systems, task models where no inter-task precedence constraints exist, such as the periodic and the sporadic task models, have received much attention. However, with the growing prevalence of multicore platforms, it is inevitable that such DAG-based real-time applications will be deployed in distributed systems where multicore machines are used as per-node computers. One emerging example application where such a deployment is expected is fractionated satellite systems [34]. Such a system consists of a number of wirelessly-connected small satellites, each of which may be controlled by a multicore machine. The overall collection of such machines is expected to support DAG-based real-time workloads such as radar and signal-processing subsystems. To support such workloads, efficient scheduling algorithms are needed. Motivated by applications such as this, we show in this section how to support real-time DAG-based applications in distributed systems.

We view a distributed system as a collection of clusters of processors, where all processors in a cluster are locally connected (e.g., on a multicore machine). A DAG-based task system can be deployed in such a setting by (*i*) assigning tasks to clusters, and (*ii*) determining how to schedule the tasks in each cluster. In addressing (*i*), overheads due to data communications among connected tasks must be considered since tasks within the same DAG may be assigned to different clusters. In addressing (*ii*), any employed scheduling algorithm should seek to minimize capacity loss.

To the best of our knowledge, in all prior work on supporting DAG-based applications in systems with multiple processors (multiprocessors or distributed systems), either global or partitioned scheduling has been assumed. As discussed in Section 2.3, if bounded deadline tardiness is the timing constraint of interest, then global approaches can often be applied on multiprocessor platforms with no loss of processing capacity [39, 76]. However, the virtues of global scheduling come at the expense of higher runtime overheads. In work on ordinary sporadic (not DAG-based) task systems, clustered scheduling, which combines the advantages of both global and partitioned scheduling, has been suggested as a compromise [9, 27]. Under clustered scheduling, tasks are first partitioned onto clusters of cores, and intra-cluster scheduling is global.

In distributed systems, clustered scheduling algorithms are a natural choice, given the physical layout of such a system. Thus, such algorithms are our focus here. Our specific objective is to develop clustered scheduling techniques and analysis that can be applied to support DAGs, assuming that bounded deadline tardiness is the timing guarantee that must be ensured. Our primary motivation is to develop such techniques for use in distributed systems, where different clusters are physically separated; however, our results are also applicable in settings where clusters are tightly coupled (e.g., each cluster could be a socket in a multi-socket system). Our results can be applied to systems with rather sophisticated precedence constraints as specified by PGM. In a distributed system, it may be necessary to transfer data from a producer in one cluster to a consumer in another through an inter-cluster network, which could cause a significant amount of data communication overhead. Thus, any proposed scheduling algorithm should seek to minimize such inter-cluster data communication.

In the previous section, we extended Goddard's work and showed that GEDF can ensure bounded deadline tardiness in general DAG-based systems with no capacity loss on multiprocessors. In the rest of this section, we show that sophisticated notions of acyclic precedence constraints can also be efficiently supported under clustered scheduling in a distributed system, provided bounded deadline tardiness is acceptable. Specifically, we propose a clustered scheduling algorithm called *CDAG* that first partitions PGM graphs onto clusters, and then uses global scheduling approaches within each cluster. We present analysis that gives conditions under which each task's maximum tardiness is bounded. The conditions derived for CDAG show that the resulting capacity loss (even in the worst-case) is small. To assess the effectiveness of CDAG in reducing inter-cluster data communications, we compare it with an optimal integer linear programming (ILP) solution that minimizes inter-cluster data communications when partitioning PGM graphs onto clusters. We assume that tasks are specified using a rate-based task model that generalizes the periodic and sporadic task models. We next describes our system model.

5.2.1 System Model

In this section, we describe the assumed system architecture. A detailed description of the PGM and the DAG-based RB task model can be found in Section 2.1.6.

We consider the problem of scheduling a set of n acyclic PGM graphs $\{\tau_1, \tau_2, ..., \tau_n\}$ on φ clusters $C_1, C_2, ..., C_\varphi$. Each cluster C_i contains λ_i processors. Clusters are connected by a network. Let B denote the minimum number of data units that can be transferred between any two clusters per time unit. Similarly, let b denote the minimum number of data units that can be transferred between any two processors within the same cluster per time unit. If the system is a distributed collection of multicore machines, then B is impacted by the speed and bandwidth of the communication network and b is impacted by the speed of the data transfer bus on a multicore chip. In this case, $b \gg B$, as local on-chip data communication is generally much faster than communication across a network.

Example. Figure 5.9(a) shows an example PGM graph system consisting of two graphs τ_1 and τ_2 where τ_1 contains four nodes with four edges and τ_2 contains two nodes with one edge. We will use this example to illustrate other concepts throughout this section. Figure 5.9(b) shows the rate-based counterpart of the PGM graphs in Figure 5.9(a) (the transformation is discussed in detail in Section 5.1). An example distributed system containing two clusters each with two processors interconnected by a network is shown in Figure 5.9(c).

For conciseness, we remove the RB superscript in job-related notation as seen in Equations (2.1) and (2.2). That is, we denote job $\tau_{l,j}^h$'s release time as $r_{l,j}^h$ and its (absolute) deadline as

$$d_{l,j}^h = r_{l,j}^h + d_l^h, \tag{5.15}$$

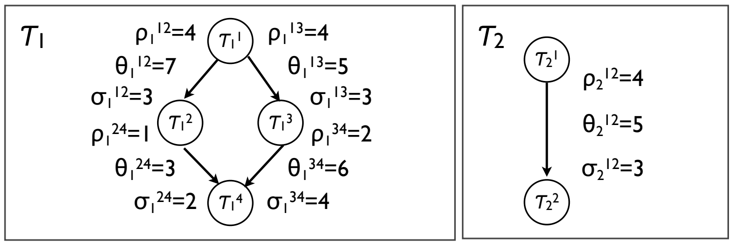
and we have

$$r_{l,j}^h \geq r_{i,v}^w. \tag{5.16}$$

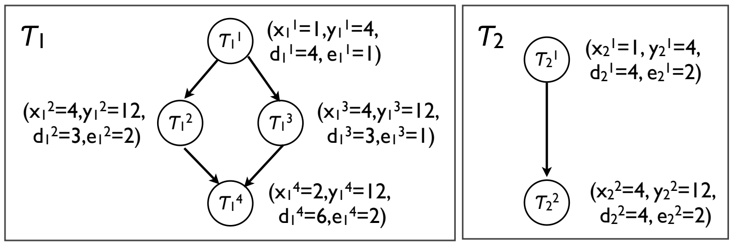
5.2.2 Algorithm CDAG

In this section, we propose Algorithm CDAG, a clustered-scheduling algorithm that ensures bounded tardiness for DAG-based systems on distributed clusters. Since inter-cluster data communication can be expensive, CDAG is designed to reduce such communication.

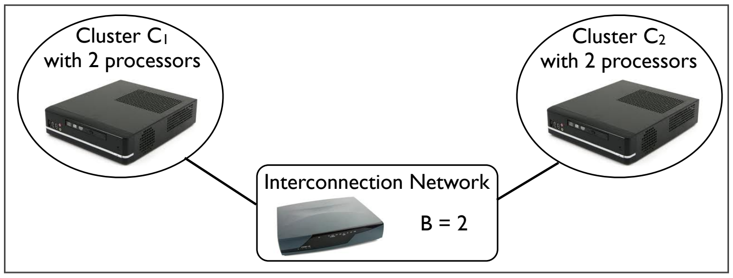
CDAG consists of two phases: an *assignment phase* and an *execution phase*. The assignment phase executes offline and assigns each PGM graph to one or more clusters. In the execution phase, PGM graphs are first transformed to ordinary sporadic tasks (where no precedence constraints arise), and then scheduled under a proposed clustered scheduling algorithm.



(a) Example PGM graph system


τ_1
τ_1^1
$(x_1^1=1, y_1^1=4, d_1^1=4, e_1^1=1)$
τ_1^2
$(x_1^2=4, y_1^2=12, d_1^2=3, e_1^2=2)$
τ_1^3
$(x_1^3=4, y_1^3=12, d_1^3=3, e_1^3=1)$
τ_1^4
$(x_1^4=2, y_1^4=12, d_1^4=6, e_1^4=2)$
τ_2
τ_2^1
$(x_2^1=1, y_2^1=4, d_2^1=4, e_2^1=2)$
τ_2^2
$(x_2^2=4, y_2^2=12, d_2^2=4, e_2^2=2)$

(b) Rate-based counterpart of (a)


Cluster C_1 with 2 processors
Cluster C_2 with 2 processors
Interconnection Network
B = 2

(c) An example distributed system

Figure 5.9: Example system used throughout this section.

5.2.3 Assignment Phase

The assignment phase assigns acyclic PGM graphs (or DAGs for short) to clusters in a way such that the inter-cluster data communication cost is reduced. Note that the total utilization of the DAGs (or portions of DAGs) assigned to a cluster must not exceed the total capacity of that cluster. CDAG contains an assignment algorithm that is designed to partition DAGs onto clusters such that both the inter-cluster data communication cost and any bin-packing-related capacity loss are minimized.

To provide a better understanding of the problem of partitioning DAGs onto clusters with minimum inter-cluster data communication cost, we first formulate it as an ILP, which provides an optimal solution. (Note that, due to potential capacity loss arising from partitioning DAGs, the ILP approach may not find a feasible assignment. However, if the ILP approach cannot find a feasible solution, then the given task set cannot be assigned to clusters under any partitioning algorithm.)

Definition 5.3. For any edge e_i^{jk} of DAG τ_i, its *edge data weight*, w_i^{jk}, is defined to be $\rho_i^{jk} \cdot \dfrac{x_i^j}{y_i^j}$. A larger edge data weight indicates a larger data communication cost between tasks connected by the corresponding edge. For any edge e_i^{jk} of DAG τ_i , its *communication cost* ϖ_i^{jk} is defined to be w_i^{jk} if the corresponding connected nodes τ_i^j and τ_i^k are assigned to different clusters, and 0 otherwise.

Note that the above definition does not consider data consuming rates. This is because data is stored in memory local to the consuming node, as discussed in Section 2.3.3. Thus, only produced data needs to be transferred.

Example. For task τ_1^1 of DAG τ_1 in Figure 5.9, we can use previous techniques presented in Section 5.1 to calculate its execution rates, which are $x_1^1 = 1$ and $y_1^1 = 4$. Thus, for edge e_1^{12} of τ_1, its edge data weight $w_1^{12} = \rho_1^{12} \cdot \frac{x_1^1}{y_1^1} = 4 \cdot \frac{1}{4} = 1$. Intuitively, node τ_1^1 produces one data unit on average on edge e_1^{12} per time unit, given its execution rate $(x_1^1, y_1^1) = (1, 4)$.

Definition 5.4. The *total communication cost* of any given DAG-based system τ, denoted ϖ_{sum}, is given by $\sum_{\tau_i \in \tau} \sum_{e_i^{jk} \in \tau_i} \varpi_i^{jk}$.

ILP formulation. We are given a set τ of tasks (each task corresponds to a node in a DAG) and a set ξ of clusters. To reduce clutter in the ILP formulation, we denote tasks more simply as $\tau_1, \tau_2, ...$, and let $w_{i,j}$ be the data weight of the edge connecting tasks τ_i and τ_j ($w_{i,j} = 0$ if i and j are not connected). (This simplified notation is used only for the ILP formulation.)

For all $\tau_i \in \tau$, let $x_{i,k}$ be a binary decision variable that equals 1 when task τ_i is assigned to cluster C_k, and 0 otherwise. For all $(\tau_i, \tau_j) \in \tau \times \tau$, let $y_{i,j}$ be a binary decision variable that equals 1 if tasks τ_i and τ_j are assigned to the same cluster, and 0 otherwise.

Our goal is to minimize the total communication cost. An ILP formulation of this optimization problem is then:

Minimize

$$\sum_{i \in \tau} \sum_{j \in \tau} w_{i,j} \cdot (1 - y_{i,j}) \tag{5.17}$$

subject to the constraints below. Note that by Definitions 5.3 and 5.4, (5.17) represents the total communication cost.

- Each task must be assigned to one cluster:

$$\textstyle\sum_{C_k \in \xi} x_{i,k} = 1, \forall \tau_i \in \tau.$$

- The total utilization of all tasks assigned to a cluster must not exceed the total capacity of that cluster:

$$\textstyle\sum_{\tau_i \in \tau} x_{i,k} \cdot u_i \leq \lambda_k, \forall C_k \in \xi.$$

- $y_{i,j}$ should be 1 when two tasks are assigned to the same cluster, and 0 otherwise:

$$y_{i,j} \leq x_{i,k} - x_{j,k} + 1, \forall (\tau_i, \tau_j) \in \tau \times \tau, \forall C_k \in \xi,$$

$$y_{i,j} \leq -x_{i,k} + x_{j,k} + 1, \forall (\tau_i, \tau_j) \in \tau \times \tau, \forall C_k \in \xi.$$

By solving the ILP above, we obtain an optimal assignment that gives the minimum total communication cost as long as there exists a feasible assignment.

Example. Consider assigning DAGs τ_1 and τ_2 in Figure 5.9 to two clusters. τ_1 has a utilization of $1/4 + 2/3 + 1/3 + 1/3 = 19/12$ and τ_2 has a utilization of $1/2 + 2/3 = 7/6$. By formulating this assignment problem as an ILP according to the above approach, an optimal solution is to assign all tasks of τ_1 to the first cluster and all tasks of τ_2 to the second cluster, which leads to $\varpi_{sum} = 0$.

A polynomial-time assignment algorithm. Although the ILP solution is optimal, it has exponential time complexity. We now propose a polynomial-time algorithm to assign DAGs to clusters. This algorithm tries to minimize the total communication cost, which is achieved by locally minimizing communication costs when assigning each DAG.

Definition 5.5. For any task τ_i^j of DAG τ_i, its *task data weight*, w_i^j, is defined to be $\sum_{\tau_i^j \rightarrow \tau_i^k} w_i^{jk}$, where $\tau_i^j \rightarrow \tau_i^k$ denotes that τ_i^j has an outgoing edge to τ_i^k.

Definition 5.6. For any DAG τ_i, its *average data weight*, w_i, is defined to be $\frac{\sum_{e_i^{jk} \in \tau_i} w_i^{jk}}{E_i}$, where E_i is the total number of edges in τ_i. A DAG τ_i's *data weight* is defined to be $\sum_{e_i^{jk} \in \tau_i} w_i^{jk}$.

Example. For DAG τ_1 in Figure 5.9, by Definition 5.3, $w_1^{12} = w_1^{13} = 1$, $w_1^{24} = 1/3$, and $w_1^{34} = 2/3$. Thus, by Definition 5.6, τ_1 has an average data weight of $\frac{3}{4}$. By Definition 5.5, task τ_1^1 of τ_1 has a data weight of $w_1^1 = w_1^{12} + w_1^{13} = 2$. Intuitively, node τ_1^1 produces two data units on average on its outgoing edges per time unit, given its execution rate $(x_1^1, y_1^1) = (1, 4)$.

The proposed DAG assignment algorithm, denoted *ASSIGN*, is shown in Figure 5.10.

Algorithm description. Algorithm *ASSIGN* assigns DAGs to clusters in two phases. In the first phase (lines 1-7), it assigns DAGs in largest-average-data-weight-first order to clusters in smallest-capacity-first order, which gives a higher possibility for DAGs with larger average data weight to be fully assigned to a cluster. DAGs that cannot be assigned to clusters in the first phase are considered in the second phase (lines 8-15). For each unassigned DAG in order, its tasks are ordered by depth and then tasks at the same depth are ordered by data weight, which gives tasks with larger data weight a greater chance to be assigned to the same cluster as their predecessor tasks (lines 8-9). Then each task in order is assigned to clusters in largest-capacity-first order (lines 10-15). Task τ_i^k is assigned to cluster C_j if τ_i^k can receive its full share of its utilization from C_j (lines 13-14). If not, then C_j is excluded from being considered for any of the later tasks, and the next cluster in order will be considered for scheduling τ_i^k (line 15).

Example. Consider the example DAG system in Figure 5.9 to be partitioned under the proposed algorithm. By Definitions 5.3 and 5.4, τ_1 has an average data weight of $3/4$ and τ_2 has an average data weight of 1. Thus, τ_2 is ordered before τ_1. Since τ_2 has a utilization of $7/6$ and τ_1 has a utilization of $19/12$, the tasks of τ_2 are assigned to the first cluster, and the tasks of τ_1 are assigned to the second cluster, which leads to $\varpi_{sum} = 0$.

```
ASSIGN
$u(C_i)$: CAPACITY OF CLUSTER $C_i$, INITIALLY $u(C_i) = \lambda_i$
$\xi$: A LIST OF CLUSTERS $\{C_1, C_2, ..., C_\varphi\}$
$\zeta$: A LIST OF DAGS $\{\tau_1, \tau_2, ..., \tau_n\}$

PHASE 1:
 1  Order DAGs in $\zeta$ by largest average data weight first
 2  Order clusters in $\xi$ by smallest capacity first
 3  for each DAG $\tau_i$ in $\zeta$ in order
 4      for each cluster $C_j$ in $\xi$ in order
 5          if $u_i \leq u(C_j)$
 6              Assign all tasks of $\tau_i$ to $C_j$
 7              Remove $\tau_i$ from $\zeta$; $u(C_j) := u(C_j) - u_i$
PHASE 2:
 8  for each DAG $\tau_i$ in $\zeta$ in order
 9      Order tasks within $\tau_i$ by smallest depth first, then
        order tasks within $\tau_i$ and at the same depth by
        largest task data weight first
10  Order clusters in $\xi$ by largest capacity first
11  for each task $\tau_i^k$ of DAG $\tau_i$ in $\zeta$ in order
12      for each cluster $C_j$ in $\xi$ in order
13          if $u_i^k \leq u(C_j)$ then
14              Assign $\tau_i^k$ to $C_j$; $u(C_j) := u(C_j) - u_i^k$
15          else Remove $C_j$ from $\xi$
```

Figure 5.10: Psuedocode of the assignment algorithm.

Time complexity. The time complexity of Phase 1 of *ASSIGN* depends on (*i*) the sorting process (lines 1-2), which is $O(n \cdot logn + \varphi \cdot log\varphi)$, and (*ii*) the two **for** loops (lines 3-4), which is $O(N \cdot \varphi)$, where N is the number of tasks in the system (each task corresponds to a node in a DAG). Thus, Phase 1 has a time complexity of $O(n \cdot logn + \varphi \cdot log\varphi + N \cdot \varphi)$. The time complexity of Phase 2 of *ASSIGN* depends on (*i*) the sorting process (lines 9-10), which is $O(n \cdot \mu \cdot log\mu + \varphi \cdot log\varphi)$, where μ is the maximum number of tasks per-DAG, and (*ii*) the two **for** loops (lines 11-12), which is $O(N \cdot \varphi)$. Thus, Phase 2 has a time complexity of $O(n \cdot \mu \cdot log\mu + \varphi \cdot log\varphi + N \cdot \varphi)$. The overall time complexity of *ASSIGN* is thus $O(n \cdot logn + \varphi \cdot log\varphi + n \cdot \mu \cdot log\mu + N \cdot \varphi)$.

Partitioning condition. The following theorem gives a condition for *ASSIGN* to successfully partition any given DAG-based task system onto clusters. For conciseness, let us denote tasks (each task corresponds to a node in a DAG) after ordering by $\tau_1, \tau_2, ..., \tau_N$ (note that tasks within DAGs that are fully assigned to clusters in the first phase are assumed to be ordered before all other tasks here). Let $u(\tau_i)$ denote the utilization of task τ_i under this notation. Before stating the theorem, we first prove the following lemma.

Lemma 5.5. *Under Algorithm ASSIGN, if a task τ_i is the first task that cannot be assigned to any cluster, then $\sum_{k=1}^{i} u(\tau_k) > m - u_{\varphi-1}$, where $u_{\varphi-1}$ is the sum of $\varphi - 1$ largest task utilizations.*

Proof. Due to the fact that some task τ_i cannot be assigned to any cluster, the second phase of Algorithm *ASSIGN* is executed. In the second phase, if Algorithm *ASSIGN* fails to assign the i^{th} task τ_i to any cluster, then the last cluster C_φ does not have enough capacity to accommodate τ_i. Moreover, for each previous cluster C_j, where $j \leq \varphi - 1$, there exists a task, denoted T^j, that could not be assigned to C_j, and thus the next cluster in order was considered to accommodate T^j and C_j was removed from being considered again for any of the later tasks (line 15). That is, for each such cluster C_j, its remaining capacity is strictly less than the utilization of T^j (for the last cluster, we know that T^φ is τ_i). Thus, for any cluster C_j, its allocated capacity is strictly greater than $\lambda_j - u(T^j)$. Since tasks $\{\tau_1, \tau_2, ..., \tau_{i-1}\}$ have been successfully assigned, the total utilization of these tasks is equal to the total allocated capacity of clusters, which is given by $\sum_{k=1}^{i-1} u(\tau_k)$. Hence, we have

$$\sum_{k=1}^{i-1} u(\tau_k) > \sum_{j=1}^{\varphi}(\lambda_j - u(T^j))$$

$\Leftrightarrow$ $\{$adding $u(\tau_i)$ on both sides$\}$

$$\begin{aligned}
& \textstyle\sum_{k=1}^{i} u(\tau_k) > \sum_{j=1}^{\varphi}(\lambda_j - u(T^j)) + u(\tau_i) \\
\Leftrightarrow\quad & \{\text{because } \textstyle\sum_{j=1}^{\varphi} \lambda_j = m \text{ and } u(T^{\varphi}) = u(\tau_i)\} \\
& \textstyle\sum_{k=1}^{i} u(\tau_k) > m - \sum_{j=1}^{\varphi-1} u(T^j) \\
\Rightarrow\quad & \{\text{by the definition of } u_{\varphi-1}\} \\
& \textstyle\sum_{k=1}^{i} u(\tau_k) > m - u_{\varphi-1}.
\end{aligned}$$

□

Theorem 5.3. *Algorithm ASSIGN successfully partitions any DAG-based task system τ on φ clusters for which $u_{sum} \leq m - u_{\varphi-1}$.*

Proof. Let us suppose that Algorithm *ASSIGN* fails to assign the i^{th} task τ_i to any cluster. Then by Lemma 5.5, $\sum_{k=1}^{i} u(\tau_k) > m - u_{\varphi-1}$ holds. Therefore, we have

$$\begin{aligned}
& \textstyle\sum_{k=1}^{i} u(\tau_k) > m - u_{\varphi-1} \\
\Rightarrow\quad & \textstyle\sum_{k=1}^{N} u(\tau_k) = u_{sum} > m - u_{\varphi-1}.
\end{aligned}$$

Hence, any system that Algorithm *ASSIGN* fails to partition must have $u_{sum} > m - u_{\varphi-1}$. □

If φ is much smaller than m, which will often be the case in practice, then the proposed assignment algorithm results in little capacity loss even in the worst case.

Bounding ϖ_{sum}. For any given DAG system, if all DAGs can be assigned in the first phase, then $\varpi_{sum} = 0$. In the second phase, each DAG is considered in order, and if a cluster fails to accommodate a task (line 15), then it will never be considered for later tasks. Thus, it immediately follows that at most $\varphi - 1$ DAGs can contribute to ϖ_{sum}. Due to the fact that in the worst case all edges of a DAG can cause inter-cluster communication (as illustrated by the example below), an upper-bound on ϖ_{sum} under Algorithm *ASSIGN* is given by the sum of $\varphi - 1$ largest DAG data weights.

Example. Consider a scenario where three DAGs τ_i, τ_j, and τ_k are assigned to three clusters in a way as shown in Figure 5.11. Note that all edges of $\varphi - 1 = 2$ DAGs τ_j and τ_k contribute to ϖ_{sum}.

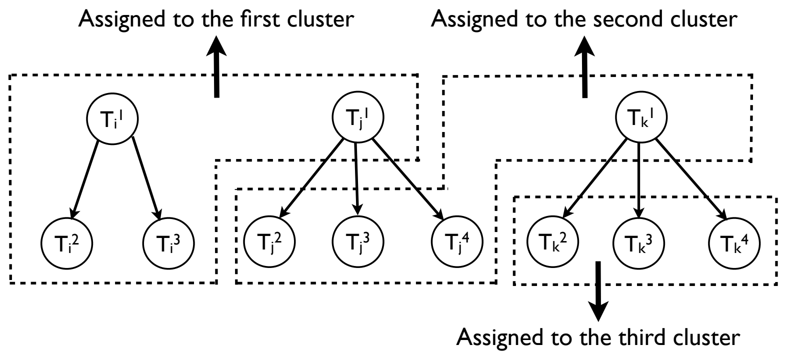

Assigned to the first cluster
Assigned to the second cluster
Assigned to the third cluster
T_i^1
T_i^2
T_i^3
T_j^1
T_j^2
T_j^3
T_j^4
T_k^1
T_k^2
T_k^3
T_k^4

Figure 5.11: Example worst-case scenario where all edges of $\varphi - 1$ DAGs contribute to the total communication cost.

5.2.4 Scheduling Phase

After executing the assignment phase, every task is mapped to a cluster. The scheduling phase of CDAG ensures that each task is scheduled with bounded tardiness. The scheduling phase consists of two steps: (*i*) transform each PGM graph into ordinary sporadic tasks by redefining job releases, and (*ii*) apply any window-constrained scheduling policy [76] such as GEDF to globally schedule the transformed tasks within each cluster.

Transforming PGM graphs into sporadic tasks. In Section 5.1, we showed that on a multiprocessor, any PGM graph system can be transformed into a set of ordinary sporadic tasks without capacity loss. The transformation process ensures that all precedence constraints in the original PGM graphs are met. This is done by redefining job releases properly. However, data communication delays (inter-cluster or intra-cluster) were not considered in Section 5.1. In this section, for each cluster, we apply the same approach but redefine job releases in a way such that data communications are considered. Later we shall show that this process still ensures bounded tardiness for any graph.

Definition 5.7. Let $F_{max}(pred(\tau_{l,j}^h), \upsilon_{l,j}^h)$ denote the latest completion time plus the data communication time among all predecessor jobs of $\tau_{l,j}^h$, where $\upsilon_{l,j}^h$ denotes the time to transfer data from the corresponding predecessor job of $\tau_{l,j}^h$ to $\tau_{l,j}^h$. For any predecessor job $\tau_{l,i}^k$ of $\tau_{l,j}^h$, $\upsilon_{l,j}^h$ can be computed by dividing ρ_l^{kh} (the number of produced data units on the corresponding edge) by the corresponding network bandwidth (i.e., B for inter-cluster data communications and b for intra-cluster data communications).

Definition 5.8. Let $t_f(\tau^h_{l,j})$ denote the completion time of job $\tau^h_{l,j}$.

The following equations can be applied to redefine job releases and deadlines in an iterative way (job $\tau^h_{l,j}$'s redefined release depends on the redefined release of $\tau^h_{l,j-1}$ where $j > 1$).

For any job $\tau^h_{l,j}$ where $j > 1$ and $h > 1$, its redefined release time, denoted $r(\tau^h_{l,j})$, is given by

$$\begin{aligned} r(\tau^h_{l,j}) &= max\big(r^h_{l,j}, r(\tau^h_{l,j-1}) + d^h_l, \\ &\quad F_{max}(pred(\tau^h_{l,j}), \upsilon^h_{l,j})\big). \end{aligned} \tag{5.18}$$

Given that a source task has no predecessors, the redefined release of any job $\tau^1_{l,j}$ $(j > 1)$ of such a task, $r(\tau^1_{l,j})$, is given by

$$r(\tau^1_{l,j}) = max\big(r^1_{l,j}, r(\tau^1_{l,j-1}) + d^1_l\big). \tag{5.19}$$

For the first job $\tau^h_{l,1}$ $(h > 1)$ of any non-source task, its redefined release, $r(\tau^h_{l,1})$, is given by

$$r(\tau^h_{l,1}) = max\big(r^h_{l,1}, F_{max}(pred(\tau^h_{l,j}), \upsilon^h_{l,j})\big). \tag{5.20}$$

Finally, for the first job $\tau^1_{l,1}$ of any source task, its release time is not altered, i.e.,

$$r(\tau^1_{l,1}) = r^1_{l,1}. \tag{5.21}$$

(Note that when redefining job releases in Section 5.1, the term $\upsilon^h_{l,j}$ did not appear in Equations (5.18)-(5.20) since data communications were not considered.)

After redefining job releases according to (5.18)-(5.20), any job $\tau^h_{l,j}$'s redefined deadline, denoted $d(\tau^h_{l,j})$, is given by

$$d(\tau^h_{l,j}) = r(\tau^h_{l,j}) + d^h_l. \tag{5.22}$$

Note that these definitions imply that each task's utilization remains unchanged. In particular, as shown in Section 5.2.5, bounded tardiness can be ensured for every transformed task in any cluster. Thus, Equations (5.18)-(5.20) delay any job release by a bounded amount, which implies that the execution rate and the relative deadline of each task is unchanged. Note also that the release time of any job $\tau_{l,j}^h$ with predecessor jobs is redefined to be at least $F_{max}(pred(\tau_{l,j}^h), \upsilon_{l,j}^h)$. Hence, the schedule preserves the precedence constraints enforced by the PGM model. Furthermore, since the release time of each $\tau_{l,j}^h$ $(j > 1)$ is redefined to be at least that of $\tau_{l,j-1}^h$ plus d_l^h, τ_l executes as a sporadic task with a period of d_l^h.

Example. Suppose that DAG τ_2 in Figure 5.9 is to be assigned to clusters in a way such that τ_1^1 and τ_1^2 are assigned to different clusters. For any job $\tau_{2,j}^2$ of τ_2, its predecessor job is $\tau_{2,j}^1$. Thus, assuming $B = 2$ as in Figure 5.1(c), by Definition 5.7, for any job $\tau_{2,j}^2$, we have $\upsilon_{2,j}^2 = \frac{\rho_2^{12}}{B} = \frac{4}{2} = 2$. Figure 5.12(a) shows the original job releases for τ_2^1 and τ_2^2 and Figure 5.12(b) shows the redefined job releases according to Equations (5.18)-(5.20) and the corresponding job executions. (Insets (c) and (d) are considered later.) Given that job $\tau_{2,1}^1$ completes at time 4, according to Equation (5.20), the release of $\tau_{2,1}^2$ is redefined to be at time 6. According to Equation (5.19), the release of $\tau_{2,2}^1$ is redefined to be at time 6. Then, $\tau_{2,2}^1$ completes at time 8. According to Equation (5.18), the release of $\tau_{2,2}^2$ is redefined to be at time 10, which is the completion time of its predecessor job $\tau_{2,2}^1$ plus the data communication time. Similarly, releases of other jobs can be defined by Equations (5.18)-(5.20). Note that the redefined job releases are in accordance with the sporadic task model. Moreover, τ_2^1 and τ_2^2 execute as if they were ordinary sporadic tasks, and yet all precedence constraints are satisfied.

5.2.5 Tardiness Bound

Given a PGM-specified system, by applying the strategy presented above, we obtain a transformed task system τ containing only independent sporadic tasks. Then, we can use GEDF to schedule tasks within each cluster with no capacity loss.

In the previous section, we derived a tardiness bound for any PGM system scheduled on a multiprocessor (which can be considered as a single cluster, as a special case of our multi-cluster system) under GEDF, without considering the communication time, as stated below.

Theorem 5.4. *The tardiness of any job $\tau_{l,j}^h$ of any task τ_l^h at depth k within a DAG τ_l scheduled under GEDF on a multiprocessor is at most $(k+1)\cdot\Delta + 3(k+1)\cdot y_l^{max}$, where $y_l^{max} = max(y_l^1, y_l^2, ..., y_l^z)$ (z is the number of nodes within τ_l) and Δ denotes the tardiness bound of τ_l with respect to its*

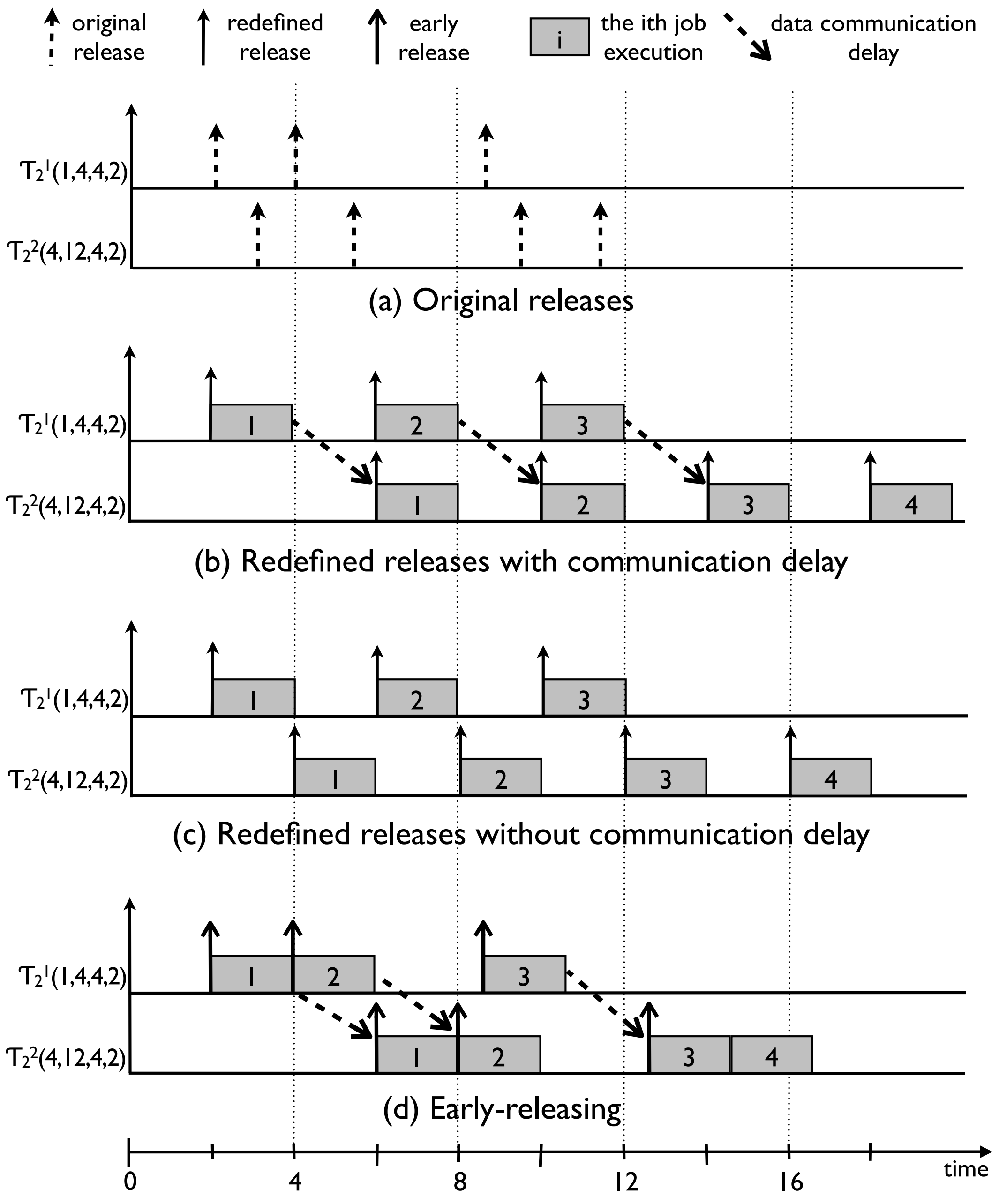


Figure 5.12: Illustrating various ideas on redefining job releases for DAG τ_2 in Figure 5.9.

redefined deadlines, as defined in Definition 5.2, i.e.,

$$t_f(\tau_{l,j}^h) - d(\tau_{l,j}^h) \leq \Delta. \tag{5.23}$$

Figure 5.12(b) shows the redefined releases and the job executions after considering data communication times (as covered earlier). Figure 5.12(c) shows the redefined releases and the corresponding job executions assuming no data communication time for DAG τ_2. As seen, the data communication further delays the redefined releases to later points of time. By bounding such data communication times and appropriately incorporating them into the prior tardiness bound (i.e., the one assuming no communication time), we are able to derive a final tardiness bound for every task in the given PGM system scheduled under CDAG, as stated in the following theorem. Before proving Theorem 5.5, we first state two lemmas that have been proved in Section 5.1 for systems without considering data communications.

Lemma 5.6. *For any job $\tau_{l,j}^1$, $r(\tau_{l,j}^1) - r_{l,j}^1 < 2 \cdot y_l^1$.*

Proof. By (5.19), $r(\tau_{l,j}^1)$ is *independent* of the data communication time. Thus, the proof is exactly the same as the one for proving Lemma 5.3 in Section 5.1. □

Lemma 5.7. *For any two jobs $\tau_{l,j}^h$ and $\tau_{l,k}^h$ of τ_l^h, where $j < k$, $i \cdot y_l^h \leq r_{l,j}^h < (i+1) \cdot y_l^h$ ($i \geq 0$), and $(i+w) \cdot y_l^h \leq r_{l,k}^h < (i+w+1) \cdot y_l^h$ ($w \geq 0$), we have $r_{l,k}^h - r_{l,j}^h > (k-j) \cdot d_l^h - 2 \cdot y_l^h$.*

Proof. The objective of this lemma is to prove the stated properties on $r_{l,k}^h$ and $r_{l,j}^h$, which are the *original releases* of any two jobs $\tau_{l,j}^h$ and $\tau_{l,k}^h$ of any task τ_l^h. Thus, the proof does *not* involve any data communication time. Therefore, the proof is exactly the same as the one for proving Lemma 5.4 in Section 5.1. □

Now we prove the following theorem.

Theorem 5.5. *The tardiness of any job $\tau_{l,j}^h$ of any task τ_l^h at depth k within a DAG τ_l scheduled under CDAG with respect to its original deadline is at most*

$$(k+1) \cdot \Delta + 3(k+1) \cdot (y_l^{max} + max(v_{l,j}^h)), \tag{5.24}$$

where $max(\upsilon_{l,j}^{h})$ denotes the maximum data communication time between any predecessor job of $\tau_{l,j}^{h}$ and $\tau_{l,j}^{h}$.

Proof. This theorem can be proved by induction on task depth. In the base case, by Theorem 5.4 and the fact that τ_l^1 has no predecessors, its tardiness with respect to its newly-defined deadline, $d(\tau_{l,j}^1)$, is at most Δ. By Lemma 5.6, $r(\tau_{l,j}^1) - r\tau_{l,j}^1 < 2 \cdot y_l^1$. Thus, with respect to its original deadline, $d\tau_{l,j}^1$, $\tau_{l,j}^1$ has a tardiness bound of $\Delta + 2 \cdot y_l^1 < \Delta + 3 \cdot y_l^{max}$.

For the induction step, let us assume (5.24) holds for any task τ_l^w at depth at most $k-1$, $k \geq 1$. Then, the tardiness of any job $\tau_{l,v}^w$ of τ_l^w is at most $k \cdot \Delta + 3k \cdot (y_l^{max} + max(\upsilon_{l,j}^h))$, i.e.,

$$t_f(\tau_{l,v}^w) - d_{l,v}^w \leq k \cdot \Delta + 3k \cdot (y_l^{max} + max(\upsilon_{l,j}^h)). \tag{5.25}$$

We want to prove that for any job $\tau_{l,j}^h$ of any task τ_l^h at depth k, $t_f(\tau_{l,j}^h) - d_{l,j}^h \leq (k+1) \cdot \Delta + 3(k+1) \cdot (y_l^{max} + max(\upsilon_{l,j}^h))$. According to (5.18) and (5.20), there are three cases to consider regarding $\tau_{l,j}^h$'s newly-defined release time $r(\tau_{l,j}^h)$.

Case 1. $r(\tau_{l,j}^h) = r_{l,j}^h$. By Theorem 5.4, we know that $t_f(\tau_{l,j}^h) - d(\tau_{l,j}^h) \leq \Delta$. Given that $d(\tau_{l,j}^h) = d_{l,j}^h$, we have $t_f(\tau_{l,j}^h) - d_{l,j}^h \leq \Delta < (k+1) \cdot \Delta + 3(k+1) \cdot (y_l^{max} + max(\upsilon_{l,j}^h))$.

Case 2. $r(\tau_{l,j}^h) = F_{max}(pred(\tau_{l,j}^h), \upsilon_{l,j}^h)$. Let $\tau_{l,v}^w$ be the predecessor of $\tau_{l,j}^h$ that has the latest completion time among all predecessors of $\tau_{l,j}^h$ ($\tau_{l,v}^w$ exists because the depth of τ_l^h is at least one). Thus, we have

$$r(\tau_{l,j}^h) = F_{max}(pred(\tau_{l,j}^h), \upsilon_{l,j}^h) \leq t_f(\tau_{l,v}^w) + max(\upsilon_{l,j}^h). \tag{5.26}$$

Therefore,

$$\begin{aligned}
& t_f(\tau_{l,j}^h) - d_{l,j}^h \\
& \{\text{by } (5.15)\} \\
= \quad & t_f(\tau_{l,j}^h) - r_{l,j}^h - d_l^h \\
= \quad & t_f(\tau_{l,j}^h) - r(\tau_{l,j}^h) + r(\tau_{l,j}^h) - r_{l,j}^h - d_l^h \\
& \{\text{by } (5.22)\} \\
= \quad & t_f(\tau_{l,j}^h) - d(\tau_{l,j}^h) + d_l^h + r(\tau_{l,j}^h) - r_{l,j}^h - d_l^h \\
& \{\text{by } (5.23)\}
\end{aligned}$$

$\leq \quad \Delta + d_l^h + r(\tau_{l,j}^h) - r_{l,j}^h - d_l^h$

$= \quad \Delta + r(\tau_{l,j}^h) - r_{l,j}^h$

$\{\text{by } (5.16) \text{ and } (5.26)\}$

$\leq \quad \Delta + t_f(\tau_{l,v}^w) + max(\upsilon_{l,j}^h) - r_{l,v}^w$

$\{\text{by } (5.15)\}$

$= \quad \Delta + t_f(\tau_{l,v}^w) + max(\upsilon_{l,j}^h) - d_{l,v}^w + d_l^w$

$\{\text{by } (5.25)\}$

$\leq \quad \Delta + k \cdot \Delta + 3k \cdot (y_l^{max} + max(\upsilon_{l,j}^h))$

$+max(\upsilon_{l,j}^h) + d_l^w$

$\{\text{by } (5.1)\}$

$< \quad (k+1) \cdot \Delta + 3(k+1) \cdot (y_l^{max} + max(\upsilon_{l,j}^h)).$

Case 3. $j > 1 \wedge r(\tau_{l,j}^h) = r(\tau_{l,j-1}^h) + d_l^h$. Let $\tau_{l,q}^h$ $(q < j)$ denote the last job of τ_l^h released before $\tau_{l,j}^h$ such that $r(\tau_{l,q}^h) = r_{l,q}^h$ or $r(\tau_{l,q}^h) = F_{max}(pred(\tau_{l,q}^h), \upsilon_{l,j}^h)$. $\tau_{l,q}^h$ exists because according to (5.20) and (5.21), there exists at least one job, $\tau_{l,1}^h$, such that $r(\tau_{l,1}^h) = r_{l,1}^h$ or $r(\tau_{l,1}^h) = F_{max}(pred(\tau_{l,1}^h), \upsilon_{l,j}^h)$. Depending on the value of $r(\tau_{l,q}^h)$, we have two subcases.

Case 3.1. $r(\tau_{l,q}^h) = r_{l,q}^h$. By the definition of $\tau_{l,q}^h$, the release time of any job $\tau_{l,k}^h$, where $q < k \leq j$, is redefined to be $r(\tau_{l,k}^h) = r(\tau_{l,k-1}^h) + d_l^h$. Thus, we have

$$r(\tau_{l,j}^h) = r(\tau_{l,q}^h) + (j - q) \cdot d_l^h. \tag{5.27}$$

Therefore, we have

$t_f(\tau_{l,j}^h) - d_{l,j}^h$

$\{\text{by } (5.15)\}$

$= \quad t_f(\tau_{l,j}^h) - r_{l,j}^h - d_l^h$

$= \quad t_f(\tau_{l,j}^h) - r(\tau_{l,j}^h) + r(\tau_{l,j}^h) - r_{l,j}^h - d_l^h$

$\{\text{by } (5.22)\}$

$= \quad t_f(\tau_{l,j}^h) - d(\tau_{l,j}^h) + d_l^h + r(\tau_{l,j}^h) - r_{l,j}^h - d_l^h$

$\{\text{by } (5.23)\}$

$$\begin{aligned}
&\le \quad \Delta + d_l^h + r(\tau_{l,j}^h) - r_{l,j}^h - d_l^h \\
&= \quad \Delta + r(\tau_{l,j}^h) - r_{l,j}^h \\
&\qquad \{\text{by (5.27) and Lemma 5.7}\} \\
&< \quad \Delta + (r(\tau_{l,q}^h) + (j-q)\cdot d_l^h) - (r_{l,q}^h \\
&\qquad +(j-q)\cdot d_l^h - 2\cdot y_l^h) \\
&\qquad \{\text{by the case condition}\} \\
&= \quad \Delta + 2\cdot y_l^h \\
&< \quad (k+1)\cdot\Delta + 3(k+1)\cdot(y_l^{max} + max(\upsilon_{l,j}^h)).
\end{aligned}$$

Case 3.2. $r(\tau_{l,q}^h) = F_{max}(pred(\tau_{l,q}^h), \upsilon_{l,j}^h)$. Let $\tau_{l,v}^w$ denote a predecessor job of $\tau_{l,q}^h$ with $t_f(\tau_{l,v}^w) = F_{max}(pred(\tau_{l,q}^h), \upsilon_{l,j}^h) - dc_{l,q}^h(\tau_{l,v}^w) = r(\tau_{l,q}^h) - dc_{l,q}^h(\tau_{l,v}^w)$, where $dc_{l,q}^h(\tau_{l,v}^w)$ denotes the data communication time between $\tau_{l,v}^w$ and $\tau_{l,q}^h$. We have

$$\begin{aligned}
&\qquad t_f(\tau_{l,j}^h) - d_{l,j}^h \\
&\qquad \{\text{similarly to the derivation in Case 3.1}\} \\
&< \quad \Delta + (r(\tau_{l,q}^h) + (j-q)\cdot d_l^h) - (r_{l,q}^h \\
&\qquad +(j-q)\cdot d_l^h - 2\cdot y_l^h) \\
&= \quad \Delta + r(\tau_{l,q}^h) - r_{l,q}^h + 2\cdot y_l^h \\
&\qquad \{\text{by the case condition and (5.16)}\} \\
&\le \quad \Delta + t_f(\tau_{l,v}^w) + dc_{l,q}^h(\tau_{l,v}^w) - r_{l,v}^w + 2\cdot y_l^h \\
&\qquad \{\text{by (5.15)}\} \\
&= \quad \Delta + t_f(\tau_{l,v}^w) + dc_{l,q}^h(\tau_{l,v}^w) - d_{l,v}^w + d_l^w + 2\cdot y_l^h \\
&\qquad \{\text{by (5.25)}\} \\
&\le \quad \Delta + k\cdot\Delta + 3k\cdot(y_l^{max} + max(\upsilon_{l,j}^h)) + dc_{l,q}^h(\tau_{l,v}^w) \\
&\qquad +d_l^w + 2\cdot y_l^h \\
&\qquad \{\text{by (5.1)}\} \\
&< \quad (k+1)\cdot\Delta + 3(k+1)\cdot(y_l^{max} + max(\upsilon_{l,j}^h)).
\end{aligned}$$

□

Note that a per-task response time bound can be obtained from the above tardiness bound by adding the task's relative deadline. Such bounds are useful in settings where response time is used as the performance metric.[5] Note also that since no capacity loss occurs during the scheduling phase, any PGM system is *schedulable* with bounded response times as long as it can be partitioned onto clusters under CDAG.

5.2.6 Improving Job Response Times

Similar to the discussion in Section 5.1.4, we can apply the early-releasing technique to improve job response times.

According to Equations (5.18)-(5.20), we delay job releases to transform DAGs into sporadic tasks. However, excessive release delays are actually unnecessary and actual response times can be improved by applying early-releasing. The earliest time at which job $\tau_{l,j}^h$ may execute is defined by its *early-release time* $\varepsilon(\tau_{l,j}^h)$, where $\varepsilon(\tau_{l,j}^h) \leq r(\tau_{l,j}^h)$. For any job $\tau_{l,j}^h$, its early-releasing time can be defined as

$$\varepsilon(\tau_{l,j}^h) = \begin{cases} r_{l,j}^h & \text{if } h = 1 \\ F_{max}(pred(\tau_{l,j}^h), \upsilon_{l,j}^h) & \text{if } h > 1. \end{cases}$$

Thus, an unfinished job $\tau_{l,j}^h$ is *eligible* for execution at time t if $\tau_{l,j-1}^h$ has completed by t (if $j > 1$) and $t \geq \varepsilon(\tau_{l,j}^h)$.

Example. Consider again the scheduling of τ_2 as shown in Figure 5.12(b). Figure 5.12(d) shows early releases as defined above and the corresponding GEDF schedule. As seen, most jobs' response times are improved. For instance, $\tau_{2,2}^2$ now completes at time 10, two time units earlier than the case without early-releasing.

5.2.7 Experiments

In this section, we describe experiments conducted using randomly-generated DAG sets to evaluate the effectiveness of CDAG in minimizing capacity loss and total communication cost. We do this by comparing CDAG with the optimal ILP solution. The experiments focus on three performance metrics: (*i*) capacity loss, (*ii*) total communication cost, and (*iii*) each test's runtime performance.

In our experiments, we selected a random target size for DAGs, from at least one task to 100 per DAG. Then tasks within each DAG were generated based upon distributions proposed by Baker

[5]In some PGM-specified applications, deadlines are not specified but bounded response times are still required [51].

[8]. The source task of each DAG was assumed to be released sporadically, with a period uniformly distributed over $[10ms, 100ms]$. The produce amount of each edge was varied from 10 data units to 1000 data units. For every edge of each DAG, its produce amount, threshold, and consume amount were assumed to be the same. Valid execution rates were calculated for non-source tasks within each DAG using results from Section 5.1. Task utilizations were distributed using four uniform distributions, $[0.05, 0.2]$ (light), $[0.2, 0.5]$ (medium), $[0.5, 0.8]$ (heavy), and $[0.05, 0.8]$ (uniform). Task execution costs were calculated from execution rates and utilizations. We generated six clusters, each with a random processor count from 4 to 16, with a total processor count of 48. We assumed $B = 10$ and $b = 1000$. For each choice of utilization distribution, a cap on overall utilization was systematically varied within $[16, 48]$. For each combination of utilization cap and utilization distribution, we generated 100 DAG sets. Each such DAG set was generated by creating DAGs until total utilization exceeded the corresponding utilization cap, and by then reducing the last DAG's utilization so that the total utilization equalled the utilization cap.

The schedulability results that were obtained are shown in Figures 5.13-5.16. In all of these Figures, "CDAG" denotes the schedulability results achieved by CDAG, "Thm. 1 Bound" denotes the worst-case utilization bound of CDAG as stated in Theorem 1, and "ILP" denotes the schedulability results achieved by ILP. Each curve in each figure plots the fraction of the generated DAG sets that the corresponding approach successfully scheduled, as a function of total utilization. (Note that the range of the x-axis in all insets is given by $[43, 48]$.) As Figures 5.13-5.16 show, under all four utilization distributions, CDAG yields schedulability results that are very close to that achieved by ILP. Moreover, the worst-case utilization bound in Theorem 1 is reasonable. For example, under the light per-task utilization distribution, the worst-case utilization bound of CDAG ensures that any DAG set with a total utilization up to 47 can be successfully scheduled in a distributed system containing 48 processors.

Table 5.3 shows the total communication cost achieved by both approaches categorized by the total utilization U_{sum} using the light per-task utilization distribution. In these experiments, all DAG sets were guaranteed to be schedulable since the total utilization of any DAG set (at most 44) is less than the worst-case utilization bound of CDAG, which is 47. In Table 5.3, for each U_{sum}, the total communication cost under ILP or CDAG was computed by taking the average of the total communication cost over the 100 generated DAG sets. The total communication cost for each

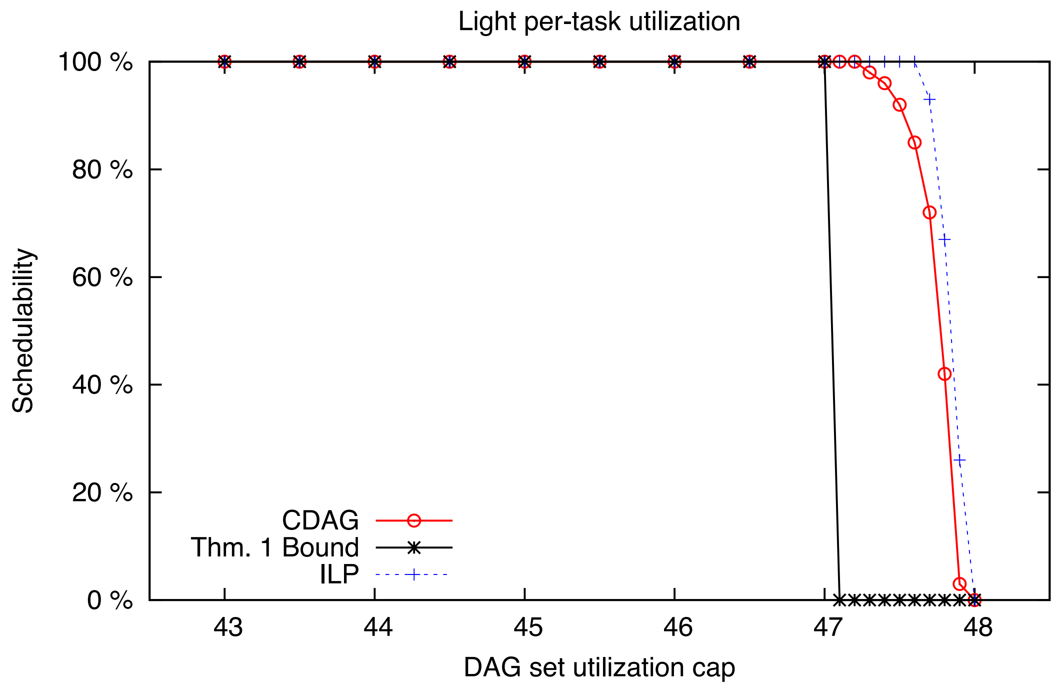


Figure 5.13: Schedulability results: light per-task utilization distribution.

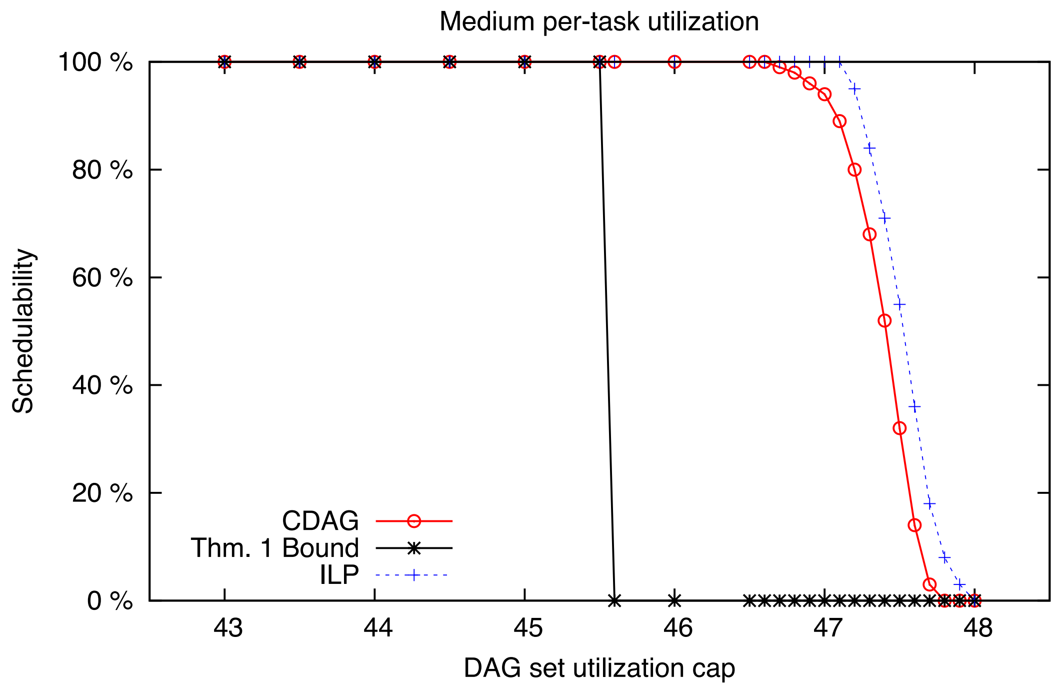


Figure 5.14: Schedulability results: medium per-task utilization distribution.

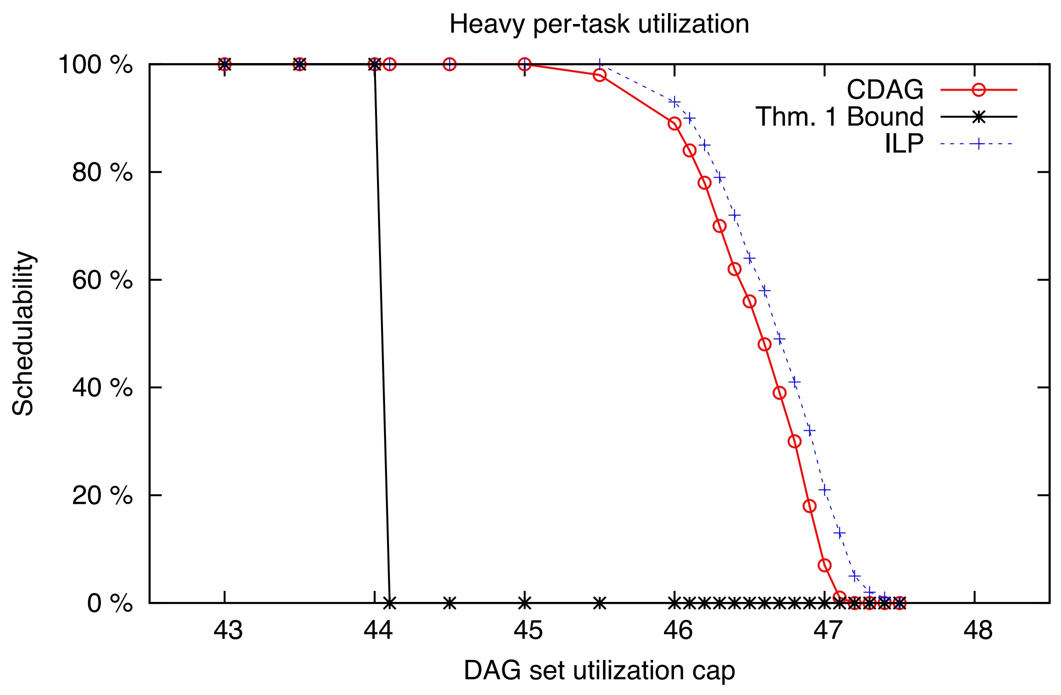


Figure 5.15: Schedulability results: heavy per-task utilization distribution.

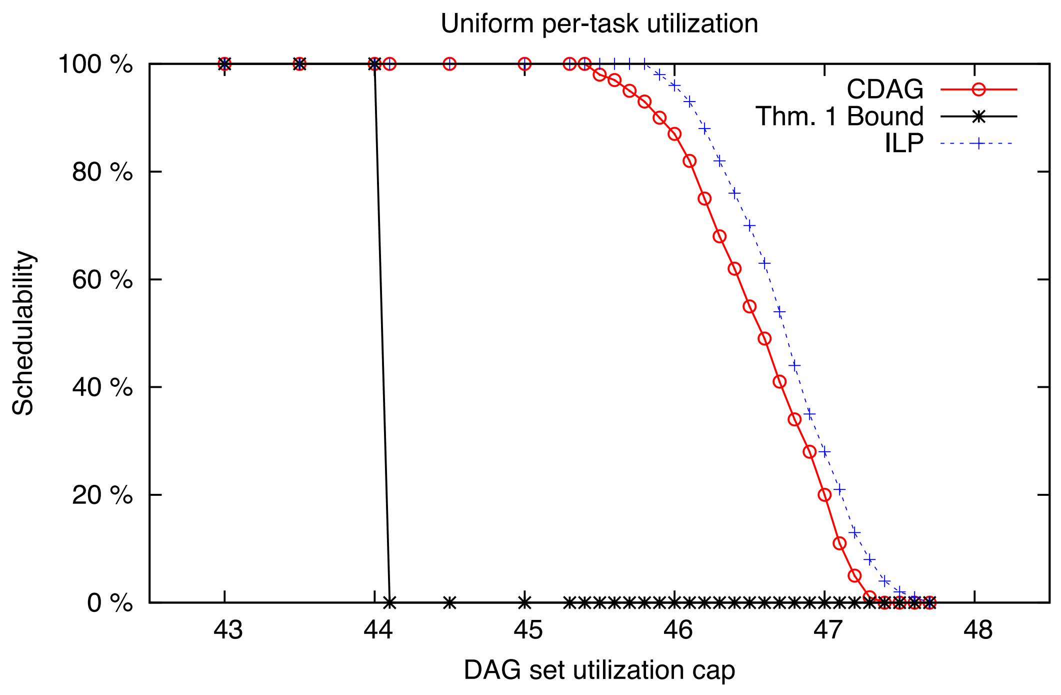


Figure 5.16: Schedulability results: uniform per-task utilization distribution.

Table 5.3: Total communication cost.

DAG set utilization / Method	U_{sum}=16	U_{sum}=20	U_{sum}=24	U_{sum}=28	U_{sum}=32	U_{sum}=36	U_{sum}=40	U_{sum}=44
ILP	6.4	7.5	10.6	17	22.5	26.2	29.8	32.6
CDAG	78.1	87.2	130.5	146.3	173.1	187.7	231.2	248.7
Total	34588.9	45048.8	52968.1	58895.3	69895.9	77125.5	86741.4	93775.9

Table 5.4: Runtime performance.

# of tasks / Method	N=100	N=200	N=300	N=400	N=500
ILP	51.9 (s)	165.9 (s)	412.2 (s)	2848.1 (s)	37791.8 (s)
CDAG	0.48 (ms)	0.58 (ms)	0.62 (ms)	0.56 (ms)	0.71 (ms)

generated DAG set is given by ϖ_{sum} as defined in Definition 5.4. The label "Total" represents the maximum communication cost of the DAG set, which is given by $\sum_{\tau_i \in \zeta} \sum_{e_i^{jk} \in \tau_i} w_i^{jk}$ where (as noted earlier) ζ represents the corresponding DAG set. As seen, CDAG is effective in minimizing the total communication cost. The total communication costs achieved by CDAG are close to the optimal ones achieved by ILP and are significantly smaller than the maximum communication costs. For example, when $U_{sum} = 44$, CDAG achieves a total communication cost of 248.7 data units while ILP gives an optimal solution of 32.6 data units, both of which are almost negligible compared to the maximum communication cost, which is 93775.9 data units.

Regarding runtime performance, Table 5.4 shows the average time to run an experiment as a function of the number of tasks N using the light per-task utilization distribution. For each N in the set $\{100, 200, 300, 400, 500\}$, we generated ten DAG sets and recorded the average running time of both ILP and CDAG. CDAG consistently took less than 1 ms to run while ILP ran for a significantly longer time, sometimes prohibitively so. For instance, when $N = 500$, ILP took more than 10 hours on average per generated DAG set.

5.3 Chapter Summary

In this chapter, we proposed a variant of EDF that can achieve no capacity loss for scheduling PGM graphs in multicore systems while providing timing correctness guarantees. We later extended this work to support PGM graphs in a distributed system containing multiple multicore-based clusters. The impact of this work is demonstrated by the fact that we closed a problem that stood open for 12 years. Since PGM is widely used today in the design of many signal processing and radar applications deployed in a number of submarines and helicopters in service, these research results provide designers of such military systems a set of analytically correct and practically efficient methodologies that can be applied to avoid capacity loss. Such military systems are often mission-critical; any reduction of the number of hardware components is thus significant as it reduces complexity and improves reliability.

CHAPTER 6

Multiprocessor Scheduling of SRT Tasks with Non-Preemptive Sections, Self-Suspensions, and Graph-based Precedence Constraints[1]

In the previous three chapters, we presented solutions that can support real-time tasks with either self-suspensions or graph-based precedence constraints on multiprocessors. In this chapter, we consider this issue in the context of sporadic task systems in which mixed types of runtime behaviors may exist due to non-preemptive sections, self-suspensions, and graph-based precedence constraints. The timing correctness of such a system may be quite difficult to analyze, particularly if deadline misses cannot be tolerated. However, we show in this chapter that the situation is not nearly so bleak, if bounded deadline tardiness is acceptable.

Specifically, we address the problem of deriving conditions under which bounded tardiness can be ensured when *all* of the above-mentioned behaviors—non-preemptive sections, graph-based precedence constraints,[2] and self-suspensions—are allowed. For conciseness, we use NGS task systems to name such systems. In considering this problem, we focus specifically on the GEDF algorithm. Our main result is a transformation process that converts any implicit-deadline sporadic NGS task system into a simpler system with only self-suspensions. In the simpler system, each task's maximum job response time is at least that of the original system. This result allows tardiness bounds to be established by focusing only on the impacts of suspensions.

[1]Contents of this chapter previously appeared in preliminary form in the following paper:
Cong Liu and James Anderson. Scheduling Suspendable, Pipelined Tasks with Non-Preemptive Sections in Soft Real-Time Multiprocessor Systems, Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 23-32, 2010.

[2]To enable our results to be applied to systems with rather sophisticated graph-based precedence constraints, we consider PGM task graphs.

The rest of this chapter is organized as follows. Section 6.1 describes our system model. The transformation discussed above is obtained via a sequence of sub-transformations, which are described in Sections 6.2.1 and 6.2.2. In Section 6.3, a tardiness bound is derived. Section 6.4 concludes this chapter.

6.1 System Model

We consider the problem of scheduling a set $\tau^{NGS} = \{G_1, ..., G_n\}$ of n SRT PGM task graphs on $m \geq 2$ identical processors under GEDF, where any task within a graph may self-suspend and contain non-preemptive sections. As shown in Chapter 5, a PGM task graph can be naturally represented as a DAG-based RB task. Thus, the more sophisticated PGM task graph considered in this chapter can be represented as a DAG-based RB self-suspending task (defined in Section 2.1.7), as we discuss in the next section.

Example. Figure 6.1 depicts an example GEDF schedule of an NGS task system, scheduled on three processors. This task system contains a PGM task graph containing two subtasks, each with a period of 10 time units, and another PGM task graph containing three subtasks, each with a period of 10 time units. τ_1^1 executes for 2 time units, then suspends for 3 time units, executes for another 2 time units, and finally suspends for 1 time unit. τ_1^2 executes for 4 time units, then suspends for 1 time unit, and executes for another 2 time units. τ_2^1 executes for 3 time units, and then suspends for 1 time unit. τ_2^2 executes for 9 time units. τ_2^3 first suspends for 1 time unit, then executes non-preemptively for 3 time units and then preemptively for another time unit, suspends for 1 time unit, and finally executes for 4 time units. As seen in the GEDF schedule, $\tau_{2,1}^2$ misses its deadline at time 20 by 1 time unit, which causes $\tau_{2,1}^3$ to start its first suspension phase at time 21. At time 24, $\tau_{2,1}^3$ starts executing its first computation phase since $\tau_{1,3}^1$, $\tau_{1,2}^2$, $\tau_{2,3}^1$ are suspended at that time. Since this execution is non-preemptive, at time 25, $\tau_{1,2}^2$ preempts $\tau_{2,1}^2$ instead of $\tau_{2,1}^3$.

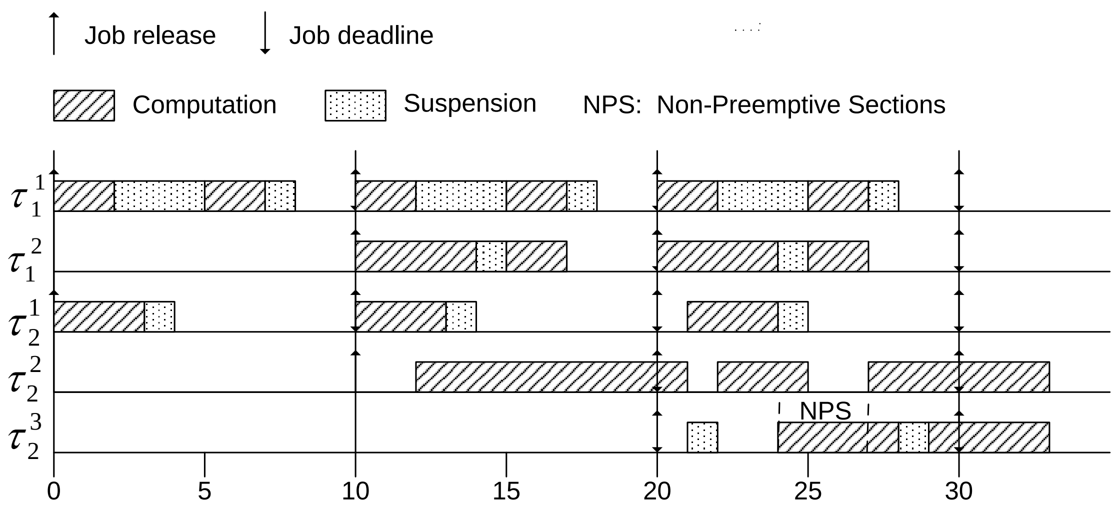


Figure 6.1: Example NGS task system.

6.2 Transformation.

In this section, we show how to transform τ^{NGS}, a sporadic NGS task system, into a sporadic task system with only suspensions. This transformation requires two steps:

1. Transform τ^{NGS} into τ^{GS}, where τ^{GS} denotes a fully preemptive suspendable PGM task graph system, by treating blocking times due to non-preemptive sections as suspensions. This is dealt with in Section 6.2.1.

2. Transform τ^{GS} into τ^{S}, where τ^{S} is a sporadic task system with only suspensions (i.e., it contains no PGM task graphs and it is fully preemptive), by eliminating graph-based precedence constraints as done in Chapter 5. This is dealt with in Section 6.2.2.

6.2.1 Transforming τ^{NGS} to τ^{GS}

We transform τ^{NGS} into τ^{GS} by treating blocking times due to non-preemptive sections as suspensions.

Definition 6.1. We say that a task system τ is *concrete* if the actual execution cost and suspension time of every job of each task is fixed. For any τ (τ may be any of the task systems mentioned in the roadmap at the end of the prior section), we let $\overline{\tau}$ denote any arbitrary concrete instantiation of it.

A job $\tau_{l,j}^{h}$ is non-preemptively blocked, or *NP-blocked*, at time t if it is among m highest-priority enabled jobs according to GEDF, but it cannot execute because lower-priority jobs are executing

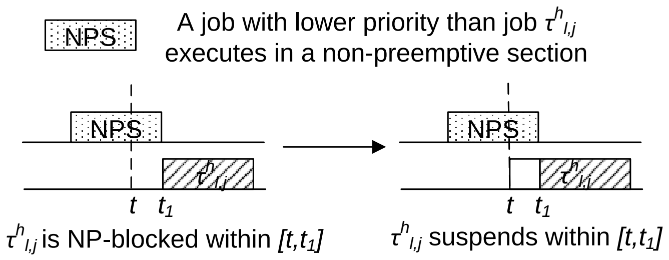

NPS
A job with lower priority than job $\tau^h_{l,j}$ executes in a non-preemptive section
NPS
$\tau^h_{l,j}$
t t_1
$\tau^h_{l,j}$ is NP-blocked within $[t,t_1]$
NPS
$\tau^h_{l,j}$
t t_1
$\tau^h_{l,j}$ suspends within $[t,t_1]$

Figure 6.2: Modeling NP-blocking as suspension.

non-preemptively at t. This can happen only when $\tau^h_{l,j}$ commences executing one of its computation phases. Given that any job $\tau^h_{l,j}$ has at most c^h_l such phases and the maximum length of any job's non-preemptive section is at most b_{max}, we have the following lemma.

Lemma 6.1. *Any job $\tau^h_{l,j}$ in τ^{NGS} can be NP-blocked for at most $c^h_l \cdot b_{max}$ time units.*

Given Lemma 6.1, we can define τ^{GS} by simply treating NP-blocking times as suspensions. That is, we view all non-preemptive sections as preemptive, and for any subtask τ^h_l, we increase s^h_l by $c^h_l \cdot b_{max}$, which by Lemma 6.1 upper-bounds the NP-blocking time of τ^h_l. This is illustrated in Figure 6.2. The theorem below immediately follows.

Theorem 6.1. *For any concrete instantiation $\overline{\tau}^{NGS}$ of τ^{NGS}, there exists a concrete instantiation $\overline{\tau}^{GS}$ of τ^{GS} such that $\overline{\tau}^{NGS}$ and $\overline{\tau}^{GS}$ have equivalent GEDF schedules.*[3]

Note that this transformation strongly exploits the fact that, in our task model, suspension phases are upper-bounded, and hence, can be reduced to reflect actual NP-blocking times. Note also that the "reverse" of this theorem may not hold: for a concrete instantiation $\overline{\tau}^{GS}$ of τ^{GS}, there may not exist a concrete instantiation $\overline{\tau}^{NGS}$ of τ^{NGS} such that $\overline{\tau}^{GS}$ and $\overline{\tau}^{NGS}$ have equivalent GEDF schedules.

Corollary 4. For any subtask τ^h_l, if the maximum response time of any job of τ^h_l in any GEDF schedule for τ^{GS} is z time units, then the maximum response time of any such job in any GEDF schedule for τ^{NGS} is at most z time units.

[3]That is, if S^{NGS} (S^{GS}) is the GEDF schedule for $\overline{\tau}^{NGS}$ ($\overline{\tau}^{GS}$), then job $\tau^h_{l,j}$ is scheduled at time t in S^{NGS} iff it is scheduled at time t in S^{GS}.

6.2.2 Transforming τ^{GS} to τ^{S}

After obtaining τ^{GS}, we further transform τ^{GS} into τ^{S}. τ^{GS} can be viewed as a special PGM task system where subtasks within each PGM may self-suspend. That is, the only difference between τ^{GS} and the ordinary PGM task system considered in Chapter 5 is that subtasks within PGM task graphs in τ^{GS} may self-suspend.

Sections 5.1.1 and 5.1.2 in Chapter 5 demonstrate approaches of transforming an ordinary PGM task system into an ordinary sporadic task system. We can apply these approaches in order to transform τ^{GS} into τ^{S}. As described in Section 5.1.1, we first represent each PGM graph in τ^{GS} as a DAG-based RB task by mapping PGM nodes to RB tasks. Since this process is independent of whether subtasks within PGM task graphs self-suspend, we can apply the same approach presented in Section 5.1.1.

We then transform the obtained DAG-based RB task system to an ordinary sporadic self-suspending task system τ^{S} by applying the approach presented in Section 5.1.2. We are able to apply this same approach because it only modifies job releases and deadlines, and thus can be applied to the DAG-based RB task system transformed from τ^{GS}. After this transformation, we obtain τ^{S} that consists of a set of independent sporadic self-suspending tasks.

6.3 A Tardiness Bound

We now derive a tardiness bound for the original task system τ^{NGS}. As described in Section 5.1.3 in Chapter 5, this tardiness bound can be obtained by first deriving a tardiness bound for each job with respect to its redefined deadline (as presented in Theorem 5.1). Since the actual tardiness bound of any job should be with respect to its original deadline, we then upper-bound the length between each job's original deadline and its redefined deadline.

As shown in Theorem 5.1, for ordinary PGM task graphs, we can obtain a set of independent ordinary sporadic tasks after the transformation. Therefore, Theorem 5.1 applies the tardiness bound derived for ordinary sporadic task systems in [41]. However, since τ^{S} consists of a set of self-suspending tasks, we instead apply the tardiness bound derived for sporadic self-suspending task systems as given in Theorem 3.3 in Section 3.4.5, as stated below.

Definition 6.2. Let $\Delta = \dfrac{\overline{E} - e_l - s_l}{m - \overline{U}_{m-1}} + e_l^h + s_l^h$, where e_{sum} is the sum of all subtasks' WCET, e_{min} is the smallest WCET among all subtasks, and U_{m-1} is the total utilization of $m-1$ subtasks with the largest utilizations.

Note that Definition 6.2 differs from Definition 5.2 in terms of the specific definition of the term Δ. The Δ terms seen in Theorem 5.1 and Theorem 6.2 given below represent the tardiness bound for ordinary sporadic task systems and self-suspending task systems, respectively.

Theorem 6.2. *In any GEDF schedule for the sporadic task system τ on m processors, if $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ where v^j is defined in Definition 3.16 in Section 3.4.1 in Chapter 3, then the tardiness of any job $\tau_{l,j}^h$, with respect to its redefined deadline, is at most Δ defined in Definition 6.2.*

We now bound the actual tardiness any job $\tau_{l,j}^h$ may experience with respect to its original deadline. The following theorem gives such a bound. Let $y_l^{max} = max(y_l^1, y_l^2, ..., y_l^h)$, where h is the number of subtasks in G_l. Lemma 6.2 and Theorem 6.3 given below were originally proved in Lemma 5.4 and Theorem 5.5 presented in Section 5.1.3. They continue to hold in this case because their proofs do not rely on whether jobs self-suspend. Note that in Theorem 6.3, the utilization constraint $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ and the term Δ are different from Theorem 5.5 because the tardiness bound with respect to a job's redefined deadline and the utilization constraint in this case are given in Theorem 6.2.

Lemma 6.2. *For any two jobs $\tau_{l,j}^h$ and $\tau_{l,k}^h$ of τ_l^h in τ^{RB}, where $j < k$, $i \cdot y_l^h \leq r^{RB}(\tau_{l,j}^h) < (i+1) \cdot y_l^h$ ($i \geq 0$), and $(i+w) \cdot y_l^h \leq r^{RB}(\tau_{l,k}^h) < (i+w+1) \cdot y_l^h$ ($w \geq 0$), we have $r^{RB}(\tau_{l,k}^h) - r^{RB}(\tau_{l,j}^h) > (k-j) \cdot d_l^h - 2 \cdot y_l^h$.*

Theorem 6.3. *In any GEDF schedule for τ^{GS} on m processors, if $U_{sum} + \sum_{i=1}^{m} v^j \leq m$, then the tardiness of any job $\tau_{l,j}^h$ of a task τ_l^h at depth k, with respect to its original deadline, is at most $(k+1) \cdot \Delta + 3(k+1) \cdot y_l^{max}$, where Δ is given in Definition 6.2.*

6.4 Chapter Summary

In this chapter, we presented a method for transforming a sporadic NGS task system into a simpler sporadic task system with only suspensions. The transformation allows maximum response-time bounds derived for sporadic suspending task systems to be applied to sporadic NGS task systems.

CHAPTER 7

A Response Time Bound for Scheduling Real-Time Parallel Tasks on a Multiprocessor[1]

The growing prevalence of multicore platforms has resulted in the wider applicability of parallel programming models such as OpenMP [28] and MapReduce [36]. Such models can be applied to parallelize certain segments of programs, thus better utilizing hardware resources and possibly shortening response times. Many applications implemented under such parallel programming models have SRT constraints. Examples include real-time parallel video and image processing applications [5, 43] and computer vision applications such as colliding face detection and feature tracking [66]. In these applications, providing fast and bounded response times for individual video frames is important, to ensure smooth video output. However, achieving this at the expense of using conservative HRT analysis is not warranted. In this chapter, we consider how to schedule parallel task systems that require such SRT performance guarantees on multicore processors.

Parallel task models pose new challenges to real-time scheduling since intra-task parallelism has to be specifically considered. Recent work (as reviewed in Section 2.3.5) on scheduling real-time sporadic parallel tasks have focused on providing HRT guarantees under GEDF or PDM scheduling. However, as discussed above, viewing parallel tasks as HRT may be overkill in many settings and furthermore may result in significant schedulability-related capacity loss. Thus, our focus is to instead ensure bounded response times in supporting parallel task systems by applying SRT scheduling analysis techniques. Specifically, we consider whether it is possible to specify reasonable constraints under which bounded response times can be guaranteed using global real-time scheduling techniques, for sporadic parallel task systems that are not HRT in nature.

[1]Contents of this chapter previously appeared in preliminary form in the following paper:
Cong Liu and James Anderson. Supporting Soft Real-Time Parallel Applications on Multicore Processors, Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pages 114-123, 2012.

In this chapter, we show that parallel task systems can be supported on multiprocessors under GEDF-like schedulers with bounded response times. Our analysis shows that on a two-processor platform, no capacity loss results for any parallel task system. Despite this special case, on a platform with more than two processors, utilization constraints are needed. To discern how severe such constraints must fundamentally be, we present a parallel task set with minimum utilization that is unschedulable on any number of processors. This task set violates our derived constraint and has unbounded response times. The impact of utilization constraints can be lessened by restructuring tasks to reduce intra-task parallelism. We propose optimization techniques that can be applied to determine such a restructuring. Finally, we present the results of experiments conducted to evaluate the applicability of the derived schedulability condition. We describe our parallel task model next.

7.1 System Model and Notation

We consider the problem of scheduling a set $\tau = \{\tau_1, ..., \tau_n\}$ of n independent sporadic parallel tasks (defined in Section 2.3.5) on m processors. For clarity, a summary of important terms defined so far, as well as some additional terms defined later, is presented in Table 7.1. We assume $v_i^{max} \geq 2$ holds for at least one task τ_i; otherwise, the considered task system is simply an ordinary sporadic task system (without intra-task parallelism).

Each parallel task τ_i has a specified relative deadline of d_i, which may differ from p_i (thus, our analysis is applicable to soft real-time arbitrary-deadline sporadic parallel tasks). We do not use such deadlines in prioritizing jobs, but rather assign each job $\tau_{i,k}$ a priority point at $d_{i,k} = r_{i,k} + p_i$ and schedule jobs on a global earliest-priority-point-first (GEPPF) basis. That is, earlier priority points are prioritized over later ones.[2] We assume that ties are broken by task ID (lower IDs are favored).

Note that the parallel task model and the PGM task graph model (presented in Chapter 5) are not equivalent (i.e., a parallel task cannot be modelled as a PGM graph, and vice versa) as the former expresses intra-task precedence constraints while the later expresses inter-task precedence constraints.

[2]GEDF becomes a special case of GEPPF when $d_i = p_i$ holds for each τ_i.

Table 7.1: Summary of notation.

$\tau_{i,h}^{j}$	j^{th} segment of the h^{th} job of task τ_i
$\tau_{i,h}^{j,k}$	k^{th} thread of segment τ_i^j of the h^{th} job of task τ_i
s_i	Number of segments of task τ_i
$e_i^{j,k}$	Worst-case execution cost of thread $\tau_i^{j,k}$
e_i^j	Worst-case execution cost of segment τ_i^j
e_i	Worst-case execution cost of task τ_i
e_i^{min}	Best-case execution cost of task τ_i
v_i^{max}	Maximum number of threads in any segment of task τ_i
v_{max_i}	Maximum number of threads of any segment of the task that has the i^{th} maximum number of threads of any segment among all tasks

7.2 Response Time Bound

We derive a response time bound for GEPPF by comparing the allocations to a task system τ in a PS schedule (Definition 2.5) and an actual GEPPF schedule of interest for τ, both on m processors, and quantifying the difference between the two. Note that parallelism is not considered in the PS schedule. A valid PS schedule exists for τ if $U_{sum} \leq m$ holds. Also note that according to the parallel task model, the per-task utilization could be greater than one. This is a key difference in comparison to most prior work where a PS schedule is considered. A PS schedule for an example parallel task system is shown in Figure 7.1.

Our response time bound derivation focuses on a given task system τ. We order jobs in τ by GEPPF, and break ties by task ID. Let $\tau_{l,j}$ be a job of a task τ_l in τ, $t_d = d_{l,j}$, and S be a GEPPF schedule for τ with the following property.

(P2) The response time of every job $\tau_{i,k}$ of higher priority than $\tau_{l,j}$ is at most $x + p_i + e_i$ in S, where $x \geq 0$.

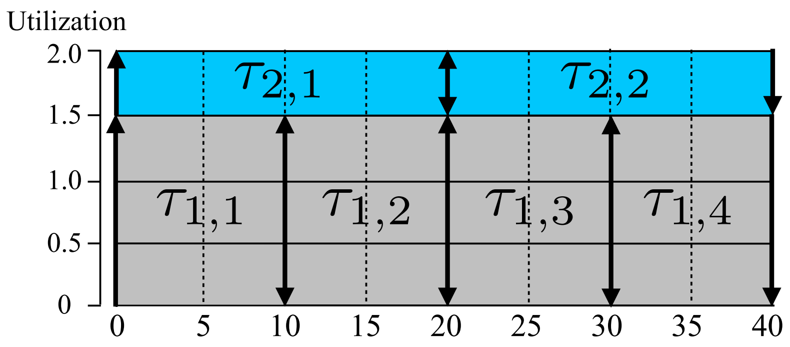


Figure 7.1: PS schedule for a task system containing two tasks. Task τ_1 has a period of 10 time units and a utilization of 1.5. Task τ_2 has a period of 20 time units and a utilization of 0.5. As seen in the PS schedule, intra-task parallelism is not considered and each job completes exactly at its deadline.

Our objective is to determine the smallest x such that the response time of $\tau_{l,j}$ is at most $x + p_l + e_l$. This would by induction imply a response time of at most $x + p_i + e_i$ for all jobs of every task τ_i, where $\tau_i \in \tau$. We assume that $\tau_{l,j}$ finishes after t_d, for otherwise, its response time is trivially no greater than p_l. The steps for determining the value for x are as follows.

1. Determine an upper bound on the work pending for tasks in τ that can compete with $\tau_{l,j}$ after t_d. This is dealt with in Lemmas 7.1 and 7.2 in Section 7.2.1.

2. Determine a lower bound on the amount of work pending for tasks in τ that can compete with $\tau_{l,j}$ after t_d, required for the response time of $\tau_{l,j}$ to exceed $x + p_l + e_l$. This is dealt with in Lemma 7.3 in Section 7.2.2.

3. Determine the smallest x such that the response time of $\tau_{l,j}$ is at most $x + p_l + e_l$, using the above upper and lower bounds. This is dealt with in Theorem 7.1 in Section 7.2.3.

Definition 7.1. **d** $= \{\tau_{i,h} : (d_{i,h} < t_d) \vee (d_{i,h} = t_d \wedge i \leq l)\}$.

d is the set of jobs with deadlines at most t_d with priority at least that of $\tau_{l,j}$. These jobs do not execute beyond t_d in the PS schedule. Note that $\tau_{l,j}$ is in **d**. Also note that jobs not in **d** have lower priority than those in **d** and thus do not affect the scheduling of jobs in **d**. For simplicity, we will henceforth assume that no job not in **d** executes in either the PS or GEPPF schedule. To avoid distracting "boundary cases," we also assume that the schedule being analyzed is prepended with a schedule in which no deadlines are missed that is long enough to ensure that all previously released jobs referenced in the proof exist.

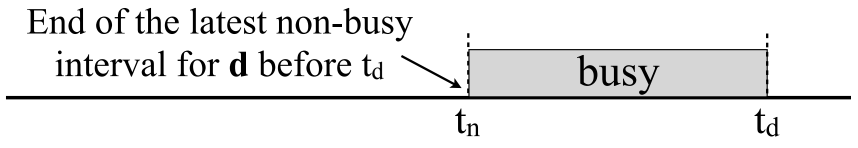


Figure 7.2: Definition of t_n.

According to Property (P2), job $\tau_{l,j-1}$ has a response time of at most $x + p_l + e_l$. Thus, the completion time of $\tau_{l,j-1}$, denoted t_p (p for predecessor), is given by

$$t_p \leq r_{l,j-1} + p_l + x + e_l \leq r_{l,j} + x + e_l = t_d - p_l + x + e_l. \tag{7.1}$$

Definition 7.2. A time instant t is *busy* for a job set J if all m processors execute jobs in J at t. A time interval is busy for J if each instant within it is busy for J.

The following claim follows from the definition of LAG.

Claim 8. If $LAG(\mathbf{d}, t_2, S) > LAG(\mathbf{d}, t_1, S)$, where $t_2 > t_1$, then $[t_1, t_2)$ is non-busy for **d**. In other words, *LAG* for **d** can increase only throughout a non-busy interval.

An interval could be non-busy for **d** only if there are not enough enabled jobs in **d** to occupy all available processors.

Since **d** includes all jobs of higher priority than $\tau_{l,j}$, the competing work for $\tau_{l,j}$ after time t_d is given by the amount of work pending at t_d for jobs in **d**, which is given by $LAG(\mathbf{d}, t_d, S)$.

7.2.1 Upper Bound

In this section, we determine an upper bound on $LAG(\mathbf{d}, t_d, S)$. We first upper bound $lag(\tau_i, t, S)$ ($t \in [0, t_d]$) in Lemma 7.1 below. Then, in Lemma 7.2, we upper bound $LAG(\mathbf{d}, t_d, S)$ by summing individual task lags.

Definition 7.3. Let t_n be the end of the latest non-busy interval for **d** before t_d, if any; otherwise, let $t_n = 0$ (see in Figure 7.2).

By the above definition and Claim 8, we have

$$LAG(\mathbf{d}, t_d, S) \leq LAG(\mathbf{d}, t_n, S). \tag{7.2}$$

Lemma 7.1. $lag(\tau_i, t, S) \leq u_i \cdot x + (u_i + 1) \cdot e_i$ *for any task* τ_i *and* $t \in [0, t_d]$.

Proof. Let $d_{i,k}$ be the deadline of the earliest pending job of τ_i, $\tau_{i,k}$, in the schedule S at time t. If such a job does not exist, then $lag(\tau_i, t, S) = 0$, and the lemma holds trivially. Let γ_i be the amount of work $\tau_{i,k}$ performs before t.

By the selection of $\tau_{i,k}$, we have

$$\begin{aligned} lag(\tau_i, t, S) &= \sum_{h \geq k} lag(\tau_{i,h}, t, S) \\ &= A(\tau_{i,k}, r_{i,k}, t, PS) - A(\tau_{i,k}, r_{i,k}, t, S) \\ &\quad + \sum_{h>k} \big(A(\tau_{i,h}, r_{i,h}, t, PS) \\ &\quad - A(\tau_{i,h}, r_{i,h}, t, S)\big). \end{aligned} \tag{7.3}$$

By the definition of PS, $A(\tau_{i,k}, r_{i,k}, t, PS) \leq e_i$, and $\sum_{h>k} A(\tau_{i,h}, r_{i,h}, t, PS) \leq u_i \cdot \max(0, t - d_{i,k})$ (the latter follows because each such job $\tau_{i,h}$ executes with rate u_i in PS while active, and the sum of the active intervals under consideration is at most $t - d_{i,k}$). By the selection of $\tau_{i,k}$, $A(\tau_{i,k}, r_{i,k}, t, S) = \gamma_i$, and $\sum_{h>k} A(\tau_{i,h}, r_{i,h}, t, S) = 0$. By setting these values into (7.3), we have

$$lag(\tau_i, t, S) \leq e_i - \gamma_i + u_i \cdot \max(0, t - d_{i,k}). \tag{7.4}$$

There are two cases to consider.

Case 1. $d_{i,k} \geq t$. In this case, (7.4) implies $lag(\tau_i, t, S) \leq e_i - \gamma_i \leq u_i \cdot x + (u_i + 1) \cdot e_i$.

Case 2. $d_{i,k} < t$. In this case, because $t \leq t_d$ and $d_{l,j} = t_d$, $\tau_{i,k}$ is not the job $\tau_{l,j}$. Thus, by Property (P2), $\tau_{i,k}$ has a response time of at most $x + p_i + e_i$. Since $\tau_{i,k}$ is the earliest pending job of τ_i at time t, the earliest possible completion time of $\tau_{i,k}$ is at t^+. Thus, we have $t - r_{i,k} < t^+ - r_{i,k} \leq x + p_i + e_i$, which (because $d_{i,k} = r_{i,k} + p_i$) implies $t - d_{i,k} = t - r_{i,k} - p_i < x + e_i$.

Setting this value into (7.4), we have $lag(\tau_i, t, S) < e_i - \gamma_i + u_i \cdot (x + e_i) \leq u_i \cdot x + (u_i + 1) \cdot e_i$. □

Lemma 7.2 below upper bounds $LAG(\mathbf{d}, t_d, S)$. We first define some needed terms.

Definition 7.4. Let U be the sum of the $min(m-1, n)$ largest task utilizations. Let E be the largest value of the expression $\sum_{\tau_i \in \gamma} \big((u_i + 1) \cdot e_i\big)$, where γ denotes any set of $min(m-1, n)$ tasks in τ.

Lemma 7.2. $LAG(\boldsymbol{d}, t_d, S) \leq U \cdot x + E$.

Proof. By (7.2), we have $LAG(\mathbf{d}, t_d, S) \le LAG(\mathbf{d}, t_n, S)$. By summing individual task lags at t_n, we can bound $LAG(\mathbf{d}, t_n, S)$. If $t_n = 0$, then $LAG(\mathbf{d}, t_n, S) = 0$, so assume $t_n > 0$. Consider the set of tasks $\beta = \{\tau_i : \exists \tau_{i,h}$ in **d** such that $\tau_{i,h}$ is enabled at $t_n^-\}$. Given that the instant t_n^- is non-busy, there are not enough enabled jobs in **d** to occupy all m processors. More precisely, there are not enough enabled threads belonging to jobs in **d** to occupy all m processors. There could be at most $min(m-1, n)$ parallel tasks that have enabled jobs at t_n^- since each such parallel task has at least one enabled thread at t_n^-; that is, $|\beta| \le min(m-1, n)$.

If task τ_i does not have pending jobs at t_n^-, then $lag(\tau_i, t_n, S) \le 0$. Therefore, we have

$$
\begin{aligned}
&LAG(\mathbf{d}, t_d, S)\\
&\quad\{\text{by } (7.2)\}\\
\le& LAG(\mathbf{d}, t_n, S)\\
&\quad\{\text{by } (2.5)\}\\
=& \sum_{\tau_i : \tau_{i,h}^{w} \in \mathbf{d}} lag(\tau_i, t_n, S)\\
\le& \sum_{\tau_i \in \beta} lag(\tau_i, t_n, S)\\
&\quad\{\text{by Lemma 7.1}\}\\
\le& \sum_{\tau_i \in \beta} \big(u_i \cdot x + (u_i + 1) \cdot e_i\big).
\end{aligned}
$$

By Definition 7.4 and because $|\beta| \le min(m-1, n)$, we have $LAG(\mathbf{d}, t_d, S) \le \sum_{\tau_i \in \beta} \big(u_i \cdot x + (u_i + 1) \cdot e_i\big) \le U \cdot x + E$. □

7.2.2 Lower Bound

In the following lemma, we determine a lower bound on $LAG(\mathbf{d}, t_d, S)$ that is necessary for the response time of $\tau_{l,j}$ to exceed $x + p_l + e_l$.

Definition 7.5. If any thread of any segment of job $\tau_{i,h}$ is enabled at time t but does not execute at t, and at least one processor is executing some job other than $\tau_{i,h}$ at t, then $\tau_{i,h}$ is *preempted* at t (see Figure 7.3).

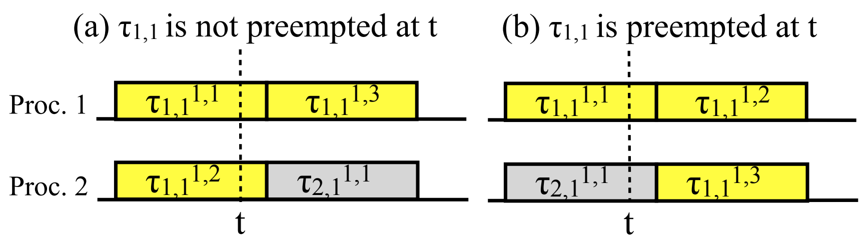


Figure 7.3: Illustration of a preemption. Job $\tau_{1,1}$ has one segment with three parallel threads, executed on two processors. In inset **(a)**, although $\tau_{1,1}^{1,3}$ is enabled but does not execute at time t, $\tau_{1,1}$ is not preempted at t since both processors are executing threads of $\tau_{1,1}$. In inset **(b)**, $\tau_{1,1}$ is preempted by $\tau_{2,1}$ at t.

Definition 7.6. Let v_{max_i} denote the maximum number of threads of any segment of the task that has the i^{th} maximum number of threads of any segment among tasks in τ.

If $\sum_{i=1}^{n} v_{max_i} \leq m$, then each thread of each segment of each task in τ can be executed on a processor without being preempted, which implies that each task $\tau_k \in \tau$ has a bounded response time of $e_k^{min} < x + p_k + e_k$. Thus, we consider the other case, where $\sum_{i=1}^{n} v_{max_i} > m$. Moreover, since we assume that there exists at least one task $\tau_k \in \tau$ with $v_k^{max} \geq 2$ (as discussed in Section 2), we have $v_{max_1} \geq 2$. Thus, if $n > m$, then $\sum_{i=1}^{m} v_{max_i} > m$ holds. Therefore, we have

$$\sum_{i=1}^{min(m,n)} v_{max_i} > m. \tag{7.5}$$

Definition 7.7. Let

$$Q = \begin{cases} 2 & \text{if } v_{max_1} > m \\ \min\{k \mid \sum_{i=1}^{k} v_{max_i} > m\} & \text{if } v_{max_1} \leq m. \end{cases}$$

Q is used in Lemma 7.3 below to obtain a lower bound on $LAG(\mathbf{d}, t_d, S)$; the two conditions in the definition of Q arise because of different subcases considered in the proof of Lemma 7.3. Note that by the above definition and (7.5), we have

$$2 \leq Q \leq min(m,n) \leq m. \tag{7.6}$$

Lemma 7.3. *If the response time of $\tau_{l,j}$ exceeds $x+p_l+e_l$, then $LAG(\mathbf{d}, t_d, S) > Q \cdot x - (m-1) \cdot e_l$.*

Proof. Throughout the proof of this lemma, we assume $\sum_{i=1}^{n} v_{max_i} > m$ and $v_{max_1} \geq 2$ both hold, for reasons discussed above. We prove the contrapositive: we assume that

$$LAG(\mathbf{d}, t_d, S) \quad \leq \quad Q \cdot x - (m-1) \cdot e_l \tag{7.7}$$

holds and show that the response time of $\tau_{l,j}$ cannot exceed $x + p_l + e_l$. Let η_l be the amount of work $\tau_{l,j}$ performs by time t_d in S. Define y as follows.

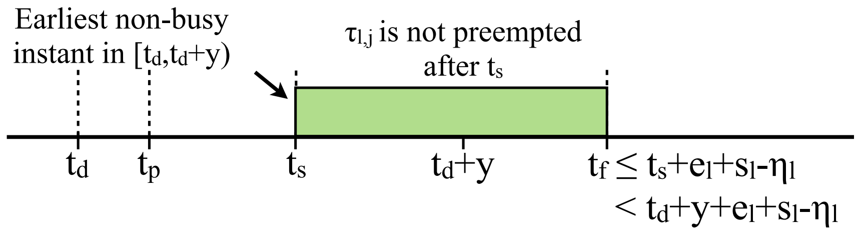
$$y = \frac{Q}{m} \cdot x + \frac{\eta_l}{m} \tag{7.8}$$

Let W be the amount of work due to jobs in **d** that can compete with $\tau_{l,j}$ after $t_d + y$, including the work due for $\tau_{l,j}$. Let t_f be the completion time of $\tau_{l,j}$. We consider two cases.

Case 1. $[t_d, t_d + y)$ *is a busy interval for* **d**. In this case, we have

$$\begin{aligned}
W &= LAG(\mathbf{d}, t_d, S) - my \\
&\quad \{\text{by } (7.7)\} \\
&\leq Q \cdot x - (m-1) \cdot e_l - my \\
&\quad \{\text{by } (7.8)\} \\
&= Q \cdot x - (m-1) \cdot e_l - Q \cdot x - \eta_l \\
&= -(m-1) \cdot e_l - \eta_l \\
&< 0.
\end{aligned}$$

Because GEPPF is work-conserving (i.e., GEPPF idles a processor only when there is no enabled job), at least one processor is busy until $\tau_{l,j}$ completes. Thus, the amount of work performed by the system for jobs in **d** during the interval $[t_d + y, t_f)$ is at least $t_f - t_d - y$. Hence, $t_f - t_d - y \leq W < 0$. Therefore, the response time of $\tau_{l,j}$ is



Earliest non-busy instant in [td,td+y)
τl,j is not preempted after ts
td
tp
ts
td+y
tf ≤ ts+el+sl-ηl
< td+y+el+sl-ηl

Figure 7.4: Subcase 2.1

$$
\begin{aligned}
t_f - r_{l,j} &= t_f - t_d + p_l \\
&< y + p_l \\
&\quad \{\text{by (7.8)}\} \\
&= \frac{Q}{m}\cdot x + \frac{\eta_l}{m} + p_l \\
&\quad \{\text{by (7.6)}\} \\
&\leq x + e_l + p_l.
\end{aligned}
$$

Case 2. $[t_d, t_d + y)$ *is a non-busy interval for* **d**. Let $t_s \geq t_d$ be the earliest non-busy instant in $[t_d, t_d + y)$. Recall (see (7.1)) that t_p is the completion time of job $\tau_{l,j-1}$. We consider three subcases.

Subcase 2.1. $t_p \leq t_s$ *and* $\tau_{l,j}$ *is not preempted within* $[t_p, t_s)$. As illustrated in Figure 7.4, in this case, $\tau_{l,j}$ can start execution at t_s because t_s is non-busy. Since $\tau_{l,j}$ is not preempted within $[t_s, t_p)$, $\tau_{l,j}$ completes by $t_s + e_l - \eta_l$. Thus, because $t_s < t_d + y$, $\tau_{l,j}$ finishes by time

$$
\begin{aligned}
t_s + e_l - \eta_l &< t_d + y + e_l - \eta_l \\
&\quad \{\text{by (7.8)}\} \\
&= t_d + \frac{Q}{m}\cdot x + \frac{\eta_l}{m} + e_l - \eta_l \\
&\quad \{\text{by (7.6)}\} \\
&\leq r_{l,j} + p_l + x + e_l.
\end{aligned}
$$

Subcase 2.2 $t_p \le t_s$ *and* $\tau_{l,j}$ *is preempted within* $[t_p, t_s)$. If $t_f \le y + t_d$, then

$$\begin{aligned} t_f - r_{l,j} &\le y + t_d - r_{l,j} \\ &\quad \{\text{by (7.8)}\} \\ &= \frac{Q}{m} \cdot x + \frac{\eta_l}{m} + p_l \\ &\quad \{\text{by (7.6)}\} \\ &\le x + e_l + p_l. \end{aligned}$$

So assume $t_f > y + t_d$. Let $t_1 > t_s$ be the earliest time when $\tau_{l,j}$ is preempted. As shown in Figure 7.5, by the definition of t_s and t_1, $\tau_{l,j}$ executes throughout $[t_s, t_1)$ without being preempted. Because $\tau_{l,j}$ is preempted at t_1, t_1 is busy with respect to **d**. Let t_2 be the last time $\tau_{l,j}$ resumes execution after being preempted if such a time exists; if such a time does not exist, which implies that $\tau_{l,j}$ is preempted until t_f, then let $t_2 = t_f$ (note that by Definition 7.5, some threads of τ_l^j can execute while τ_l^j is preempted). Within $[t_1, t_2)$, $\tau_{l,j}$ could be preempted multiple times. By Definition 7.5, all such intervals during which $\tau_{l,j}$ is preempted must be busy in order for the preemption to happen. Given that $t_f \le t_2 + e_l - \eta_l$, if $t_2 \le y + t_d$, then $t_f \le y + t_d + e_l - \eta_l$, in which case, because $t_d - r_{l,j} = p_l$, the response time of $\tau_{l,j}$ is

$$\begin{aligned} t_f - r_{l,j} &\le y + p_l + e_l - \eta_l \\ &\quad \{\text{by (7.8)}\} \\ &\le \frac{Q}{m} \cdot x + p_l + e_l \\ &\quad \{\text{by (7.6)}\} \\ &\le x + p_l + e_l, \end{aligned}$$

as required.

If $t_2 > t_d + y$, then the amount of work due to **d** performed within $[t_d, t_d + y)$ is at least $my - (m-1) \cdot \min(e_l, y)$ because all intervals during which $\tau_{l,j}$ is preempted are busy, and $\tau_{l,j}$ can execute for at most e_l time in $[t_d, y + t_d)$. (Within intervals in $[t_s, t_d + y)$ where $\tau_{l,j}$ is not preempted, at least one processor is occupied by $\tau_{l,j}$.) Thus, the amount of work that can compete with $\tau_{l,j}$ after

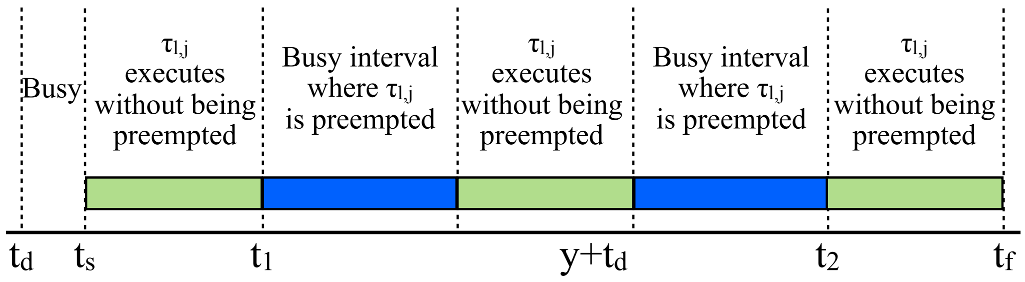

Busy
τl,j executes without being preempted
Busy interval where τl,j is preempted
τl,j executes without being preempted
Busy interval where τl,j is preempted
τl,j executes without being preempted
td
ts
t1
y+td
t2
tf

Figure 7.5: Subcase 2.2

$t_d + y$ is

$$\begin{aligned} W &\leq LAG(\mathbf{d}, t_d, S) - (my - (m-1) \cdot \min(e_l, y)) \\ &\quad \{\text{by (7.7)}\} \\ &\leq Q \cdot x - (m-1) \cdot e_l - (my - (m-1) \cdot \min(e_l, y)) \\ &\leq Q \cdot x - my \\ &\quad \{\text{by (7.8)}\} \\ &= -\eta_l \\ &\leq 0. \end{aligned}$$

Since W is defined to be the amount of work due to jobs in **d** that can compete with $\tau_{l,j}$ after $t_d + y$ and $W \leq 0$, the latest completion time of $\tau_{l,j}$ is at $t_d + y + e_l - \eta_l$. Therefore, the response time of $\tau_{l,j}$ is

$$\begin{aligned} t_f - r_{l,j} &\leq t_d + y + e_l - \eta_l - r_{l,j} \\ &= y + e_l - \eta_l + (t_d - r_{l,j}) \\ &= y + e_l - \eta_l + p_l \\ &\quad \{\text{by (7.8)}\} \\ &= \frac{Q}{m} \cdot x + \frac{\eta_l}{m} + e_l - \eta_l + p_l \\ &\quad \{\text{by (7.6)}\} \\ &\leq x + e_l + p_l. \end{aligned}$$

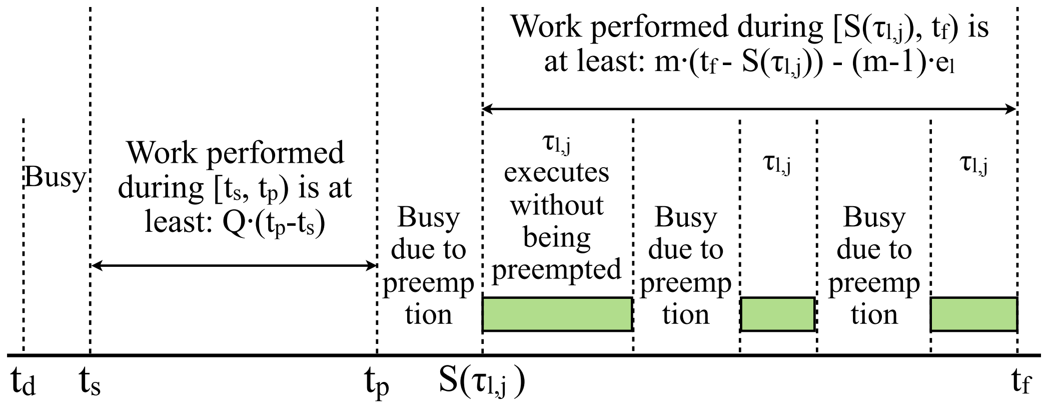

Work performed during [S(τl,j), tf) is at least: m·(tf - S(τl,j)) - (m-1)·el
Busy
Work performed during [ts, tp) is at least: Q·(tp-ts)
Busy due to preemption
τl,j executes without being preempted
Busy due to preemption
τl,j
Busy due to preemption
τl,j
td
ts
tp
S(τl,j)
tf

Figure 7.6: Subcase 2.3

Subcase 2.3: $t_p > t_s$. The earliest time $\tau_{l,j}$ can commence execution is t_p, as shown in Figure 7.6. Let $S(\tau_{l,j})$ be the time when $\tau_{l,j}$ starts execution for the first time. If $\tau_{l,j}$ is not preempted after t_p, then $\tau_{l,j}$ starts execution at t_p and completes no later than $t_p + e_l^{min}$. Thus, we have

$$\begin{aligned}
t_f - r_{l,j} =& t_p + e_l^{min} - r_{l,j} \\
& \quad \{\text{by (7.1)}\} \\
\leq& t_d - p_l + x + e_l + e_l^{min} - r_{l,j} \\
=& x + e_l + e_l^{min} \\
& \quad \{\text{because } e_l^{min} \leq p_l\} \\
\leq& x + e_l + p_l.
\end{aligned}$$

The other possibility is that $\tau_{l,j}$ gets preempted after t_p. Let λ denote the set of tasks including τ_l that have ready jobs in **d** at any time instant within $[t_s, t_p)$.

We now prove that $|\lambda| \geq Q$ holds. By Definition 7.5, in order for $\tau_{l,j}$ to be preempted after t_p, the number of processors required by tasks in λ (note that $\tau_l \in \lambda$) at some time instant after t_p must exceed m. Thus, the maximum total number of threads of tasks in λ that can execute in parallel at the same time must exceed m, which gives

$$\sum_{\tau_i \in \lambda} v_i^{max} > m. \tag{7.9}$$

Thus, by the definition of v_{max_k}, we have $\sum_{k=1}^{|\lambda|} v_{max_k} \geq \sum_{\tau_i \in \lambda} v_i^{max} \overset{\{\text{by (7.9)}\}}{>} m$. By Definition 7.7, we consider two cases: $v_{max_1} \leq m$ and $v_{max_1} > m$. If $v_{max_1} \leq m$, then $|\lambda| \geq Q$ holds. On the other hand, if $v_{max_1} > m$, then although $\sum_{k=1}^{|\lambda|} v_{max_k} > m$ may hold when $|\lambda| = 1$, λ clearly needs to contain at least two tasks in order for $\tau_{l,j}$ to be preempted (namely, τ_l and at least one other task). Thus, $|\lambda| \geq Q$ also holds in this case.

Because $|\lambda| \geq Q$, we know that at least Q tasks have ready jobs in **d** at any time instant within $[t_s, t_p)$, which occupy at least Q processors throughout the interval $[t_s, t_p)$. Thus, the amount of work due to **d** performed in $[t_s, t_p)$ is at least $Q \cdot (t_p - t_s)$. We now complete the proof of Subcase 7.2.2 (and thereby Lemma 7.3).

By the definitions of t_s and t_p, $[t_d, t_s)$ and $[t_p, S(\tau_{l,j}))$ are busy for **d**. As discussed above, the amount of work due to **d** performed in $[t_s, t_p)$ is at least $Q \cdot (t_p - t_s)$. Moreover, the amount of work due to **d** performed in $[S(\tau_{l,j}), t_f)$ is at least $m \cdot (t_f - S(\tau_{l,j})) - (m-1) \cdot e_l$.[3] Thus, we have

$$\begin{aligned} LAG(\mathbf{d}, t_d, S) \quad \geq \quad & m \cdot (t_s - t_d) + Q \cdot (t_p - t_s) + m \cdot (S(\tau_{l,j}) - t_p) \\ & + m \cdot (t_f - S(\tau_{l,j})) - (m-1) \cdot e_l. \end{aligned}$$

By (7.7), we therefore have

$$\begin{aligned} & Q \cdot x - (m-1) \cdot e_l \\ \geq \quad & m \cdot (t_s - t_d) + Q \cdot (t_p - t_s) \\ & + m \cdot (S(\tau_{l,j}) - t_p) \\ & + m \cdot (t_f - S(\tau_{l,j})) - (m-1) \cdot e_l, \end{aligned}$$

which gives,

[3]We apply the same reasoning as used in Subcase 2.2. All intervals in $[S(\tau_{l,j}), t_f)$ during which $\tau_{l,j}$ is preempted are busy, and $\tau_{l,j}$ can execute for at most e_l time in $[S(\tau_{l,j}), t_f)$. (Within such intervals, at least one processor is occupied by $\tau_{l,j}$.)

$$t_f - t_d \;\leq\; \frac{Q}{m} \cdot x + \big(1 - \frac{Q}{m}\big) \cdot (t_p - t_s). \tag{7.10}$$

Also, we have $t_p - t_s \leq t_p - t_d \overset{\{\text{by (7.1)}\}}{\leq} t_d - p_l + x + e_l - t_d = x - p_l + e_l$. Therefore,

$$\begin{aligned} t_f - r_{l,j} =& t_f - t_d + p_l \\ & \{\text{by (7.10)}\} \\ \leq& \frac{Q}{m} \cdot x + \big(1 - \frac{Q}{m}\big) \cdot (x - p_l + e_l) + p_l \\ & \{\text{by (7.6)}\} \\ \leq& x + p_l + e_l. \end{aligned}$$

□

7.2.3 Determining x

Setting the upper bound on $LAG(\mathbf{d}, t_d, S)$ in Lemma 7.2 to be at most the lower bound in Lemma 7.3 will ensure that the response time of $\tau_{l,j}$ is at most $x + p_l + e_l$. By solving for the minimum x that satisfies the resulting inequality, we obtain a value of x that is sufficient for ensuring a response time of at most $x + p_l + e_l$. By Lemmas 7.2 and 7.3, this inequality is

$$\begin{aligned} & U \cdot x + E \\ \leq\;\; & Q \cdot x - (m-1) \cdot e_l. \end{aligned}$$

Solving for x, we have

$$x \geq \frac{E + (m-1) \cdot e_l}{Q - U}. \tag{7.11}$$

If x equals the right-hand side of (7.11), then the response time of $\tau_{l,j}$ will not exceed $x + p_l + e_l$. A value for x that is independent of the parameters of τ_l can be obtained by replacing $(m-1) \cdot e_l$ with $max_l((m-1) \cdot e_l)$ in (7.11).

Theorem 7.1. *With x as defined above, the response time for any task τ_l scheduled under GEPPF is at most $x + p_l + e_l$, provided $U < Q$, where U and Q are defined in Definition 7.4 and Definition 7.7, respectively.*

7.2.4 A Case with No Utilization Loss

The following corollary shows that GEPPF results in no capacity loss for scheduling any parallel task system on two processors.

Corollary 5. For two-processor systems, the response time for any task τ_l scheduled under GEPPF is at most $x + p_l + e_l$, where $x = \dfrac{E + (m-1) \cdot e_l}{Q - max_i(u_i)}$ and $max_i(u_i)$ is the maximum task utilization of tasks in τ.

Proof. If the system only contains one task, then clearly this task, denoted τ_1, has bounded response time, which is given by $e_1^{min} \leq x + p_1 + e_1$. If the system contains more than one task, then by Defs. 7.4 and 7.7 and $m = 2$, we have $U = max_i(u_i)$ and $Q = 2 = m$. Thus, the utilization constraint in Theorem 7.1 becomes $max_i(u_i) < Q = m$, which always holds. □

7.2.5 Cases with Utilization Loss

As shown in Theorem 7.1 and Corollary 5, the utilization constraint $U < Q$ is needed on $m \geq 3$ processors while no utilization constraint is needed on $m = 2$ processors. By Defs. 7.4 and 7.7, in the worst case, $U = U_{sum}$ and $Q = 2$. This implies that in some cases even when m is arbitrarily large, $U_{sum} < 2$ is needed in our analysis. Since no capacity loss can be achieved on two processors as shown in Corollary 5, we can schedule any parallel task system with $U_{sum} = 2$ on only two processors (i.e., leave the other $m - 2$ processors idle if $m > 2$). Thus, in the worst case, $U_{sum} \leq 2$ (rather than $U_{sum} < 2$) is needed under our analysis for any parallel task system to have bounded response times for $m \geq 3$ processors. To discern how severe such constraints must fundamentally be, we next show that for any $m \geq 3$, there exists a parallel task system with a total utilization of $2 + \sigma$ that has unbounded response times, where σ can be an arbitrarily small value. This proves that utilization constraints are fundamental for parallel task systems scheduled on $m \geq 3$ processors. (Note that this task set also violates our derived utilization constraint.)

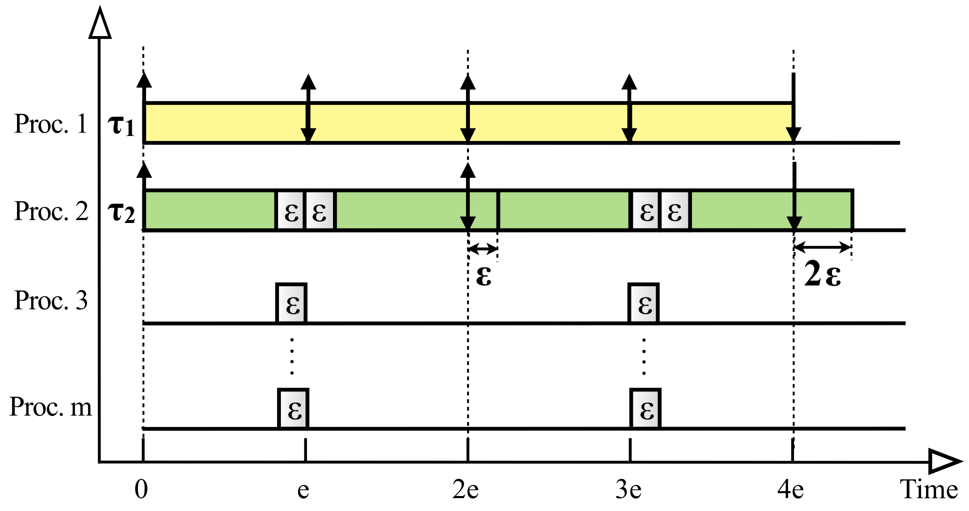


Figure 7.7: The worst-case parallel task set.

Worst-case parallel task set. Consider a parallel task system containing two parallel tasks. Task τ_1 has only one segment that contains one thread with an execution cost of e time units, and τ_1 has a period of e time units. Thus, τ_1 has a utilization of 1.0. Task τ_2 has three segments, where the first segment contains one thread with an execution cost of $e - \varepsilon$ time units, where ε can be an arbitrarily small value, the second segment contains m parallel threads, each of which has an execution cost of ε time units, and the third segment contains one thread with an execution cost of e time units. τ_2 has a period of $2e$ and a utilization of $\frac{e - \varepsilon + m \cdot \varepsilon + e}{2e} = 1 + \frac{(m-1)}{2e} \cdot \varepsilon$. Thus, this task set has a total utilization of $2 + \frac{(m-1)}{2e} \cdot \varepsilon$, or rather $2 + \sigma$, where $\sigma = \frac{(m-1)}{2e} \cdot \varepsilon$ can be arbitrarily small.

Figure 7.7 shows the GEPPF schedule of this parallel task system on any $m \geq 3$ processors. It is clearly seen that task τ_2's response time grows unboundedly regardless of m.

7.2.6 Optimization

The capacity loss seen in the utilization constraint $U < Q$ is mainly caused by a small value of Q. (Note that by Definition 7.4, U is completely determined by the tasks' execution costs and periods, which are fixed parameters.) By Definition 7.7, Q depends on v_{max_i} $(1 \leq i \leq n)$. If the value of v_{max_i} can be decreased, then the value of Q is increased.

To decrease v_{max_i} $(1 \leq i \leq n)$, we can seek to decrease v_k^{max} (the maximum number of threads in any segment of τ_k) for each task $\tau_k \in \tau$. This can be done by splitting any segment of τ_k with a large number of threads into multiple sequential sub-segments, each of which has a smaller number of threads, thus decreasing v_k^{max}. Notice that a critical constraint to enable such splittings is to ensure that $e_k^{min} \leq p_k$ still holds for any task τ_k after splitting; otherwise, response times may grow unboundedly. Thus, for each task, we need to determine the maximum degree to which its segments can be split.

We propose algorithm *Q-Optimization* to increase Q for any given parallel task system τ by decreasing v_k^{max} for each task $\tau_k \in \tau$, as discussed above. The pseudo-code for this algorithm is given in Figures 7.8–7.10. Applying this algorithm can also reduce response time bounds, as seen in Section 5.2.7.

Algorithm description. Algorithm *Q-Optimization* seeks to increase the value of Q by decreasing the maximum number of threads in any segment of each task. In the code, v_i^{max} (v_i^{secmax}) denotes the number of threads in the segment of τ_i with the largest (second largest) number of threads. Note that if all segments of task τ_i contain the same number of threads, then $v_i^{secmax} = 0$.

We first describe the function *SPLIT* (shown in Figure 7.9) used in the main algorithm (shown in Figure 7.8). *SPLIT*(τ_k,H) splits the segments with the maximum number of threads into a number of sequential sub-segments, each with at most H threads (lines 1-4 in function *SPLIT*). (Note that several variables used in this function are defined in algorithm *Q-Optimization* shown in Figure 7.8.) Threads are assigned to each of these sub-segments in smallest-thread-ID-first order, until either a sub-segment contains H threads or all threads have been assigned. Then in line 5, function *COMBINE* (shown in Figure 7.10) seeks to combine any two sub-segments that originally belong to the same segment into one segment if the sum of the number of threads in both sub-segments is no greater than the maximum number of threads of any segment. Finally, function *SPLIT* calculates e_k^{min} (line 6 in function *SPLIT*) using the method we discussed in Section 2.

Now we describe algorithm *Q-Optimization* in detail. First we make two important observations. ***(i)*** For any task τ_i that contains at least two segments having a different number of threads, we desire to reduce the number of threads of its segments that contain the maximum number of threads among all segments of τ_i to no less than v_i^{secmax}. Further reductions do not reduce v_i^{max}. ***(ii)*** For any task τ_i containing at least two segments that have the same number of threads, we desire to reduce the number of threads of such segments by the same amount. Reducing any such segment's thread count by a greater amount than the others does not reduce v_i^{max}.

Motivated by these two observations, the algorithm first executes *SPLIT*(τ_k, v_k^{secmax}), which splits each of the segments in τ_k that have the maximum number of threads into a sequential number of sub-segments, each with at most v_k^{secmax} threads. After such a splitting, if $e_k^{min} < p_k$ and $v_k^{max} \neq 1$ (lines 5-7 in algorithm *Q-Optimization*), then we set the *further-split-flag* to be true, which implies that there is still the potential for us to split τ_k to further reduce v_k^{max}.

On the other hand, if $e_k^{min} > p_k$ after such a splitting (line 8 in algorithm *Q-Optimization*), then it implies that such a splitting causes e_i^{min} to exceed τ_k's period (which causes τ_k to have unbounded response times) and is thus invalid. Since this splitting is invalid, we restore the task structure to the one before the splitting (lines 9-10 in algorithm *Q-Optimization*). Thus, we now know that it is impossible to split segments in τ_k to reduce v_k^{max} to equal v_k^{secmax}. However, by splitting, we might still be able to reduce v_k^{max} to some number between v_k^{secmax} and v_k^{max} (realized by lines 12-14 in algorithm *Q-Optimization*). Note that the minimum value of such a number is given by C_k (for otherwise it would have been possible to reduce v_k^{max} to equal v_k^{secmax} given that C_k is initially $v_k^{secmax} + 1$.). Therefore, starting from C_k, the algorithm uses the *SPLIT* function and compares the resulting e_k^{min} with p_k to determine whether any such splitting is valid (lines 11-18 in algorithm *Q-Optimization* using the logic discussed above).

Optimization example. Since we seek to decrease v_k^{max} for each task τ_k in any given task system using the same optimization algorithm, we use one example task τ_1 to illustrate the idea. In this example, $m = 4$ and τ_1 originally has five segments, as illustrated in Figure 7.11(a). The notation $\tau_1^{i,j}(e)$ in Figure 7.11 denotes that thread $\tau_1^{i,j}$ has an execution cost of e time units. τ_1 has a period of 18 time units, thus $p_1 = 18$.

Because we want to decrease v_1^{max}, we first try to decrease the number of threads of segments in τ_1 that have the largest number of threads, which are τ_1^2 and τ_1^3 (realized by executing algorithm *Q-Optimization*). Therefore, according to observations (i) and (ii) discussed above, we split each of τ_1^2 and τ_1^3 into two sequential sub-segments, one with three threads and the other one with one thread (realized by executing line 4 in algorithm *Q-Optimization*), as shown in Figure 7.11(b) (note that in the figure updated segment notations are used after each splitting). After this splitting, we obtain

```
ALGORITHM: Q-OPTIMIZATION
  further-split-flag: BOOLEAN
  $C_k$: INTEGER, INITIALLY $v_k^{secmax} + 1$
  $v_k^{max}$: INTEGER, DEFINED IN SECTION 4
  $v_k^{secmax}$: INTEGER, DEFINED IN SECTION 4
  $h$: INTEGER, INITIALLY $h := 1$
  $A_k$: THE SET OF SEGMENTS IN $\tau_k$ THAT HAVE $v_k^{max}$ THREADS

for each parallel task $\tau_k \in \tau$
    further-split-flag := false
    do
        SPLIT($\tau_k$,$v_k^{secmax}$)
        if $e_k^{min} < p_k$
            if $v_k^{max} \neq 1$
                then further-split-flag := true
        else if $e_k^{min} > p_k$
            Restore the structure of $\tau_k$ to the one before the last splitting and
            update segment notations, $A_k$, $v_k^{max}$, $v_k^{secmax}$, and $C_k$ accordingly
            while $C_k < v_k^{max}$
                SPLIT($\tau_k$,$C_k$)
                if $e_k^{min} \leq p_k$
                    break
                else
                    Restore the structure of $\tau_k$ to the one before the last splitting and
                    update segment notations, $A_k$, $v_k^{max}$, $v_k^{secmax}$, and $C_k$ accordingly
                    $C_k := C_k + 1$
    while further-split-flag = true
```

Figure 7.8: Algorithm *Q-Optimization*.

```
FUNCTION: SPLIT($\tau_k$,$H$)
1  for each segment $\tau_k^j \in A_k$
2      Split $\tau_k^j$ into $\left\lceil \frac{v_k^j}{H} \right\rceil$ sequential sub-segments, each with at most $H$ threads, and
3          assign threads to each sub-segment in smallest-thread-ID-first order and
4          update segment notations, $A_k$, $v_k^{max}$, $v_k^{secmax}$, and $C_k$ accordingly
5  COMBINE($\tau_k$)
6  Calculate $e_k^{min}$
```

Figure 7.9: Function *SPLIT*.

```
FUNCTION: COMBINE($\tau_k$)
1  while $\tau_k^h$ exists
2      if $\tau_k^h$ and $\tau_k^{h+1}$ (if any) are sub-segments that originally belong to the same segment, and $v_k^h + v_k^{h+1} \leq v_k^{max}$
3          then combine $\tau_k^h$ and $\tau_k^{h+1}$ into one segment and
4          update segment notations, $A_k$, $v_k^{max}$, $v_k^{secmax}$, and $C_k$ accordingly
5      else
6          $h := h + 1$
```

Figure 7.10: Function *Combine*.

$e_1^{min} = 15 < p_1 = 18$ (we apply the same method discussed in Section 2 to obtain e_1^{min}). Thus, this splitting is valid (as verified in lines 5-7 in algorithm *Q-Optimization*). Now we obtain a task τ_1 in which segments τ_1^2, τ_1^4, and τ_1^6 have the largest number of threads (three threads per segment), while segment τ_1^1 has the second largest number of threads (one thread per segment). Therefore, we again try to reduce the number of threads of τ_1^2, τ_1^4, and τ_1^6 to no less than the number of threads of τ_1^1. This can be achieved by splitting each of these three segments into three sequential segments, each of which contains only one thread (again, realized by executing line 4 in algorithm *Q-Optimization*). However, after such a splitting, we have $e_1^{min} = 28 > p_1 = 18$. Thus, such a splitting is invalid (as verified in lines 8-10 in algorithm *Q-Optimization*).

Therefore, our goal now is trying to reduce the number of threads of τ_1^2, τ_1^4, and τ_1^6 to a smallest possible number, which is two threads per segment in this case (realized by executing lines 11-18 in

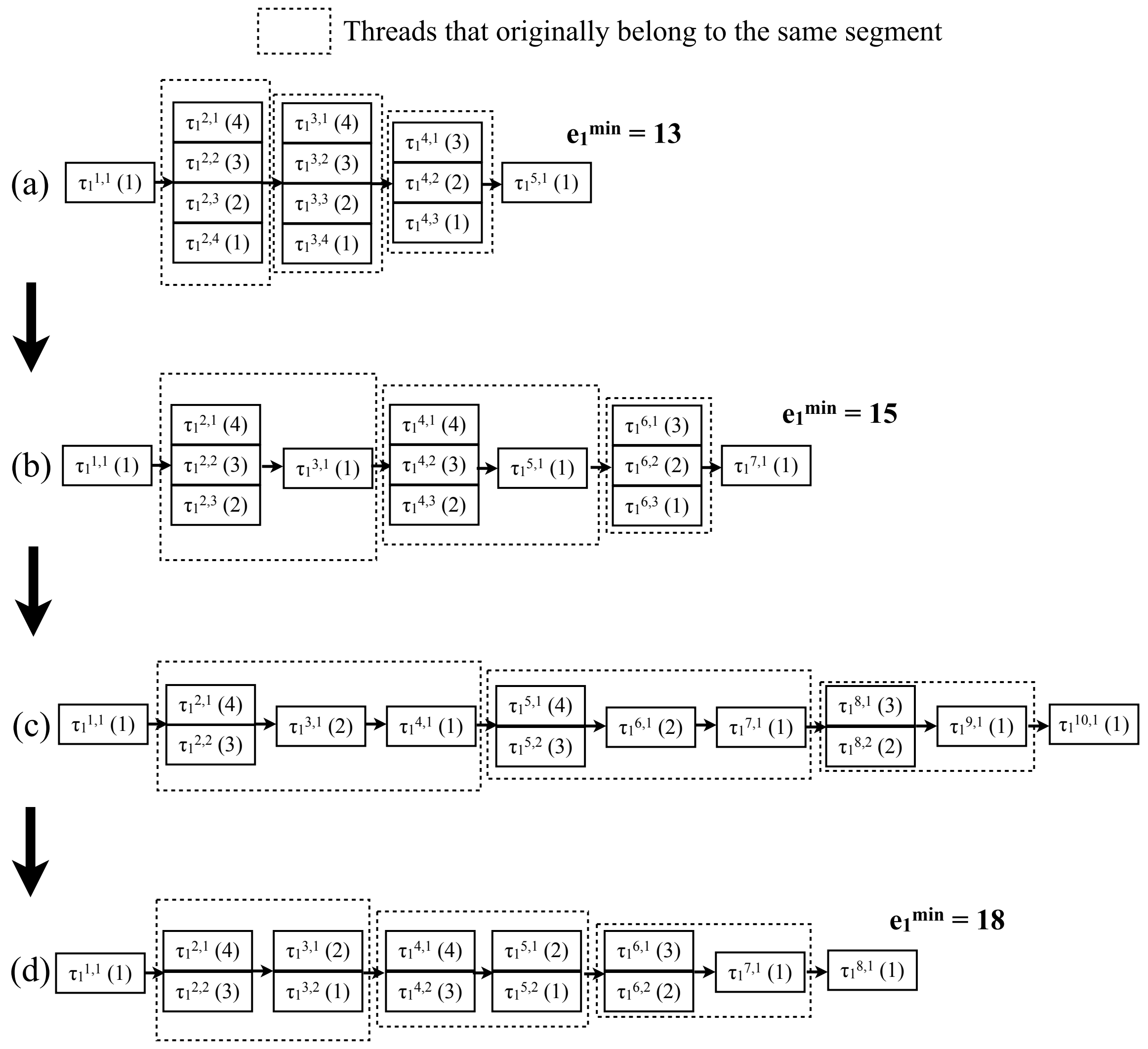


Figure 7.11: Illustration of the optimization algorithm.

algorithm *Q-Optimization*). As shown in Figure 7.11(c), we split each of τ_1^2, τ_1^4, and τ_1^6 into two sequential sub-segments, one with two threads and another one with one thread. Also notice that after this splitting, τ_1^3 and τ_1^4 originally belonged to the same segment, and τ_1^6 and τ_1^7 originally belonged to the same segment. Since combining τ_1^3 and τ_1^4 (as well as τ_1^6 and τ_1^7) into one sub-segment does not increase v_1^{max}, we combine them in such a way to decrease e_1^{min} (realized by executing function *COMBINE*), as illustrated in Figure 7.11(d). After this splitting, we have $e_1^{min} = 18 = p_1$. Thus, we cannot split segments any further (as verified in lines 13-14 in algorithm *Q-Optimization*), and we successfully reduce v_1^{max} from 4 to 2.

7.2.7 Experimental Evaluation

In this section, we describe experiments conducted using randomly-generated parallel task sets to evaluate the applicability of the response time bound in Theorem 7.1. Moreover, we evaluate whether the optimization algorithm can effectively improve schedulability (with respect to bounded response times) and reduce the bound.

Experimental setup. In our experiments, parallel task sets were generated as follows. The number of segments of each task was uniformly distributed over [1, 30]. The number of threads of each segment was distributed differently for each experiment using three uniform distributions: [1, m/2] (*low parallelism*), [m/2, m] (*high parallelism*), and [1, m] (*random parallelism*), where m is the number of processors. The execution cost of each thread was uniformly distributed over [1*ms*,100*ms*]. The worst-case execution cost e_i and the best-case execution cost e_i^{min} of each parallel task τ_i were then calculated using the approach discussed in Section 2. Then, for each task τ_i, its period was uniformly distributed over [e_i^{min}, $e_i^{min} + e_i$], and its utilization was calculated using e_i and p_i. We also varied the system utilization U_{sum} within {0.1, 0.2, ..., m }. For each U_{sum}, 1,000 parallel task sets were generated for systems with four, six, and eight processors. Each such parallel task set was generated by creating parallel tasks until total utilization exceeded U_{sum}, and by then reducing the last task's utilization so that the total system utilization equalled U_{sum}. For each generated system, we first checked schedulability (i.e., the ability to ensure bounded response times) and the magnitude of response time bounds using Theorem 7.1. Then, for each such generated system, we applied the optimization algorithm and re-checked schedulability and response time bounds. In all figures and tables presented in this section, we let "Original" and "Optimization" denote results under the original analysis and results after applying the optimization algorithm *Q-Optimization*.

Results. The schedulability results that were obtained on four-, six-, and eight-processor systems with different degrees of intra-task parallelism are shown in Figures 7.12-7.20, respectively. In these figures, the x-axis denotes U_{sum} and each curve plots the fraction of the generated parallel task sets the corresponding approach successfully scheduled, as a function of U_{sum}. As seen, our analysis can provide reasonable schedulability. For example, as shown in Figure 7.12, on four processors with low parallelism, all parallel task sets have bounded response times until U_{sum} reaches 3.0 and more than 40$\%$ of the task sets still have bounded response times when U_{sum} reaches 3.3. Moreover, the optimization algorithm is able to effectively improve schedulability, especially when the processor count is large or the intra-task parallelism is high. For example, as illustrated in Figure 7.20, on eight processors with random parallelism, the optimization algorithm can improve schedulability by more than 400$\%$ in many cases (e.g., when $U_{sum} = 3.0$). Such improvements tend to increase with increasing processor count or increasing parallelism. This is because when m becomes larger or the number of threads per segment increases, it is easier to increase Q by applying the optimization algorithm, which is intuitive according to the definition of Q. Note that, when schedulability drops significantly, it does so at an integral values of U_{sum}. For example, as seen in Figure 7.12, when U_{sum} reaches 3.0, schedulability drops from 100$\%$ to less than 50$\%$ under Original. This is because when U_{sum} reaches 3.0, by Definition 7.4, U may also equal 3.0 since some parallel tasks very likely have utilization greater than 1.0. Thus, Q has to be 4.0 instead of 3.0 (when the utilization is below 3.0) in order for the utilization constraint $Q > U$ to hold; this obviously makes this constraint much more severe.

Figures 7.21–7.23 show the computed response time bounds using Theorem 7.1 under Original and Optimization. To better illustrate the magnitude of the response time bounds, we plot *relative response time bounds*. A task's relative response time bound is given by the ratio of its response time bound divided by its period. The data in Figure 7.12 shows average relative response time bounds obtained by considering all tasks in certain selected task sets. Such task sets were selected by considering values of U_{sum} for which 100$\%$ schedulability can be ensured, which guarantees all such task sets valid response time bounds. For example, on four processors, we calculated the average relative response time bound over task sets whose utilizations are within $[0.1, 3)$ (all such task sets are schedulable and thus have valid response time bounds). As seen in the figure, our analysis can achieve reasonable response time bounds. For example, as shown in Figure 7.21, on four processors with low parallelism, the average relative response time bound is around nine. The benefit of the optimization algorithm is apparent. For example, as illustrated in Figure 7.22, on eight processors with high parallelism, we can reduce the average relative response time bound from around 33 to less than 18. This is because applying the optimization algorithm only increases Q and does not change other values in the response time bound expression shown in Theorem 7.1.

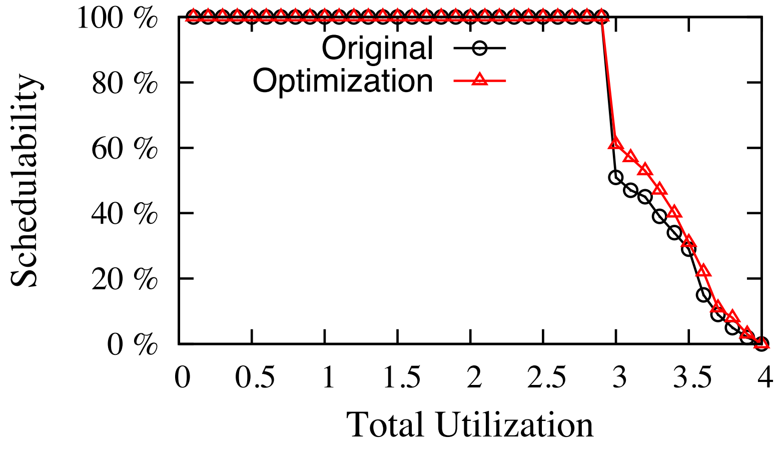


Figure 7.12: Schedulability: m = 4, low parallelism.

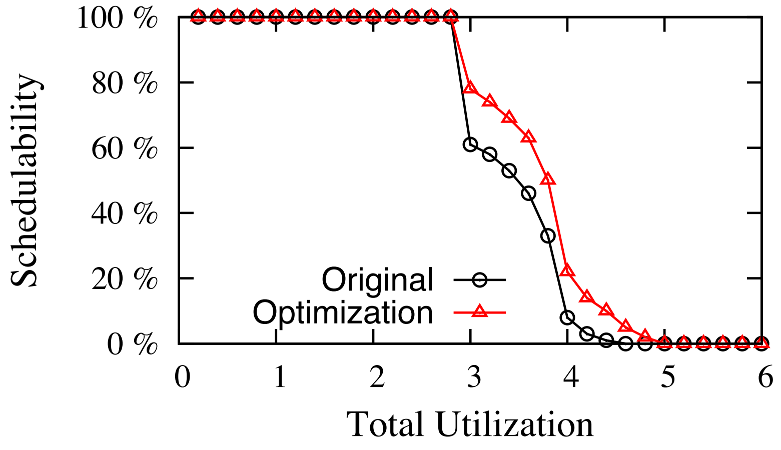


Figure 7.13: Schedulability: m = 6, low parallelism.

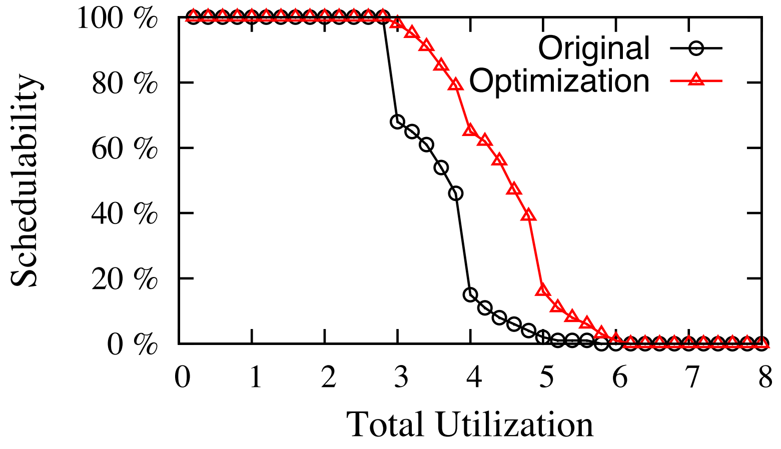


Figure 7.14: Schedulability: m = 8, low parallelism.

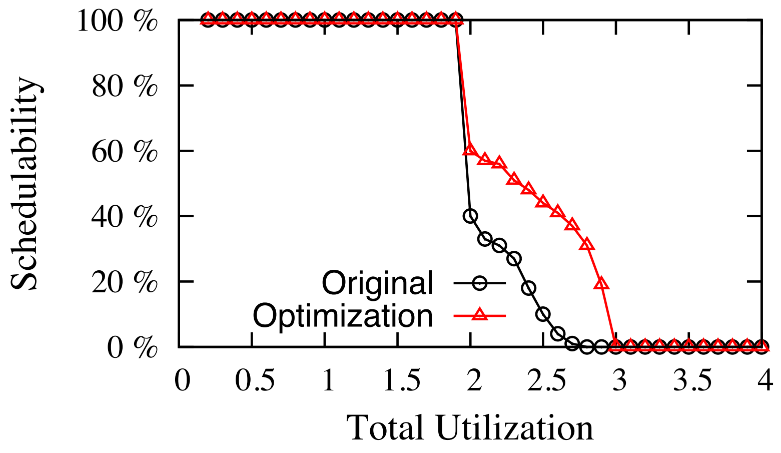


Figure 7.15: Schedulability: m = 4, high parallelism.

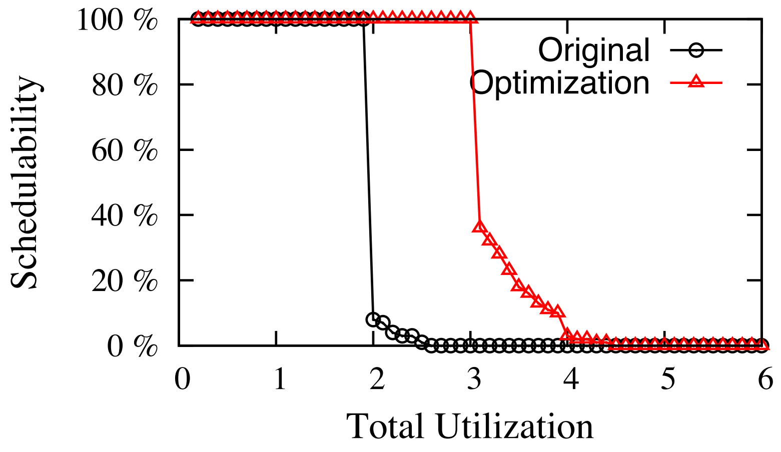


Figure 7.16: Schedulability: m = 6, high parallelism.

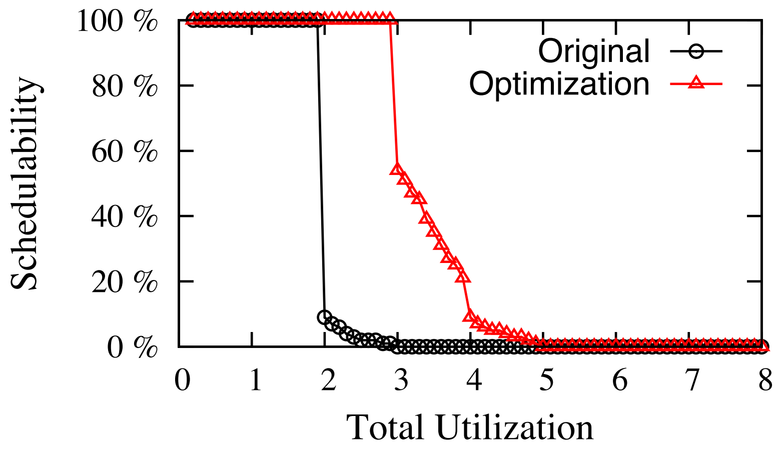


Figure 7.17: Schedulability: m = 8, high parallelism.

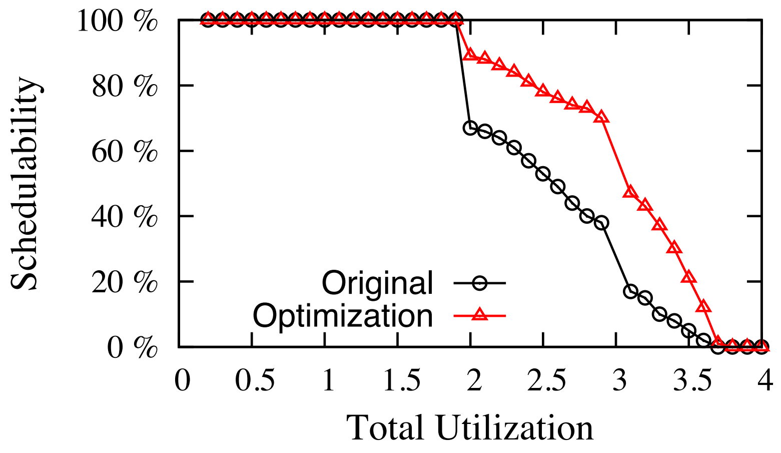


Figure 7.18: Schedulability: m = 4, random parallelism.

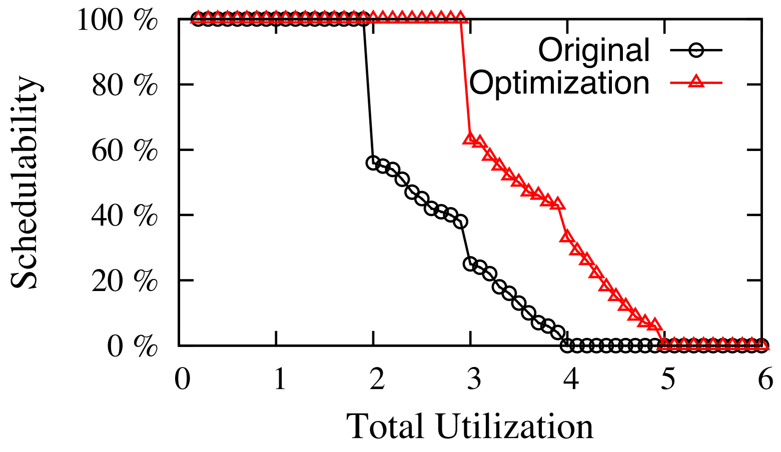


Figure 7.19: Schedulability: m = 6, random parallelism.

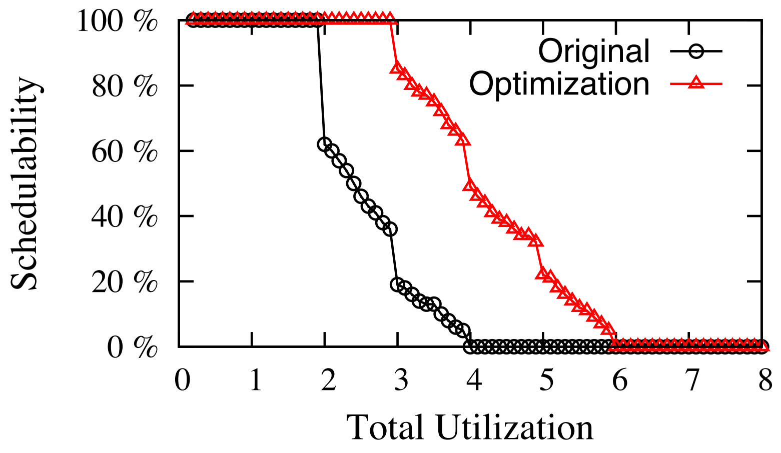


Figure 7.20: Schedulability: m = 8, random parallelism.

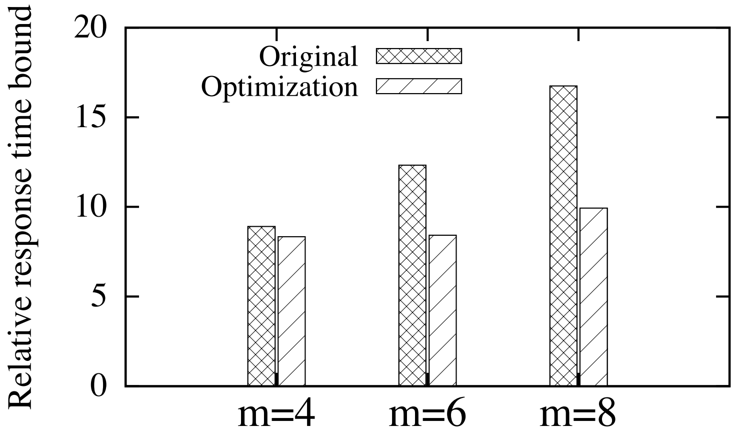


Figure 7.21: Response time bounds: low parallelism.

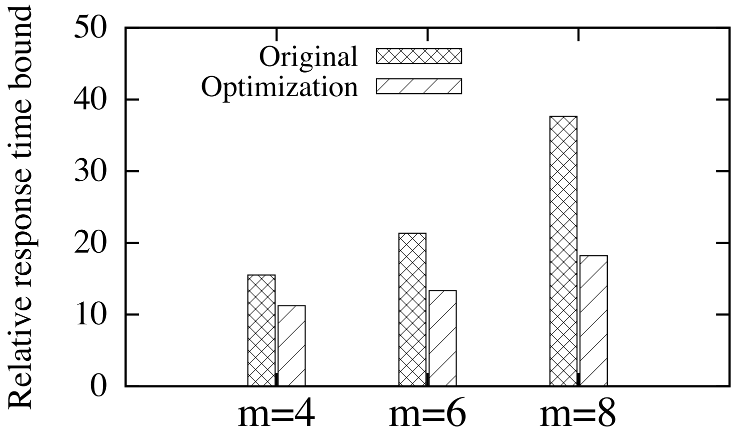


Figure 7.22: Response time bounds: high parallelism.

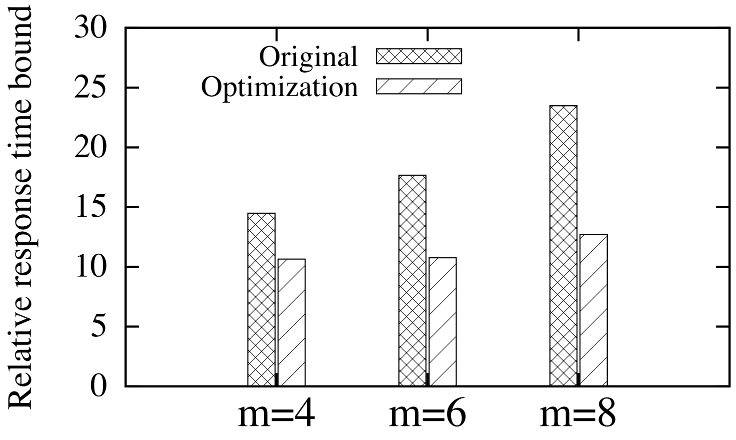


Figure 7.23: Response time bounds: random parallelism.

7.3 Chapter Summary

We have presented schedulability analysis for sporadic parallel task systems under GEPPF scheduling. The proposed analysis shows that such systems can be efficiently supported on multiprocessors with bounded response times. In experiments presented herein, our analysis is proved to provide good performance with respect to both schedulability and response time bounds. In future work, it would be interesting to investigate more practical parallel task models where data is communicated among segments within a parallel task. Moreover, allowing more general parallel execution patterns such as cycles would expand the applicability of our results. Last but not the least, as shown by the worst-case parallel task set (presented in Section 7.2.5), there exist parallel task systems with small utilizations that cannot be scheduled on a multiprocessor with an arbitrary number of processors. This fact implies that utilization might not be the best metric to evaluate schedulability for parallel task sets. In future work, we plan to investigate other metrics such as the ratio between the maximum degree of parallelism among tasks and the number of processors.

CHAPTER 8

Conclusions and Future Work

The main objective of the research reported in this dissertation is to enable real-world real-time applications containing complex runtime behaviors to be efficiently supported on multiprocessors. To achieve this objective, we have designed new multiprocessor real-time scheduling algorithms and efficient schedulability tests for tasks that may contain self-suspensions, graph-based precedence constraints, non-preemptive sections, and parallel execution segments. Our major goal is to avoid over-provisioning systems and to reduce the number of needed hardware components to the extent possible while providing temporal correctness guarantees. In the following, we first summarize our results in Section 8.1, and then discuss open questions and future work in Section 8.2.

8.1 Summary of Results

In Chapter 1, we formulated the thesis statement that this dissertation strived to support, as stated below.

> *Capacity loss can be significantly reduced on multiprocessors while providing non-trivial SRT and HRT guarantees for sophisticated real-time applications that contain common types of runtime behaviors including self-suspensions, graph-based precedence constraints, non-preemptive sections, and parallel execution segments by designing new real-time scheduling algorithms and developing new schedulability tests.*

In support to this thesis statement, our research makes novel contributions in advancing the state-of-the-art to support more sophisticated but practical applications in real-time systems. In the following, we briefly recapitulate the key points of Chapters 3–7.

8.1.1 Multiprocessor Schedulability Tests for Globally-Scheduled Self-Suspending Task Systems

The first set of contributions we summarize is two multiprocessor GEDF schedulability tests proposed in Chapter 3 for SRT self-suspending task systems. The approach presented in Section 3.2 serves as the first attempt at dealing with self-suspensions on globally-scheduled SRT multiprocessors.

New suspension-aware global SRT schedulability tests. To deal with self-suspensions, the common suspension-oblivious approach, which simply integrates suspensions into per-task WCET requirements, is rather pessimistic. In order to improve upon this approach, we have proposed in Chapter 3 two new suspension-aware schedulability tests and one effective technique that can further improve schedulability. Specifically, in the first schedulability test (presented in Section 3.2), we derived a general tardiness bound, which is applicable to either GEDF or GFIFO, that expresses tardiness as a function of task parameters. This bound shows that task systems consisting of both self-suspending tasks and ordinary computational tasks that do not suspend can be supported with bounded tardiness if

$$\xi_{max} < 1 - \frac{U^s_{sum} + U^c_L}{m}, \tag{8.1}$$

where ξ_{max} is the maximum suspension ratio defined in Definition 3.11, U^s_{sum} is the total utilization of all self-suspending tasks in the system, and U^c_L is the total utilization of the $m - 1$ computational tasks of highest utilization.

From Equation (8.1), we see that significant utilization loss may occur when ξ_{max} is large. Thus, in order to improve the utilization bound, it is desirable to decrease the value of ξ_{max}. Motivated by this, we showed that ξ_{max} can be effectively decreased by treating *partial* suspensions as computation. That is, we consider intermediate choices between the two currently-available extremes of treating *all* (as is commonly done) or *no* suspensions as computation. Our technique (presented in Section 3.3) can find the amount of the suspension time of each task that should be treated as computation in order for the task system to satisfy the utilization constraint and thus become schedulable. Experiments presented in Section 3.3.4 demonstrated the effectiveness of the proposed technique.

Although our proposed first suspension-aware analysis (presented in Section 3.2) improves upon the suspension-oblivious approach for many task systems, it unfortunately does not fully address the root cause of pessimism due to suspensions, and thus may still cause significant utilization loss. The worst-case scenario that serves as the root source of pessimism in this analysis is the following: *all n self-suspending tasks have jobs that suspend at some time t simultaneously, thus causing t to be non-busy*. Motivated by this, we derived a much improved schedulability test (presented in Section 3.4) that shows that any given sporadic self-suspending task system is schedulable under GEDF scheduling with bounded tardiness if $U_{sum} + \sum_{i=1}^{m} v^j \leq m$ holds, where U_{sum} is the total system utilization and v^j is the j^{th} maximum suspension ratio, where a task's *suspension ratio* is given by the ratio of its suspension time over its period. We showed in Section 3.4.6 that our derived schedulability test theoretically dominates the suspension-oblivious approach [86], and our previously proposed suspension-aware analysis (presented in Section 3.2) if every task in the system is a self-suspending task. As demonstrated by experiments in Section 3.4.7, our proposed test significantly improves upon prior methods with respect to schedulability, and is often able to guarantee schedulability with little or no utilization loss while providing low predicted tardiness.

Multiprocessor HRT schedulability tests for self-suspending task systems under GEDF and GTFP scheduling. Although the techniques presented in Section 3.2-3.4 can handle the SRT case, how to support HRT sporadic self-suspending task systems on multiprocessors (other than using the suspension-oblivious approach) remains as an open issue. As the first attempt at solving this problem, we presented in Chapter 4 global suspension-aware multiprocessor schedulability analysis techniques for HRT arbitrary-deadline sporadic self-suspending task models under both GEDF and global TFP scheduling. Our analysis shows that schedulability is much less impacted by suspensions than computation on multiprocessors. For any job, suspensions of jobs with higher priorities do *not* contribute to the competing work that may prevent the job from executing (while computation does). Indeed, as shown by experiments in Section 4.2.4, HRT schedulability tests based on our new analysis proved to be superior to the method of treating all suspensions as computation. However, as will be discussed in Section 8.2.2, this analysis could still be quite pessimistic in certain cases, and one of our most important future work is to improve upon it by deriving new schedulability tests for HRT sporadic self-suspending task systems.

8.1.2 Multiprocessor Scheduling of SRT PGM Task Systems

Our work pertaining to the scheduling of SRT PGM task systems on multiprocessors and in distributed systems is presented in Chapter 5.

Supporting PGM task systems on multiprocessors. In Section 5.1, we proposed a variant of GEDF that can be used on multiprocessors as the underlying scheduling algorithm for SRT PGM task systems. The associated analysis shows that the main complicating factor in supporting graph-based dependencies in a multicore setting is workload burstiness, which may cause deadline burstiness. Thus, we proposed a technique that effectively postpones deadlines of certain tasks to avoid such burstiness without affecting timing correctness. We showed that the proposed solution is able to achieve no capacity loss for executing PGM task graphs in multiprocessor systems while providing timing correctness guarantees. That is, any PGM task system τ is SRT schedulable if $u_{sum}(\tau) \leq m$ holds.

Supporting PGM task systems in a distributed system. In Section 5.2, we further showed how to extend the above PGM approach for application in distributed systems comprised of clusters of processors, where scheduling within each cluster is global. Our main contribution in that work was to develop a method for assigning tasks to clusters so as to minimize the amount of data movement across clusters. Once tasks are so assigned, those in each cluster can be scheduled globally as described above (with some slight adjustments due to potential dependencies across clusters). Note that this same task-assignment method can be applied in a fully partitioned system (where each cluster is just one processor), in which case Goddard's orignal work (with some slight modifications) can be applied on each processor. Although our focus in this work was distributed systems, the same techniques can be applied to schedule tasks on a (large) multicore platform in a clustered fashion.

8.1.3 Multiprocessor SRT Scheduling of Task Systems with Mixed Types of Complex Runtime Behaviors

Besides handling self-suspensions (in Chapters 3 and 4) and graph-based precedence constraints (in Chapter 5) independently, in Chapter 6, we also investigated how to support sophisticated real-time task systems containing multiple types of such complex runtime behaviors. We considered this issue in the context of SRT sporadic task systems in which non-preemptive sections, self-suspensions, and graph-based precedence constraints co-exist. Specifically, we addressed the problem of deriving conditions under which bounded tardiness can be ensured when *all* of the above-mentioned behaviors—non-preemptive sections, graph-based precedence constraints, and self-suspensions—are allowed. In considering this problem, we focused specifically on GEDF. Our main result is a transformation process that converts any implicit-deadline periodic task system with self-suspensions, graph-based precedence constraints, and non-preemptive sections into a simpler system with only suspensions. In the simpler system, each task's maximum job response time is at least that of the original system. This result allows tardiness bounds to be established by focusing only on the impacts of suspensions. It thus enables prior results on systems with suspensions (presented in Chapter 3 and 4) to be applied to derive tardiness bounds for more complex systems, as scheduled by GEDF.

8.1.4 A Tardiness Bound for Multiprocessor Real-Time Parallel Tasks

Finally, in Chapter 7, we presented a tardiness bound for GEDF-scheduled sporadic parallel task systems on multiprocessors. Our analysis shows that on a two-processor platform, no utilization loss results for any parallel task system. Despite this special case, on a platform with more than two processors, utilization constraints are needed. To discern how severe such constraints must fundamentally be, we presented a parallel task set with a utilization of approximately 2.0 that is unschedulable on any number of processors. This task set violates our derived constraint and has unbounded response times. The impact of utilization constraints can be lessened by restructuring tasks to reduce intra-task parallelism. We proposed optimization techniques that can be applied to determine such a restructuring.

8.2 Future Work

We now discuss some of the challenges that remain in the research area of real-time systems and are relevant to this dissertation.

8.2.1 General Results on Scheduling Real-Time Task Systems with Complex Runtime Behaviors

In Chapter 6, we investigated how to support sophisticated SRT task systems containing non-preemptive sections, self-suspensions, and graph-based precedence constraints. Since we have also presented in this dissertation solutions on supporting real-time parallel task systems and HRT self-suspending task systems, an important future work is to combine all these results together. Specifically, we would like to present a general multiprocessor scheduling design and analysis framework that can efficiently support both HRT and SRT task systems with all aforementioned complex runtime behaviors including non-preemptive sections, self-suspensions, graph-based precedence constraints, and parallel execution segments.

8.2.2 Improved Multiprocessor Schedulability Tests for HRT Sporadic Self-Suspending Task Systems

In Chapter 4, we presented global suspension-aware multiprocessor schedulability analysis techniques for HRT sporadic self-suspending task systems under both GEDF and GTFP scheduling. However, similar to the observation we discussed in the beginning of Section 3.4, these techniques do not address the worst-case scenario due to self-suspensions, which is the following: *all self-suspending tasks have jobs that suspend at non-busy time instants*. For the SRT case, in Section 3.4, we presented a technique that can transform a schedule on a processor-by-processor basis to eliminate this worst-case scenario, which enables us to derive an improved schedulability test that only results in an $O(m)$ suspension-related capacity loss. Motivated by this, we plan to extend the ideas used in our $O(m)$ SRT schedulability analysis to apply to HRT sporadic self-suspending task systems. In the HRT case, it may be similarly possible to transform the analyzed schedule on a processor-by-processor basis. However, transformation intervals may not be defined on the basis of tardy jobs, as done in Section 3.4.

8.2.3 Supporting Practical Suspending Task Models on Multiprocessors

As mentioned in Chapter 2, all prior work on scheduling self-suspending task systems assumes that each task's suspensions are simply upper-bounded and will not be interfered with by other tasks' suspensions. However, in practice, a task's suspensions are often interfered with by other tasks that access the same device. For example, a common scenario for suspensions to occur is when a task accesses an I/O device. There has been much work done on scheduling disk I/O requests in order to improve response times. This implies that in most (if not all) cases, a task's suspensions will be interfered with by other tasks' suspensions. Thus, it would be interesting to allow a task's suspension lengths to be affected by other tasks' suspensions (e.g., due to contention when multiple tasks simultaneously access the same shared resource). Solving this problem is equivalent to solving the problem of scheduling tasks that access two different classes of resources (e.g., CPUs and digital signal processors). Due to its difficulty, the general problem of real-time scheduling on two unrelated classes of resources has received limited attention in the real-time community.[1] This real-time scheduling problem is challenging because a task's execution on both classes of processors could be interfered with and delayed by other tasks. Due to the difficulty of this problem, we may first solve the uniprocessor version of the problem assuming there are only two classes of processors, each with one processor. Then the multiprocessor case can be solved by proposing and applying efficient partitioning algorithms to partition tasks on both classes of processors.

8.2.4 Supporting DAGs with Mixed Timing Constraints on Multiprocessors

For scheduling multiprocessor DAG-based task systems, if the deadline of every task within any DAG must be viewed as hard, then significant processing capacity must be sacrificed. On the other hand, if all deadlines are viewed as soft, then using multiprocessor global scheduling schemes such as GEDF can achieve no capacity loss; however, the resulting tardiness bound could be large. In practice, there exists a class of DAG-based applications that only require the exit task (i.e., the sink task) to meet its deadline. In other words, tasks other than the exit task within a DAG have soft deadlines. An excellent motivating application is the MapReduce application, which has emerged as an important paradigm in many large-scale data processing applications in modern data centers. MapReduce consists of two functions, map and reduce. The map phase partitions the entire input

[1]The end-to-end scheduling approach [86] can be applied to guarantee that any job's computation and suspensions phases complete within certain time bounds so that the whole job can meet its deadline. However, the end-to-end approach may result in severe utilization loss, and cannot be efficiently used in SRT systems. Note that for a relevant problem, namely the problem of real-time scheduling on a multiprocessor containing heterogeneous but related resources (i.e., a multiprocessor containing two CPUs with different speeds), several approaches have been proposed [4, 50, 102, 103, 104].

dataset (which could be very large in practice) into several smaller chunks that can be executed in parallel. Then the reduce phase aggregates all the partial results produced during the map phase and generates the final result. MapReduce applications can be naturally modelled as DAGs. It is often required that the final reduce phase (i.e., the exit task) must complete by a specified deadline. In other words, the map tasks are allowed to miss deadlines. Motivated by this, we plan to investigate the problem of supporting DAG-based task systems with mixed timing constraints on multiprocessors. For any DAG, it is only required to meet the exit task's deadline. Intuitively, allowing the deadlines of non-exit tasks to be missed offers some flexibility that could be utilized to improve overall system utilization. Thus, we plan to design and analyze real-time multiprocessor scheduling algorithms that can improve the utilization for scheduling such systems with mixed timing constraints, compared to the case where all deadlines must be viewed as hard.

8.2.5 Scheduling Heterogeneous Multiprocessor DAG-based Systems

In practice, many real-time DAG-based applications often contain tasks that may access several different classes of resources such as executing on a CPU and reading data from disk. In many CPSs, such applications become even more common since system components often access external devices such as sensors to obtain physical information. Motivated by this, we plan to investigate the problem of scheduling DAG-based systems on multiprocessors containing different classes of resources. We plan to focus on two classes of resources, computational resource such as CPUs and I/O resources such as disks. There are two goals for this research: (*i*) determine whether SRT DAG-based systems can be supported with no utilization loss, and (*ii*) design effective disk and CPU co-scheduling algorithms to improve the response time performance. (Note that this is different from Section 8.2.3 since tasks that access different classes of resources have inter-task dependencies such as DAG-based dependencies while the tasks considered in Section 8.2.3 that access different classes of resources have only intra-task dependencies, i.e., suspensions.)

8.2.6 Reducing the Magnitude of Tardiness Bounds

In this dissertation, we mainly focused on deriving tardiness bounds for real-time task systems with complex runtime behaviors. However, these bounds may not be tight. We plan to investigate the problem of reducing the magnitude of the tardiness bounds derived in this dissertation. One promising technique to apply is compliant vector analysis (CVA) [45]. CVA has been used to obtain tardiness bounds that are superior to previously known bounds for sporadic task systems [45, 46]. In contrast to the lag-based analysis framework used in this dissertation, which computes a single tardiness bound for all the tasks in the system, CVA seeks to derive a separate tardiness bound for each task. The resulting tardiness bounds for different tasks are more specific and thus tighter than a single general bound.

8.2.7 Innovations in Many-Core Real-Time Systems

A recent multicore-related development is the emergence of many-core architectures, where tens to thousands of processors are placed on the same chip. Many-core architectures are different from traditional multicore architectures because they rely on message passing mechanisms instead of shared memory for inter-core communication to ensure better scalability. This different inter-core communication method brings new challenges to support real-time graph-based applications on many-core platforms, because delays due to inter-core communications may depend upon the locality of processors where each part of the application graph is allocated. Thus, instead of scheduling only computations, we now have to consider data locality and judiciously schedule computations and communications at the same time. We plan to investigate this open problem by proposing locality-aware application mapping and processor selection strategies to minimize inter-core communications while providing timing correctness guarantees. Another challenging research topic imposed by many-core architectures is thermal management. It has been shown that power dissipation almost scales up linearly with the number of cores, which also directly increases operation temperature [85]. Without proper thermal management, thermal runaway and on-chip "hotspots" can be easily created on many-core chips; these reduce chip lifetime and may also significantly degrade chip performance. Therefore, we plan to propose effective thermal-aware task mapping and data routing schemes; the goal is to guarantee that the operating temperature of the chip does not exceed its critical temperature such that the reliability, stability, and certain performance objectives are achieved. One promising idea is to balance loads on both CPUs and routing links when making mapping decisions such that hotspots can be avoided. Moreover, selectively turning off certain cores or reducing their operating voltages could be effective methods to better manage heat on many-core chips.

8.2.8 Supporting Data-Intensive Workloads in Networked Real-Time Systems

With the rapidly growing popularity of Web 2.0 technologies and social business, an increasing number of applications are emerging that require large volumes of data and devote most of their processing time to analysis and manipulation of data. Many organizations have started to use data-intensive cluster computing systems (e.g., Hadoop) for applications such as Google's MapReduce. An important requirement of executing such data-intensive applications is to satisfy real-time constraints. For example, in time-sensitive processes such as detecting fraud, millions of daily call detail records must be analyzed in real time in order to predict customer churn faster. The data-intensive nature of such applications poses new challenges that we plan to investigate: (a) Is there a way to efficiently support applications with more frequent and longer suspensions than computation (due to intensive I/O access) in networked real-time systems? (b) What is the impact of data locality on guaranteeing fast and bounded response times? (c) What data parallel processing techniques are most efficient for supporting data-intensive applications in multicore systems with both high throughput and timing correctness? To answer these questions, factors that have not been considered by traditional solutions, such as network topology, data transmission overheads, and data locality, have to be explicitly considered when deriving new methods.

Furthermore, emerging data-intensive analytics and "Big Data" workloads pose new challenges to system design and configuration. Current systems are often designed and configured in response to more traditional workload demands. To execute such new workloads more efficiently, one promising improvement to traditional system design is the addition of reconfigurable acceleration. Accelerators such as FPGAs are more efficient for executing highly data-parallel functions such as matrix manipulations. While reconfigurable acceleration has been studied and used successfully in several commercial systems, the problem of meeting certain performance constraints such as power budgets and response time requirements, which commonly exist in many systems, has received limited attention. Motivated by this, we plan to address the research challenges brought by considering such constraints in the design and configuration of accelerator-based systems. For example, given a certain power budget, determine how to configure the system (i.e., which and how many power-efficient processing accelerators should be used) and how to offload performance-intensive functions to the reconfigurable logic in a way such that the system throughput can be maximized.

BIBLIOGRAPHY

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

[2] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

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

[4] B. Andersson, G. Raravi, and K. Bletsas. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 239-248, 2010.

[5] D. Bailey. An optimal scheduling algorithm for parallel video processing. In *Proceedings of the 5th IEEE International Conference on Multimedia Computing and Systems*, pages 245-248, 1998.

[6] T. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

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

[8] T. Baker and S. Baruah. *Handbook of Real-Time and Embedded Systems*, chapter Schedulability analysis of multiprocessor sporadic task systems. Chapman Hall/CRC, Boca Raton, Florida, 2007.

[9] T. Baker and S. Baruah. *Schedulability analysis of multiprocessor sporadic task systems, Handbook of real-time and embedded systems*. Chapman Hall/CRC, 2007.

[10] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119-18, 2007.

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

[12] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of the 26th Real-Time Systems Symposium*, pages 330–341, pages 321-329, 2005.

[13] S. Baruah, J. Haritsa, and N. Sharma. Online scheduling to maximize task completions. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 228-237, 1994.

[14] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182-190, 1990.

[15] A. Bastoni, B. Brandenburg, and J. Anderson. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 14-24, 2010.

[16] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. Mol, S. Stuijk, V. Gheorghita, and J. Meerbergen. Dataow analysis for real-time embedded multiprocessor system design. *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, 4(1):81–108, 2005.

[17] M. Bertogna and S. Baruah. Tests for global EDF schedulability analysis. *Journal of Systems Architecture*, 57(5):487–497, 2011.

[18] M. Bertogna and S. Baruah. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 149-160, 2007.

[19] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th Real-Time Systems Symposium*, pages 149-160, 2007.

[20] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–666, 2009.

[21] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209-218, 2005.

[22] S. Bhattacharyya, P. Murthy, and E. Lee. *Software Synthesis from Data on Graphs*. Kluwer Academic Publishers, 1996.

[23] A. Block. *Adaptive Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2008.

[24] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Trarjan. Time bounds for selection. *Journal of Computer Science and Systems*, 7(4):448–461, 1973.

[25] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2011.

[26] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, 1995.

[27] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247-256, 2007.

[28] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan Kaufmann, 2000.

[29] S. Chatterjee and J. Strosnider. Distributed pipeline scheduling: A framework for distributed, heterogeneous real-time system design. *The Computer Journal*, 38(4):271–285, 1995.

[30] S. Chatterjee and J. Strosnider. A generalized admissions control strategy for heterogeneous, distributed multimedia systems. In *Proceeding of ACM Multimedia*, pages 345-356, 1995.

[31] S. Chen and S. Schlosser. MapReduce meets wider varieties of applications. Technical Report IRP-TR-08-05, Intel Labs Pittsburgh, 2008.

[32] E. Coffman, M. Garey, and D. Johnson. *Approximation algorithms for bin packing: A survey*. PWS Publishing Company, 1997.

[33] Tilera Corporation. TILE-GX Processor with up to 100 cores. http://www.tilera.com/ products/processors/, 2010.

[34] DARPA. Satellite System F6 Specification and Requirement. http://www.darpa.mil/Our_-Work/TTO/Programs/Systemf6/ System_F6.aspx, 2011.

[35] R. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[36] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Conference on Symposium on Operating System Design and Implementation*, pages 137-150, 2004.

[37] M. Dertouzos. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[38] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Information Processing*, pages 807-813, 1974.

[39] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[40] U. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 23-30, 2003.

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

[42] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[43] E. Dougherty and P. Laplante. *Introduction to Real-Time Imaging*. Wiley-IEEE Press, 1995.

[44] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[45] J. Erickson, U. Devi, and S. Baruah. Improved tardiness bounds for global EDF. In *Proceedings of the 22nd Euromicro Conference in Real-Time Systems*, pages 14-23, 2010.

[46] J. Erickson, N. Guan, and S. Baruah. Tardiness bounds for global EDF with deadlines different from periods. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, pages 286-301, 2010.

[47] F. Abazovic. Intel showcases 80-core cpu. http://www.fudzilla.com/index.php?option=com_-content& task=view& id=10107& Itemid=1, 2008.

[48] D. Feitelson. Job scheduling in multiprogrammed parallel systems. Technical report, IBM Research, 1997.

[49] D. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 71–80, pages 1-18, 1995.

[50] S. Funk. *EDF Scheduling on Heterogeneous Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2004.

[51] S. Goddard. *On the Management of Latency in the synthesis of real-time signal processing systems from processing graphs*. PhD thesis, The University of North Carolina at Chapel Hill, 1998.

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

[53] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 387-397, 2009.

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

[55] W. Horn. Some simple scheduling algorithms. *IEEE Transactions on Computers*, 21(1):177–185, 1974.

[56] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of ACM*, 23(2):317–327, 1976.

[57] R. Jain, C. Hughes, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 134-145, 2002.

[58] P. Jayachandran and T. Abdelzaher. End-to-end delay analysis of distributed systems with cycles in the task graph. In *Proceedings of the 21th Euromicro Conference on Real-Time Systems*, pages 13-22, 2009.

[59] P. Jayachandran and T. Abdelzaher. Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 233–242, pages 233-242, 2008.

[60] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99, 1992.

[61] K. Jeffay and S. Goddard. A theory of rate-based execution. In *Proceedings of the 20th IEEE International Real-Time Systems Symposium*, pages 304-314, 1999.

[62] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[63] W. Kang, S. Son, J. Stankovic, and M. Amirijoo. I/O-Aware Deadline Miss Ratio Management in Real-Time Embedded Databases. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 277-287, 2007.

[64] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transaction on Parallel and Distributed Systems*, 8(12):1268–1274, 1997.

[65] I. Kim, K. Choi, S. Park, D. Kim, and M. Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, pages 54-59, 1995.

[66] Y. Kitamura, A. Smith, H. Takemura, and F. Kishino. Parallel algorithms for real-time colliding face detection. In *Proceedings of the 4th IEEE Workshop on Robot and Human Communication*, pages 211-218, 1995.

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

[68] J. Labetoulle. *Some theorems on real time scheduling*. Computer Architecture and Networks, pages 285-298, 1974.

[69] Naval Research Laboratory. Processing Graph Method Specification, prepared by the Naval Research Laboratory for use by the Navy Standard Signal Processing Program Office (PMS-412). 1987.

[70] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st Real-Time Systems Symposium*, pages 259-268, 2010.

[71] K. Lakshmanan and R. Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3-12, 2010.

[72] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of ACM*, 23(2):105–117, 1980.

[73] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[74] S. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *Proceedings of the 2007 ACM Conference on Management of Data*, pages 55-66, 2007.

[75] J. Lehoczky, L. Sha, J. Strosnider, and H. Tokuda. *Fixed priority scheduling theory for hard real-time systems*, chapter 1, pages 1–30. Foundations of Real-Time Computing: Scheduling and Resource Management, 1991.

[76] H. Leontyev. *Compositional Analysis Techniques For Multiprocessor Soft Real-Time Scheduling*. PhD thesis, University of North Carolina at Chapel Hill, 2010.

[77] H. Leontyev and J. Anderson. A unified hard/soft real-time schedulability test for global EDF multiprocessor scheduling. In *Proceedings of the 29th Real-Time Systems Symposium*, pages 375-384, 2008.

[78] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 413-422, 2007.

[79] J. Leung and M. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[80] C. Liu. Scheduling algorithms for hard-real-time multiprogramming of a single processor. In *JPL Space Programs Summary*, pages 31-37, 1969.

[81] C. Liu and J. Anderson. A new technique for analyzing soft real-time self-suspending task systems. In *ACM SIGBED Review*, pages 29-32, 2012.

[82] C. Liu and J. Anderson. An $O(m)$ analysis technique for supporting real-time self-suspending task systems. In *Proceedings of the 33th IEEE Real-Time Systems Symposium*, pages 373-382, 2012.

[83] C. Liu and J. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 425-436, 2009.

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

[85] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin. Power-efficient time-sensitive mapping in CPU/GPU heterogeneous systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 23-32, 2012.

[86] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

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

[88] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[89] O. Moreira and M. Bekooij. Self-timed scheduling analysis for real-time applications. In *EURASIP on Advances in Signal Processing*, pages 1-15, 2007.

[90] O. Moreira, J. Mol, M. Bekooij, and J. Meebergen. Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 332-341, 2005.

[91] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM/IEEE international conference on Embedded software*, pages 57-66, 2007.

[92] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 321-330, 2012.

[93] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic. An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *Proceedings of the 24th Euromicro Conference in Real-Time Systems*, pages 13-23, 2012.

[94] S. Pak. Azul Systems extends leadership in business critical Java applications performance with the new Vega series. http://www.azulsystems.com/ press/052008vega3.htm, May 2008.

[95] J. Palencia and M. Harbour. Offset-based response time analysis of distributed systems scheduled under EDF. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 3–12, 2003.

[96] J. C. Palencia and M. Gonzlez Harbour. Response time analysis of EDF distributed real-time systems. *Journal of Embedded Computing*, 1(2):225–237, 2005.

[97] J. C. Palencia and M. Gonzlez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 26-37, 1998.

[98] T. Parks and E. Lee. Non-preemptive real-time scheduling of dataflow systems. *International Conference on Acoustics, Speech, and Signal Processing*, 5(3235-3238), 1995.

[99] R. Pellizzoni and G. Lipari. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In *Proceedings of the 11th IEEE International Real Time and Embedded Technology and Applications Symposium*, pages 66–75, 2005.

[100] C. Philips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 110-149, 1997.

[101] R. Rajkumar. Dealing with Suspending Periodic Tasks. IBM T. J. Watson Research Center, 1991.

[102] G. Raravi, B. Andersson, and K. Bletsas. Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. *Real-Time Systems*, 49(1):29–72, 2013.

[103] G. Raravi, B. Andersson, K. Bletsas, and V. Neils. Task assignment algorithms for two-type heterogeneous multiprocessors. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 34-43, 2012.

[104] G. Raravi and V. Neils. A PTAS for assigning sporadic tasks on two-type heterogeneous multiprocessors. In *Proceedings of the 33th Real-Time Systems Symposium*, pages 117-126, 2012.

[105] F. Ridouard and P. Richard. Worst-case analysis of feasibility tests for self-suspending tasks. In *Proceedings of the 14th Real-Time and Network Systems*, pages 15-24, 2006.

[106] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 47-56, 2004.

[107] R. Ritz, M. Willems, and H. Meyer. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *Proceedings of the International Conference on Signal Processing*, pages 133-143, 1995.

[108] S. Ritz and H. Meyer. Exploring the design space of a DSP-based mobile satellite receiver. In *Proceedings of the International Conference on Signal Processing*, pages 24-34, 1994.

[109] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd Real-Time Systems Symposium*, pages 217-226, 2011.

[110] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[111] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*, 77(1):67–80, 2005.

[112] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, 2006.

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

[114] K. Tindell. Adding time-offsets to schedulability analysis. Technical Report 221, University of York, 1994.

[115] C. Volker, V. Hamscher, and R. Yahyapour. Economic scheduling in grid computing. In *Proceedings of the Conference on Scheduling Strategies for Parallel Processing*, pages 71–80, pages 128-152, 2002.

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

[117] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the European Conference on Computer Systems*, pages 265-278, 2010.

[118] Y. Zhang, C. Lu, C. Gill, P. Lardieri, and G. Thaker. Configurable middleware for distributed real-time systems with aperiodic and periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 21(3):393–404, 2010.

[119] V. Zivojnovic, R. Ritz, and H. Meyer. High performance DSP software using data-flow graph transformations. In *Proceedings of the 28th Asilomar Conference on Signals, Systems and Computers*, 1994.