Toward Efficient Scheduling for Parallel Real-Time Tasks on Multiprocessors

Son Ngoc Dinh
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Computer Sciences Commons

Recommended Citation
https://openscholarship.wustl.edu/eng_etds/540

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in Engineering and Applied Science Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.
Toward Efficient Scheduling for Parallel Real-Time Tasks on Multiprocessors

by

Son Ngoc Dinh

A dissertation presented to
The Graduate School
of Washington University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

May 2020
St. Louis, Missouri
Table of Contents

List of Figures ........................................................................................................... v
List of Tables ............................................................................................................. viii
Acknowledgments ..................................................................................................... ix
Abstract ..................................................................................................................... xii

Chapter 1: Introduction ............................................................................................. 1
  1.1 Problem Statement ............................................................................................. 3
  1.2 Contributions ...................................................................................................... 5
    1.2.1 Efficient Deterministic Scheduling Algorithm for Parallel Tasks ........ 5
    1.2.2 Analysis of Global Fixed-Priority for Parallel Tasks .......................... 6
    1.2.3 Analysis of Federated Scheduling with Shared Resources .............. 6
    1.2.4 Scalable Platform for Soft Real-Time Parallel Tasks .................... 7
  1.3 Organization ...................................................................................................... 8

Chapter 2: Background and Related Work ............................................................. 9
  2.1 Real-Time Scheduling for Sequential Tasks .................................................. 9
    2.1.1 Schedulability Metrics ........................................................................... 11
    2.1.2 Uniprocessor Scheduling ...................................................................... 12
    2.1.3 Multiprocessor Scheduling .................................................................. 13
  2.2 Real-Time Locking Protocols for Sequential Tasks ...................................... 17
    2.2.1 Priority Inversion Problem .................................................................. 17
    2.2.2 Uniprocessor Locking Protocols ......................................................... 19
    2.2.3 Multiprocessor Locking Protocols ....................................................... 20
  2.3 Real-Time Scheduling and Resource Sharing for Parallel Tasks .................. 22
    2.3.1 Parallel Task Model & Metrics ............................................................. 23
## Chapter 2: Real-Time Scheduling for Parallel Tasks

### 2.3.2 Real-Time Scheduling for Parallel Tasks

### 2.3.3 Resource Sharing in Real-Time Parallel Tasks

### 2.4 DAG Task Model

## Chapter 3: Efficient Deterministic Federated Scheduling for Parallel Tasks

### 3.1 Introduction

### 3.2 A Deterministic Scheduling Algorithm for DAG Tasks

### 3.3 A Federated Scheduling Algorithm

### 3.4 Evaluation for Heavy Tasks

### 3.5 Evaluation Versus the State-of-the-Art

#### 3.5.1 The State-of-the-Art Federated-Based Scheduling

#### 3.5.2 Experimental Evaluation

### 3.6 Summary

## Chapter 4: Analysis of Global Fixed-Priority for Parallel Tasks

### 4.1 Introduction

### 4.2 Background

#### 4.2.1 Assumptions and Notations

#### 4.2.2 Critical Chain and Critical Interference

#### 4.2.3 A General Method for Bounding Response-Time

### 4.3 The State-of-the-Art Analysis for G-FP

### 4.4 Bound for Carry-Out Workload

### 4.5 Response-Time Analysis

### 4.6 Experimental Evaluation

### 4.7 Summary

## Chapter 5: Analysis of Federated Scheduling for Parallel Tasks with Shared Resources

### 5.1 Introduction

### 5.2 Task and Resource Model

### 5.3 Schedulability Analysis

### 5.4 Calculating Intra-task Blocking

### 5.5 Calculating Inter-task Blocking

---

iii
List of Figures

Figure 2.1: Example for real-time sequential task............................................. 10
Figure 2.2: Each processor has a separate ready queue in partitioned scheduling. 14
Figure 2.3: Processors share a single ready queue in global scheduling. ............... 16
Figure 2.4: Example for priority inversion on uniprocessors.............................. 18
Figure 2.5: Synchronous parallel task model. ................................................. 23
Figure 2.6: Example of DAG task............................................................... 24
Figure 2.7: Federated scheduling demonstration.............................................. 27
Figure 2.8: Example of DAG task considered in this dissertation...................... 31

Figure 3.1: The deterministic schedule computed by Algorithm 1 for the DAG in Figure 2.8. ........................................................... 40
Figure 3.2: A schedule computed by the algorithm in [11] for the DAG in Fig. 2.8. 46
Figure 3.3: Ratio of processors required by PRO and BAR. ............................. 50
Figure 3.4: Ratio of schedulable task sets for varying total utilization and number of processors. ................................................................. 51
Figure 3.5: Ratio of schedulable task sets for varying total utilization $U$. .......... 55
Figure 3.6: Ratio of schedulable task sets for varying $\frac{U_{th}}{U}$. ....................... 56
Figure 3.7: Ratio of schedulable task sets for 0.9 normalized density, 0.45 normalized density for heavy tasks, and varying $\frac{D_i}{T_i}$. .................... 57

Figure 4.1: Critical chain and critical interference of $J_k$. ............................... 65
Figure 4.2: Critical chain and critical interference of $J_k$. ............................... 66
Figure 4.3: Workload generated by an interfering task $\tau_i$ in an interval of length $\Delta$. 68
Figure 4.4: Workload generated by an interfering task $\tau_i$ in Melani et al. [81]...... 73
| Figure 4.5: | Example for a general DAG task and a nested fork-join DAG task. | 75 |
| Figure 4.6: | An illustration of generating the maximum carry-out workload. | 77 |
| Figure 4.7: | Ratio of schedulable task sets for varying total utilization and varying number of processors. | 86 |
| Figure 5.1: | Example task set with FIFO-ordered spin locks. Each node is a unit node with a work of 1 time unit. | 110 |
| Figure 5.2: | Example schedules that cause worst case work blocking and critical-path blocking for $\tau_1$ with FIFO-ordered spin locks. | 111 |
| Figure 5.3: | Example schedules for the updated number of processors for $\tau_1$ with FIFO-ordered spin locks. | 111 |
| Figure 5.4: | Example task set with priority-ordered spin locks. Each node has work of 1 time unit. | 113 |
| Figure 5.5: | Example schedules with the current and updated number of processors for $\tau_1$ with priority-ordered spin locks. | 114 |
| Figure 5.6: | Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections. | 119 |
| Figure 5.7: | Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, 4 shared resources, varying number of critical sections per resource, and short critical sections. | 119 |
| Figure 5.8: | Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, varying number of resources, 128 requests per shared resource, and short critical sections. | 119 |
| Figure 5.9: | Schedulability for $m = 36$, $n = 11$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections. | 120 |
| Figure 5.10: | Schedulability for $m = 12$, $n = 5$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections. | 120 |
| Figure 5.11: | Schedulability for $m = 36$, $n = 9$, 1 shared resource with a priority-ordered lock, short requests, varying total utilization and number of requests. | 120 |
| Figure 5.12: | Ratio of successful task sets for $m = 36$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and moderate critical sections. | 126 |
Figure 6.1: Example of a synchronous program. ........................................... 131
Figure 6.2: Examples of centralized scheduling and work stealing ................. 133
Figure 6.3: Speedup of synchronous tasks in OpenMP and Cilk Plus implementa-
tions .................................................................................... 139
Figure 6.4: Speedup of benchmark programs in OpenMP and Cilk Plus implement-
tions .................................................................................. 142
Figure 6.5: Cholesky with input size $1000 \times 1000$ base case $4 \times 4$............... 143
Figure 6.6: Deadline miss ratio of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS (pro-
viding federated scheduling service integrated with a randomized work-stealing scheduler in GNU Cilk Plus) and RTCG (providing fed-
erated scheduling service integrated with a centralized greedy scheduler in GNU OpenMP). RTWS and RTCG use the same core assignment. 152
Figure 6.7: Average relative response time of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under
RTWS (providing federated scheduling service integrated with a ran-
domized work-stealing scheduler in GNU Cilk Plus) and RTCG (pro-
viding federated scheduling service integrated with a centralized greedy scheduler in GNU OpenMP). RTWS and RTCG use the same core
assignment. .......................................................................... 153
Figure 6.8: Required number of cores of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS (pro-
viding federated scheduling service integrated with a randomized work-
stealing scheduler in GNU Cilk Plus) and RTCG (providing federated scheduling service integrated with a centralized greedy scheduler in
GNU OpenMP). We increase the number of cores for each task under
RTCG until it misses no more than 60% of deadlines. ....................... 154
# List of Tables

Table 3.1: Comparison to BAR and LI for Heavy DAG Tasks .......................... 49

Table 5.1: Notation Summary ........................................................................ 96

Table 5.2: Overhead of spin locks ............................................................... 124

Table 6.1: Median, maximum, and 99\textsuperscript{th} percentile execution times of synchronous tasks for OpenMP and Cilk Plus implementations (in milliseconds) and the ratios of the maximum execution times of Cilk Plus over OpenMP implementations. ....................................................... 138

Table 6.2: Median, maximum, and 99\textsuperscript{th} percentile execution times of Cholesky, LU, and Heat for OpenMP and Cilk Plus implementations (in seconds) and the ratio of the maximum execution times of Cilk Plus over OpenMP implementations. ....................................................... 141
Acknowledgments

First and foremost, I would like to thank my academic advisors, Dr. Chris Gill and Dr. Kunal Agrawal, who have patiently and understandingly guided me through my graduate study. With their expertise and patience, they have taught me elements of conducting research, including studying a problem at detailed, technical level as well as from a broader view, looking at a problem from different angles, and research communication. This dissertation would not have been possible without their guidance and help. I am grateful to have them as my academic advisors.

I would like to extend my sincere thanks my collaborators Dr. Jing Li and Dr. Chenyang Lu, with whom I have had chances to write papers. Dr. Jing Li has always amazed me with her technical ability and relentless drive to conduct high quality research. Dr. Chenyang Lu has given me immensely helpful advices and insights for my research during our project discussions.

I am thankful to my fellow students David Ferry, James Orr, and Chao Wang for making my graduate study a joyful journey. David has calmly helped me debug my code and fix my experimental machines while I was anxious trying to get a paper submitted. With James and Chao, I have had plenty of lunches together and we have spent fun time sharing our daily life stories as well as research ideas.
I would also like to thank the Department of Computer Science and Engineering for making this place our second home. I greatly appreciate the department staffs, who are always willing to help and support during my study.

Thanks to the National Science Foundation which has funded my graduate program under grants CCF-1136073, CCF-1137218, and CNS-1814739.

My deepest gratitude goes to my family for their constant support throughout my graduate study. My mother, even though living on the opposite side of planet Earth, has stepped alongside with me and helped me regain balance during hardships in life. My wife has sacrificed many things back home and accompanied me the whole time, through thick and thin. You two are the strongest and most warmhearted women in the world.

Son Ngoc Dinh

Washington University in Saint Louis

May 2020
Dedicated to my wife, Hà Thu Phương (Miu)
ABSTRACT OF THE DISSERTATION

Toward Efficient Scheduling for Parallel Real-Time Tasks on Multiprocessors

by

Son Ngoc Dinh

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2020

Professor Christopher D. Gill, Chair

Associate Professor Kunal Agrawal, Co-Chair

Modern real-time applications are becoming more demanding computationally while their temporal requirements, dictated by the physical world, often remain unchanged. This coupled with the increasing prevalence of multiprocessors in real-time systems necessitates that highly computation-demanding real-time tasks need to be parallelized to exploit the parallelism offered by the underlying hardware, in order to satisfy their temporal constraints. Scheduling parallel real-time tasks, however, introduces a new layer of complexity due to the allowance for intra-task parallelism. This dissertation addresses the problem of scheduling parallel real-time tasks in which tasks may (or may not) access shared non-processor resources, such as in-memory buffers or data structures. Specifically, for independent tasks, we propose new scheduling algorithms and schedulability analyses for parallel tasks with these characteristics, under federated and global scheduling. Experimental results show that the proposed algorithms and analyses improve the previously introduced methods. For parallel tasks that may access shared non-processor resources, we present a blocking analysis for two different types of spinlocks; through evaluations, we make a recommendation for a preferable ordering of locks. We also study practical runtime parallel scheduler designs for soft real-time applications and present a design that is more suitable for soft real-time systems.
Chapter 1

Introduction

Real-time computing systems have a crucial role in many areas of the modern world, from non-critical systems to safety- and mission-critical systems. Examples of non-critical systems that have real-time constraints include multimedia streaming, online gaming, virtual reality, human tracking [99], and trading and bidding systems [52, 100], as response data should be computed and delivered to end users in a predictable manner. In safety- and mission-critical systems, such as medical emergency systems [86], automotive applications [59], air traffic control systems [24], power nuclear plants [2], and spacecraft [87], real-time computing plays an even more crucial role since violating timing constraints in such systems could result in catastrophic consequences. Such safety- and mission-critical applications in which no deadline miss is permitted are called hard real-time, whereas those applications in which deadline misses are tolerable at the cost of performance degradation, such as in online gaming, are called soft real-time.

Traditionally, real-time applications were deployed on uniprocessors, i.e., multiple real-time applications would share a single processing unit. There has been an extensive body of work
invested in this setup, and the real-time systems community has attained many significant results and gained a thorough understanding of the problem. The advent of multi-core processors\(^1\) brings new opportunities and challenges to designing and developing real-time systems. On one hand, the increased processing capacity provided by multiprocessors enables a broader range of real-time applications to be deployed. On the other hand, it introduces new challenges to efficiently and correctly exploiting the enhanced underlying hardware.

Recently, a new class of real-time applications in which a task can execute simultaneously on multiple processors, has attracted much effort from the real-time systems community. This class represents real-time tasks with high computational demands that must be parallelized to execute on more than one processor at the same time to satisfy their timing constraints, e.g., deadlines. Examples of such real-time parallel tasks are computer vision in autonomous vehicles [71] and real-time hybrid structural simulation [48, 49]. Inherently, it is more difficult to schedule (and analyze schedulability for) real-time parallel tasks since each execution of a parallel task can be interfered with not only by other tasks (inter-task interference) but also by other subtasks of the same task (intra-task interference). This is different than with conventional sequential tasks, where only inter-task interference exists.

In this dissertation, we study the problem of scheduling real-time parallel tasks on multi-core processors in which tasks may be independent — that is they do not share any resources other than processors — or tasks may share access to non-processor resources such as in-memory data structures, network buffers. In the remainder of this chapter, we specifically state the problem studied by this dissertation and briefly outline its contributions.

---

\(^1\)In this dissertation, we use the terms multiprocessors and multi-core processors interchangeably. Similarly, the terms processors and cores are used interchangeably.
1.1 Problem Statement

Parallel task models were introduced in work by Kato et al. [70] and Lakshmanan et al. [72]. Parallel task models allow each individual task to execute simultaneously on multiple cores at the same time. Due to this, they are suitable for representing real-time tasks with high computational demands whose deadlines would be missed if they are executed only on a single core. In practice, parallel tasks can be implemented using prominent parallel programming languages such as Cilk Plus [40], OpenMP [23], Intel Threading Building Blocks [41], or the more primitive POSIX threads library. Since their introduction, real-time parallel tasks have attracted numerous efforts to generalize the model, and to design new scheduling algorithms, as well as to improve their analyses.

Unlike multiprocessor scheduling of sequential tasks for which efficient scheduling algorithms and analyses exist, scheduling parallel tasks is more challenging since they introduce intra-task interference in addition to inter-task interference. The mixing of these two types of interference tends to make analyzing schedulability of parallel tasks under familiar algorithms such as Global Earliest Deadline First (G-EDF) or Global Fixed-Priority (G-FP) pessimistic. Despite recent progress, there is still a gap between the analytical performance of real-time parallel tasks and that of sequential tasks on multiprocessors. This dissertation attempts to narrow this gap and proposes new algorithms and analyses for scheduling parallel tasks.

Thus far, we have been assuming that parallel tasks are independent, i.e., tasks do not share any resources other than processors. In reality, tasks may also access other shared resources such as memory or I/O devices. Typically, such shared resources are protected by locks — task must acquire the lock for a resource before it can access the shared resource, and release the lock once it finishes accessing the resource. If a task attempts to access a shared resource while it has been locked, then the requesting task is blocked and has to wait until
the requested lock becomes available. If the requesting task has higher priority than the task that is holding the lock, then priority inversion happens since the higher-priority task has to wait for the lower-priority task. Without proper management, priority inversions can be arbitrarily long, severely harming the schedulability of the system. Consequently, locking protocols and their associated analyses have been proposed to bound the duration of priority inversions [7, 54, 84, 85, 92]. Priority inversion bounds are then accounted for when one determines the schedulability for a given task set.

For real-time sequential tasks, locking protocols and associated analyses have been studied extensively for both uniprocessors and multiprocessors over a couple of decades. Locking protocols have been proposed that efficiently bound priority inversions. Examples of such locking protocols are the Priority Inheritance Protocol (PIP), Priority Ceiling Protocol (PCP) [92], and Stack Resource Policy (SRP) [7] for uniprocessors, and the Multiprocessor PCP (MPCP) [84], and Multiprocessor SRP (MSRP) [54] for multiprocessors. For parallel tasks, however, there has been little work that considers shared resources between tasks. Since intra-task parallelism is allowed for parallel tasks, a request to a shared resource issued by a given core can be blocked by not only requests from other tasks but also requests from other cores of the same task. This makes blocking analysis for parallel tasks challenging.

This dissertation addresses the two problems discussed above regarding scheduling real-time parallel tasks on multiprocessors:

1. How to efficiently schedule parallel tasks on multiprocessors to satisfy their timing constraints.

2. How blocking incurred by parallel tasks due to access to shared resources, e.g., in-memory buffers or data structures, can be analyzed and accounted for properly.
1.2 Contributions

In the following, we outline the contributions of this dissertation, which are presented in detail in the subsequent chapters.

1.2.1 Efficient Deterministic Scheduling Algorithm for Parallel Tasks

Federated scheduling was proposed by Li et al. [76] as a generalization of partitioned scheduling for parallel tasks. In this approach, parallel tasks are classified into two types: (i) tasks that require more than one processor to meet their deadlines (*heavy tasks*), and (ii) tasks that can still meet their deadlines with sequential execution on a single processor (*light tasks*). Heavy tasks and light tasks are then treated differently. In particular, each heavy task is allocated a set of dedicated processors for its execution, i.e., it does not share its processors with any other tasks; all light tasks are scheduled together on the remaining processors. Federated scheduling turns out to be a promising approach to scheduling parallel tasks due to its isolation of tasks that must execute in parallel, from tasks that can behave like sequential tasks. Consequently, heavy tasks do not interfere with each other and we can focus on scheduling heavy tasks individually; light tasks can be scheduled using any of the existing scheduling approaches for sequential tasks on multiprocessors.

Federated scheduling, however, may waste resources if the scheduling of heavy tasks is not designed carefully. The reason is that each heavy task is allocated dedicated processors, and thus the processor cycles that are not utilized by the given heavy task also cannot be utilized by other tasks. This problem gets worse if heavy tasks are over-provisioned, i.e., when they are allocated more dedicated processors than they actually require. We propose a new algorithm for scheduling heavy tasks that efficiently exploits the processors exclusively assigned to them.
As a result, it may reduce the number of dedicated processors allocated to each heavy task. We experimentally compare our algorithm with the state-of-the-art federated-based scheduling algorithms and show that our algorithm outperforms the state-of-the-art algorithms in those evaluations.

1.2.2 Analysis of Global Fixed-Priority for Parallel Tasks

We present an analysis of global fixed-priority (G-FP) scheduling for parallel tasks [43]. In G-FP, each task is assigned a fixed priority, i.e., all jobs of each task have the same priority. Tasks are then scheduled globally on a multiprocessor platform — subtasks of each task are allowed to migrate between processors during their execution. Fixed-priority scheduling, in general, has an advantage of being easier to implement compared to dynamic scheduling policies such as earliest deadline first.

Our analysis first computes the worst-case interference that a job of a task may experience due to the workload generated by jobs of higher-priority tasks. The worst-case interference is then incorporated into a response-time analysis for each task to compute an upper-bound for the task’s response time. Experimental results show that our analysis performs better than previous analyses in terms of the ratio of schedulable task sets.

1.2.3 Analysis of Federated Scheduling with Shared Resources

We also consider parallel tasks scheduled under federated scheduling in which tasks can access shared resources, such as I/O devices, memory buffers. The shared resources are reusable and mutual exclusive. Each resource is protected by a spin lock, i.e., a requesting processor spin-waits until the requested lock is available and releases the lock once it finishes accessing the resource. We propose an analysis for the blocking incurred by the tasks due to contention
for shared resources and incorporate the analyzed blocking into a *schedulability test* for the task set [44].

We consider two types of spin locks: *First-In, First-Out-ordered* (FIFO-ordered) spin locks and *priority-ordered* spin locks. For FIFO-ordered locks, requests to the same lock are served in first-come, first-served order. For priority-ordered locks, each request has an associated priority and requests to the same lock are served based on their priorities — that is, requests with the highest priority are satisfied first. We conduct numerical and empirical experiments to compare the performances of these two types of spin locks. The results show that priority-ordered spin locks give better schedulability in general, at a cost of higher analytical complexity.

### 1.2.4 Scalable Platform for Soft Real-Time Parallel Tasks

Due to the isolation of heavy and light tasks, federated scheduling can be implemented conveniently based on well-known parallel programming languages such as OpenMP [23] and Cilk Plus [40]. In particular, each heavy task written in a given parallel programming language runs independently on its dedicated processors with a separate instance of the language’s associated runtime systems. Each heavy task can employ an off-the-shelf implementation of the chosen parallel programming language with minimal modification. Since light tasks do not need to execute in parallel, they are scheduled together as sequential tasks on the remaining processors. An existing scheduling algorithm for multiprocessors can be used to schedule the light tasks.

An interesting question arises regarding choosing appropriate parallel languages and runtime systems for implementing heavy tasks. There are two different approaches for implementing the scheduler of a parallel language — the scheduler that schedules the spawned work from
concurrent threads of a parallel program — namely centralized scheduling and randomized work stealing [22]. OpenMP is a widely known language that implements centralized scheduling, whereas Cilk Plus is a notable parallel language that implements randomized work stealing. These two approaches are fundamentally different in the way they handle work spawned during the execution of a parallel program. We conduct an empirical study to compare these two strategies when combined with federated scheduling in terms of scalability and timing predictability, and make our suggestion for which one is more suitable for real-time systems [77].

1.3 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we give a background discussion and summarize related work on real-time scheduling and resource sharing for sequential and parallel tasks. In Chapter 3, we present our new scheduling algorithm for parallel tasks and a federated scheduling algorithm that improves on previously proposed federated-based scheduling algorithms. Chapter 4 presents an analysis for global fixed-priority scheduling of generalized parallel tasks. Chapter 5 discusses a blocking analysis for parallel tasks that can access share resources and are scheduled with federated scheduling. In Chapter 6, we present a case study for applying federated scheduling together with randomized work stealing to support large-scale soft real-time parallel applications. Finally, Chapter 7 concludes this dissertation and discusses future directions and open problems.
Chapter 2

Background and Related Work

In this chapter, we discuss background, introduce notation, and summarize major results attained in the field of real-time scheduling and synchronization. This gives us an appropriate context for the contributions presented in the subsequent chapters. We start with the classical problem of scheduling sequential tasks on uniprocessors.

2.1 Real-Time Scheduling for Sequential Tasks

Representation of Real-Time Tasks: Real-time tasks are often recurrent processes in which each process may release an infinite number of instances or jobs. Any two consecutive jobs of a task $\tau_i$ are released at least $T_i$ time units apart. The value $T_i$ is called the period or minimum inter-arrival time of task $\tau_i$. Execution times of jobs of $\tau_i$ are bounded above by $\tau_i$’s worst-case execution time (WCET) $C_i$. The utilization $u_i$ of $\tau_i$ is defined as the computational time required by $\tau_i$ per one time unit in each release: $u_i = \frac{C_i}{T_i}$. Similarly, the density $\delta_i$ of $\tau_i$ is defined as $\delta_i = \frac{C_i}{\min(T_i, D_i)}$. Each job of $\tau_i$ must finish within $D_i$ time units after its arrival. The value $D_i$ is thus called the relative deadline of $\tau_i$. The absolute deadline $d_{i,j}$ of a job $J_{i,j}$
of $\tau_i$ is computed by $d_{i,j} = a_{i,j} + D_i$, where $a_{i,j}$ is the arrival time (also its release time if the job is released immediately after its arrival) of $J_{i,j}$. Figure 2.1 illustrates an example for real-time sequential task. In practice, a real-time task can be implemented as a program which is triggered for execution by an external event source, such as a timer, at a rate of at most every $T_i$ time units. A more detailed discussion for an interpretation of real-time tasks in practice is given by Brandenburg [27].

![Figure 2.1: Example for real-time sequential task.](image)

**Schedulability Analysis:** Real-time scheduling studies the problem of scheduling a set of real-time tasks on a shared platform, such as a uniprocessor or a homogeneous multiprocessor machine. The goal is to schedule the tasks so that all jobs of each task meet their deadlines (in the case of hard real-time systems) on the common platform. An important notion in real-time scheduling is a **schedulability analysis** or **schedulability test** for a scheduling algorithm. Specifically, a schedulability test of a scheduling algorithm $\mathcal{A}$ for a given platform is a procedure that receives a task set as its input and determines whether the task set is schedulable by algorithm $\mathcal{A}$ on that platform or not. Before a task set can be deployed on a given platform using an algorithm $\mathcal{A}$, it must pass a schedulability test for $\mathcal{A}$. Consequently, improving the efficacy of schedulability analyses for scheduling algorithms is an important objective in real-time scheduling, in addition to developing new scheduling algorithms.
A schedulability test of algorithm $\mathcal{A}$ is called a **sufficient test** if it may incorrectly deem a task set unschedulable under $\mathcal{A}$ — that is, a task set deemed unschedulable by a sufficient test for algorithm $\mathcal{A}$ may still actually be schedulable under $\mathcal{A}$. However, all task sets that are deemed schedulable by a sufficient test are actually schedulable. A schedulability test of $\mathcal{A}$ is called an **exact test** if, for every task set, it can determine exactly whether the task set is schedulable under $\mathcal{A}$ or not.

### 2.1.1 Schedulability Metrics

To compare the performance of different scheduling algorithms, different metrics have been proposed. The purpose of those metrics is to give us a theoretical and intuitive evaluation of a scheduling algorithm’s performance in terms of its ability to schedule real-time tasks. Performance metrics can sometimes be used as a quick schedulability test, such as a utilization bound discussed below. In most cases, however, separate schedulability analyses are developed that can analyze scheduling algorithms more efficiently. Several frequently used metrics are discussed in the following.

**Utilization Bound:** A **utilization bound** is among the first metrics used for measuring schedulability. The utilization bound $U_{ub}$ for a scheduling algorithm $\mathcal{A}$ is an upper-bound of the total utilization of the input task set such that any task set with total utilization less than or equal to $U_{ub}$ is schedulable by $\mathcal{A}$. Utilization bounds can be used as a sufficient test for a given algorithm.

**Resource Augmentation Bound:** A **resource augmentation bound** or **speedup bound** is a performance metric introduced for multiprocessor scheduling of real-time tasks. The resource augmentation bound of a scheduling algorithm $\mathcal{A}$ gives a theoretical comparison of $\mathcal{A}$ with an optimal algorithm. Specifically, the resource augmentation bound of an algorithm
\(A\) is the minimum factor by which the speed of each processor of the considered multiprocessor platform must be increased in order for \(A\) to successfully schedule all task sets that are schedulable by an optimal algorithm on the original multiprocessor platform, i.e., with all processors having speed of 1. From its definition, algorithms with smaller speedup bounds are more favorable theoretically.

### 2.1.2 Uniprocessor Scheduling

The real-time scheduling problem for uniprocessors was first studied rigorously by Liu and Layland in their seminal paper [78]. In this work, they consider a preemptive, periodic task model in which each task \(\tau_i\) releases its jobs exactly \(T_i\) time units part, and has its relative deadline equal to its period, i.e., \(D_i = T_i\). Such a task \(\tau_i\) is called an implicit-deadline task. The constraint on the relative deadline can be relaxed to give more general tasks. In particular, a real-time task is called to have constrained deadline if \(D_i \leq T_i\), and arbitrary deadline if there is no constraint on \(D_i\).

Liu and Layland proved that the Earliest Deadline First (EDF) scheduling algorithm is optimal for the periodic task model — if a task set is schedulable under some algorithm, then it is also schedulable under EDF. In particular, they showed that a periodic task set is schedulable under EDF if the sum of utilizations of all tasks is less than or equal to 1.0, i.e., EDF has utilization bound of 1.0. EDF is a well-known representative for the dynamic priority scheduling class in which priorities of different jobs of the same task can be different. Another prominent class of scheduling algorithms is fixed-priority, such as Rate Monotonic (RM) or Deadline Monotonic (DM). In fixed-priority scheduling, the priority of a given task is fixed, i.e., all jobs of the task have the same priority. Liu and Layland also derived a utilization bound for RM which is computed by \(U_{ub} = n(2^{1/n} - 1)\), where \(n\) is the number
of tasks. RM is also proved to be an optimal fixed-priority algorithm for implicit-deadline, periodic task sets.

Baruah et al. [13] later studied the sporadic task model, which is a generalization of the periodic task model, with arbitrary deadlines. In the sporadic task model, each task may not necessarily release its jobs exactly $T_i$ time units apart; instead, $T_i$ only defines the minimum inter-arrival between the jobs. They proposed an exact test for EDF which runs in pseudo-polynomial time with high percentage. Zhang et al. [101] proposed an exact test for sporadic tasks scheduled under EDF that reduces the computation times compared to the previously introduced tests, thus improving the applicability of EDF in real systems.

Another notable technique for schedulability analysis is response-time analysis, which was first introduced by Audsley et al. [4]. The response time of a job is defined as the length of the interval from the job’s release to its completion. In this work, they presented a technique to bound the response times for tasks scheduled under fixed-priority scheduling algorithms such as Deadline Monotonic. Once the response-time bounds are computed, the schedulability of a task set can be determined by comparing the tasks’ response times with their deadlines. Response-time analysis turned out to be a powerful technique which can be employed for multiprocessor scheduling of sequential as well as parallel tasks. More comprehensive expositions for uniprocessor real-time scheduling are covered by Buttazzo [33] and Liu [79].

2.1.3 Multiprocessor Scheduling

There are two notable approaches regarding multiprocessor scheduling of real-time sequential tasks: global scheduling and partitioned scheduling. Partitioned scheduling is a natural extension of uniprocessor scheduling in which each task is assigned to a processor at design
time and is not allowed to migrate to other processors during the system’s run time. On each processor, an existing uniprocessor scheduling algorithm, such as EDF or DM, can be applied to scheduled the tasks assigned to it. Since efficient scheduling algorithms and analyses exist for uniprocessors, tasks can be scheduled efficiently on individual processors. Each processor in partitioned scheduling has a distinct ready queue, as illustrated in Figure 2.2.

The efficacy of partitioned scheduling, however, depends on the partitioning step. The problem of partitioning tasks to processors is basically the same as the bin packing problem, which is a NP-hard problem [55]. Therefore, there is no polynomial-time algorithm to solve it optimally unless P=NP; however, there are heuristics that can find good approximations for optimal solutions of the partitioning problem, such as first fit, next fit, best fit, and worst fit packing. These heuristics are typically applied with the tasks sorted in decreasing order of their utilizations (or densities).

Regarding schedulability analyses for partitioned scheduling, Andersson et al. [3] presented a fixed-priority scheduling algorithm for implicit-deadline tasks that has utilization bound
\( U_{ub} = m/2 \), where \( m \) is the number of processors of the system. Baruah et al. [16] showed that any sporadic, arbitrary-deadline task set satisfying the following condition is schedulable under partitioned EDF (P-EDF) with first fit decreasing density ordering:

\[
\delta_{sum} \begin{cases} 
  m - (m - 1)\delta_{max}, & \delta_{max} \leq 1/2, \\
  m/2 + \delta_{max}, & \delta_{max} \geq 1/2,
\end{cases}
\]

where \( \delta_{sum} \) is the total density of the task set, and \( \delta_{max} \) is the maximum individual density of the tasks. Fisher et al. [50] proposed a new partitioning algorithm with the DM scheduling policy and proved the associated schedulability tests for constrained- and arbitrary-deadline task sets.

In contrast to partitioned scheduling, global scheduling allows jobs to migrate between processors during their executions. For example, a job may start executing on a processor, then it is preempted, and it is later resumed on another processor. In practice, global scheduling are implemented with a single ready queue for all processors in the system, as shown in Figure 2.3. Compared to partitioned scheduling, global scheduling incurs higher overhead due to extra preemptions and migrations; it, however, potentially can exploit processors better due to its scheduling versatility. Because of its potential, there has been a substantial amount of work devoted to improving schedulability analysis of global scheduling. A comprehensive overview of multiprocessor scheduling, including both partitioned and global scheduling, is given by Davis and Burns [42]. In the following, we give a short summary of notable work on global scheduling of sequential tasks.

Baker [6] proposed new schedulability tests for constrained-deadline task sets scheduled under G-EDF and G-DM. Their tests rely on analyzing the execution window of each task \( \tau_i \) and accounting for the worst-case interference that \( \tau_i \) may incur. Baruah [10] later presented
Figure 2.3: Processors share a single ready queue in global scheduling.

A technique for analyzing G-EDF which improves on the previous work by Baker. In the same year, Bertogna et al. [17] introduced a response-time analysis for G-EDF and G-FP by identifying and accounting for the worst-case interfering pattern that a task may experience. They showed that their technique experimentally outperforms previous analyses. Guan et al. [58] later proposed a sophisticated response-time analysis for G-FP which improves on previous techniques for G-FP. Notably, Brandenburg et al. [32] recently showed that for static workloads — workloads that do not change requirements and constraints dynamically (such as adding or removing tasks at runtime) — global scheduling is not required to schedule these workloads effectively. According to their simulations and empirical studies on the LITMUS\textsuperscript{RT} testbed [27, 34], near 100% computational utilization can be achieved with semi-partitioned scheduling — a hybrid approach between partitioned and global scheduling which allows some, but not all, tasks to migrate in a pre-defined manner while the other tasks are partitioned to processors — combined with carefully crafted techniques, such as reservations and period transformation.
2.2 Real-Time Locking Protocols for Sequential Tasks

All the work reviewed in the previous section assumes that tasks are independent, i.e., they do not share resources other than processors. In practice, however, tasks may share resources such as in-memory data structures, or network buffers. Those shared resources introduce additional constraints between tasks and give rise to the problem of priority inversions. If not addressed properly, priority inversions can severely harm the temporal correctness of the system. In fact, priority inversions caused the Mars Pathfinder spacecraft [69] to reset multiple times during its mission, which resulted in losses and delays of data being transmitted to the Earth. In this dissertation, we are interested in shared resources that are protected by locks — a processor must acquire the lock before it can access a resource, and release the lock as soon as it finishes accessing the resource. The segment of code in between the acquire and release operations is called critical section. In the following we briefly summarize important results previously attained regarding locking protocols and their analyses for sequential tasks.

2.2.1 Priority Inversion Problem

Figure 2.4 shows an example for priority inversion on a single processor with three jobs. The priority order of the jobs is as follows: $J_1$ has the highest priority, followed by $J_2$; and $J_3$ has the lowest priority. Jobs $J_1$ and $J_3$ require access to a shared resource protected by a mutually exclusive lock, i.e., only one job can acquire the lock at a time, whereas $J_2$ does not require the resource. At time $t_1$, while $J_3$ is executing its critical section, $J_1$ preempts it and executes until time $t_2$. Job $J_1$ then attempts to acquire the lock to access the shared resource; however, the lock is currently held by $J_3$, thus causing $J_1$ to be blocked. $J_3$ then resumes its critical section until time $t_3$ when $J_2$ arrives and preempts it. After $J_2$ completes
at time $t_4$, $J_3$ again resumes its critical section until time $t_5$ when it releases the lock. Only at this time can $J_1$ acquire the lock and start its critical section. During the interval $[t_2, t_5)$, $J_1$ cannot execute while the lower-priority jobs $J_3$ and $J_2$ are scheduled; this violates the expected behavior of the system.

The priority inversion experienced by $J_1$ can be arbitrarily long, if, for example, $J_2$ has a large execution time or the critical section of $J_3$ is preempted by multiple intermediate-priority jobs. We note that in the interval $[t_2, t_5)$, the priority inversions in the sub-intervals $[t_2, t_3)$ and $[t_4, t_5)$ are unavoidable; these sub-intervals account for one critical section of $J_3$, which is often short in practice. The priority inversion caused by intermediate-priority jobs is, however, long since it can be formed by the whole jobs’ executions, which thus needs to be avoided. All synchronization protocols proposed, hence, aim to bound priority inversion as a function of the duration of critical sections, and not as a function of the duration of non-critical sections.

![Figure 2.4: Example for priority inversion on uniprocessors.](image-url)
2.2.2 Uniprocessor Locking Protocols

The *Priority Inheritance Protocol* (PIP), introduced by Sha et al. [92], is among the first synchronization protocols that was designed for fixed-priority scheduling on uniprocessors. They consider that each resource is protected by a binary semaphore, i.e., a requesting job will suspend if the requested lock is currently held by another job. PIP works by letting the job that holds the semaphore inherit the highest priority of the jobs blocked by it. For the jobs in Figure 2.4, at time $t_2$, $J_3$ would inherit the priority of $J_1$ since it blocks $J_1$. Therefore, when $J_2$ arrives at time $t_3$, it would not be able to preempt $J_3$, thus avoiding unnecessary priority inversion for $J_1$ caused by $J_2$. When $J_3$ releases the locks, its priority will return back to its original value. PIP, however, has two problems: (i) it does not prevent deadlocks, and (ii) the blocking duration of a job, even though bounded, can still be large since blocking can be chained.

In the same work, Sha et al. [92] proposed a more advanced protocol, named the *Priority Ceiling Protocol* (PCP), that addresses the two problems of PIP. In PCP, each semaphore is assigned a priority ceiling that is equal to the highest priority of the tasks that require this semaphore. A job can start a new critical section only if its priority is higher than all priority ceilings of all semaphores that are already locked by other jobs. They also derived a schedulability analysis for the rate-monotonic scheduling algorithm when PCP is used.

Baker [7] proposed a powerful synchronization protocol, named the *Stack Resource Policy* (SRP). Compared to PCP, SRP can work with multiunit resources, whereas PCP can only be used with mutually exclusive resources. Moreover, SRP can be applied with dynamic-priority scheduling policies such as EDF, and allows jobs to share a runtime stack (hence its name). While having these advantages over PCP, SRP maintains similar blocking properties as PCP for binary resources — it prevents deadlocks and strictly bounds priority inversion. PCP and
SRP are the two most notable protocols designed for uniprocessors, and they have greatly influenced the later work on synchronization protocols for multiprocessors.

### 2.2.3 Multiprocessor Locking Protocols

Multiprocessors with critical sections introduce *remote blocking*, which happens when a job $J_a$ running on a processor requests a lock that is currently held by another job $J_b$ running on a different processor. Consequently, if $J_b$ is preempted by a higher priority job $J_c$ while it is executing its critical section, $J_a$ will be blocked by the whole execution duration of $J_c$. Despite being solid protocols for uniprocessors, SRP and PCP do not scale across processors and cannot bound remote blocking efficiently.

To address this problem, several extensions of PCP and SRP for multiprocessors have been proposed. Rajkumar et al. [85] presented an extension of PCP for tasks scheduled with the *Partitioned Rate Monotonic* (P-RM) scheduling policy on multiprocessors with distributed memory; the protocol is thus named *Distributed PCP* (DPCP). They use a separate processor, called a synchronization processor, to run critical sections for the same resource that are issued by tasks bound to different processors. They proved bounds for priority inversion and showed that DPCP avoids deadlocks.

For shared-memory multiprocessors, Rajkumar [84] proposed another extension of PCP, called *Multiprocessor PCP* (MPCP). Under MPCP, PCP is used in each processor to handle requests to resources that are only accessed by the tasks assigned to this processor, i.e., local resources; a set of rules for accessing global resources is detailed. Gai et al. [54] later proposed an extension of SRP, called *Multiprocessor SRP* (MSRP). MSRP supports partitioned scheduling; each processor uses an EDF scheduling policy and SRP for handling local resources. A job accessing a global resource will spin-wait if the lock for the global resource is already held by
another job on a different processor; otherwise, it acquires the lock and executes its critical section of the global resource non-preemptively.

After a few years without significant progress, a new surge of interest in multiprocessor synchronization protocols has lead to numerous notable results. Block et al. [20] introduced Flexible Multiprocessor Locking Protocol (FMLP) that supports, for the first time, G-EDF, in addition to P-EDF. FMLP also allows nested requests, i.e., a job can issue a request to a resource from inside a critical section for another resource; however, the set of resources in a nested request is treated as a group and protected by a group lock, i.e., once a job acquires the group lock, it has exclusive access to all resources in this group. It supports busy-waiting and suspending mechanisms for short and long critical sections, respectively. Through experiments, they showed that FMLP outperforms MSRP in term of the schedulability ratio for randomly generated task sets.

Brandenburg et al. [28] gave a formal definition of priority inversion for multiprocessors, and proved asymptotic bounds for blocking that can be experienced by a job scheduled under global or partitioned scheduling. They considered resources protected by suspension-based locks such as semaphores or mutexes, and introduced notions of suspension-oblivious and suspension-aware schedulability analysis. In suspension-oblivious analysis, suspensions are regarded as normal executions, i.e., a job still consumes processor cycles while it is suspended; whereas, suspension-aware analysis explicitly accounts for suspension durations. Based on the asymptotic bounds, they designed variants of an optimal locking protocol, named the $O(m)$ Locking Protocol (OMLP) for which blocking for each job is bounded by $O(m)$ critical sections from other jobs, where $m$ is the total number of processors.

Even though FMLP supports nested resource requests, it uses coarse-grained locking techniques to handle such requests, and thus limits the parallelism of resource access. Ward
et al. [97] improved the support for nested requests with a fine-grained locking protocol, called the *Real-time Nested Locking Protocol* (RNLP). RNLP is flexible in the sense that it can be used on global and partitioned scheduling, and with spin locks and suspension locks. They proposed several variants of RNLP which are asymptotically optimal with respect to suspension-oblivious and suspension-aware analyses.

With regard to spin locks, Wieder et al. [98], motivated by the inclusion of spin locks into the AUTOSAR standard [1], proposed a general framework for analyzing the worst-case blocking for P-FP. To analyze the worst-case blocking for a task, they formulate a Mixed-Integer Linear Program (MILP) problem that maximizes the blocking time experienced by any job of that task. A set of constraints is then constructed to eliminate blockings corresponding to impossible schedules of the task. The MILP problem can then be solved using existing optimization solvers. Finally, the obtained worst-case blocking time is incorporated into a response-time analysis for the task. This work not only gave specific insights on the performances of different types of spin locks, but also introduced a new technique for blocking analysis that can be applied in different setups.

In addition to mutual exclusion locks, other types of locks, including reader-writer and k-exclusion locks, and corresponding synchronization protocols also have been studied in the real-time systems literature [29, 30].

### 2.3 Real-Time Scheduling and Resource Sharing for Parallel Tasks

Over the last decade, the problem of scheduling parallel tasks in real-time systems has attracted a large body of work to design new scheduling algorithms as well as improve their
analyses. In this section, we present common parallel task models that have been used in the literature and summarize recent advances in scheduling parallel tasks with and without shared resources.

### 2.3.1 Parallel Task Model & Metrics

**Parallel Task Model:** Real-time scheduling of parallel tasks studies the problem of scheduling a set of parallel tasks with timing constraints on a multiprocessor platform. The models for parallel tasks have evolved from a simpler *synchronous* task model (also known as the *fork-join* task model in several papers) [5, 37, 72, 80, 88] to a more general *directed acyclic graph* (DAG) task model [14, 25, 88, 89].

![Figure 2.5: Synchronous parallel task model.](image-url)

In the synchronous task model, illustrated in Figure 2.5, each parallel task comprises a chain of segments, and each segment consists of a number of parallel nodes. Segments of a task are synchronized, meaning that all nodes belonging to a segment must complete before the succeeding segment can start. Nodes from the same segment can execute simultaneously on
multiple cores; however, each node can only execute sequentially. In practice, parallel-for loops written in languages such as OpenMP [23], Cilk Plus [40, 53], or Intel Threading Building Blocks [41] can be modeled using the synchronous task model: each parallel-for loop is presented by a segment, and each node in the segment represents an iteration.

![Figure 2.6: Example of DAG task.](image-url)

The DAG task model is a generalization of the synchronous task model which represents each task as a directed acyclic graph. Nodes (or vertexes or subtasks) of each DAG represent sequential computation units of the program. Edges between nodes represent the dependencies between them — a node can only start executing after all predecessors have completed. Figure 2.6 shows an example of a DAG task with 7 nodes. Two basic parameters are often used in analyses of DAG tasks: work and critical-path length. Essentially, work is the sum of execution times for all nodes of a DAG task, and critical-path length is the sum of execution times of nodes belonging to a longest path of the DAG. Similar to the synchronous task model, general parallel programs written in parallel languages such as OpenMP or Cilk Plus can be modeled using the DAG model. We formally define the DAG task model that we assume for this dissertation in Section 2.4.
**Schedulability Metrics:** Similar to multiprocessor scheduling, a resource augmentation bound can be used as a metric for theoretically measuring the schedulability performance of a scheduling algorithm. In addition to a resource augmentation bound, a new metric for real-time parallel scheduling, named a *capacity augmentation bound*, was introduced by Li et al. [75, 76]. Specifically, a scheduling algorithm $\mathcal{A}$ has a capacity augmentation bound of $b$ if it can schedule any task set that satisfies the following conditions on $m$ processors of speed $b$: (i) the total utilization is at most $m$, where $m$ is the number of processors of the system, and (ii) the critical-path length of each task is at most the task’s deadline. Similar to a utilization bound, a capacity augmentation bound can serve as a simple schedulability test for parallel tasks.

### 2.3.2 Real-Time Scheduling for Parallel Tasks

Since the purpose of parallel task models is to support real-time tasks that need to run on multiple processors simultaneously to satisfy their temporal constraints (since otherwise they could just be scheduled as sequential tasks), partitioned scheduling cannot be applied directly to schedule parallel tasks. Specifically, one must first assign nodes of parallel tasks to processors before applying partitioned scheduling. In contrast, global scheduling can be used more straightforwardly to schedule parallel tasks, since it does not prevent tasks from running in parallel. Consequently, there has been a large body of work addressing analysis of global scheduling for parallel tasks.

Saifullah et al. [88] were the first to consider DAG tasks. However, in that work they only allowed simplified DAGs in which each node has unit execution time. They showed that these simplified DAGs can be converted into synchronous tasks, which can then be analyzed using their analysis. Baruah et al. [14] formalized the DAG task model for recurrent real-time sporadic processes. They gave two schedulability tests with different run time complexity and
effectiveness for a single arbitrary-deadline DAG task scheduled under an EDF scheduling policy. Bonifaci et al. [25] extended Baruah et al.’s work [14] by allowing a task set to have more than one DAG task. They proposed different schedulability tests for G-EDF and G-DM for such task sets. Baruah [9] presented an improvement to the work by Bonifaci et al. [25] and showed that the new analysis dominates the one in [25].

Li et al. [75] proved a resource augmentation bound of $2 - 1/m$ for arbitrary-deadline DAG tasks and a capacity augmentation bound of $4 - 2/m$ for implicit-deadline DAG tasks scheduled under G-EDF. In later work, Li et al. [76] improved the capacity augmentation bound of G-EDF to $\frac{3+\sqrt{5}}{2}$ for implicit-deadline tasks. They also proved a capacity augmentation bound of $2 + \sqrt{3}$ for G-RM. Chwa et al. [36] proposed an analysis for G-EDF with constrained-deadline DAG tasks. Their analysis works by accounting for the worst-case interference that a job may experience due to workload from jobs of other tasks. Their experimental evaluation showed that their analysis outperforms previously proposed analyses for G-EDF.

Besides global scheduling, another widely studied technique for scheduling parallel tasks is decomposition-based scheduling. In this technique, each parallel task is decomposed into multiple sequential subtasks with their own release times and deadlines. After decomposition, the constructed sequential subtasks can be scheduled using a multiprocessor scheduling algorithm. Notable work for this analysis technique includes [67, 88, 89]. Qamhieh et al. [83] proposed a scheduling algorithm for DAG tasks in which a thread of each task is stretched to its deadline while the remaining subtasks are assigned individual release times and deadlines. Each stretched thread is then scheduled on an exclusive processor, and the remaining subtasks are scheduled together on the remaining processors.

**Federated Scheduling:** Federated scheduling, introduced by Li et al. [76], is a scheduling approach for parallel tasks in which tasks are classified as heavy tasks, i.e., tasks that require
executing on more than one processors simultaneously to meet their deadlines, and light

tasks, i.e., tasks that can meet their deadlines running sequentially. The two types of tasks

are scheduled differently as shown in Figure 2.7. In particular, each heavy task is allocated a

set of dedicated processors, and executes exclusively on its processors without interference; all

light tasks are scheduled sequentially using an existing multiprocessor scheduling algorithm,

such as those presented in Section 2.1, on the remaining processors. Typically, heavy tasks

can be scheduled with any work-conserving (or greedy) scheduling algorithm — algorithms

that do not leave processors idle if there exists work ready to be executed.

![Figure 2.7: Federated scheduling demonstration.](image)

We include in the following the lemma proved by Li et al. [76] for the number of processors

required by task $\tau_i$ to satisfy its deadline. Our analysis for parallel tasks with shared resources,

presented in Chapter 5, is developed using this lemma.

**Lemma 1.** Given a high-utilization, implicit deadline, parallel task ($u_i > 1$) with work

$C_i$, critical-path length $L_i$ and deadline $D_i$, $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ dedicated cores are sufficient to
guarantee that all jobs of this task will meet their deadlines, when scheduled with a greedy
scheduler.
Li et al. [76] proved a capacity augmentation bound of 2 for federated scheduling of implicit-deadline parallel tasks. In particular, they state that a task set $\tau$ is schedulable under federated scheduling provided that it satisfies the following two conditions: (i) the total utilization of the task set is no more than half of the computing capacity of the system, \( \sum_{i=1}^{m} u_i \leq \frac{m}{2} \), where $m$ is the total number of processors, and (ii) for all tasks, the critical path length is no more than half the corresponding relative deadline of that task, $\forall i, L_i \leq \frac{D_i}{2}$. These two conditions can serve as a sufficient test for federated scheduling; however, task sets that do not satisfy these conditions may still be schedulable under federated scheduling if after the high-utilization tasks are allocated their cores, the low-utilization tasks can be scheduled on the remaining cores.

Federated scheduling has been shown to be a promising approach analytically [8, 11, 12, 76]. Different federated-based scheduling algorithms have used different methods to compute the number of dedicated processors allocated to a heavy task. For example, Li et al. [76] allocated a heavy task $\tau_i \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ dedicated processors as stated in Lemma 1. For light tasks, almost all federated-based algorithms reuse existing multiprocessor scheduling algorithms to schedule them. The number of processors allocated to each heavy task in [76], however, can be pessimistic since it assumes the worst-case DAG for each heavy task. Succeeding work on this scheduling approach, including semi-federated scheduling and reservation-based federated scheduling, thus aims to alleviate the potential over-provisioning of heavy tasks [8, 11, 65, 96].

In Chapter 3, we present a novel algorithm for scheduling heavy tasks that reduces the number of exclusive processors assigned to each heavy task. We also give an overview of the state-of-the-art federated-based algorithms, and experimentally compare our algorithm with them.
2.3.3 Resource Sharing in Real-Time Parallel Tasks

Although there has been encouraging progress on scheduling independent parallel tasks as discussed previously, little work has been done with regard to scheduling parallel tasks that can access shared non-processor resources, such as memory buffers, shared data objects, or I/O devices. Scheduling parallel tasks with shared resources appears to be more challenging due to intra-task parallelism. This leads to intra-task contention for shared resources between different threads of a job, in addition to contention with threads from jobs of other tasks, i.e., inter-task contention. This adds a new layer of complexity to the problem since the blocking times caused by intra- and inter-task contentions of a job may affect each other. More work needs to be done to understand the problem fully, including clarifying the relationship between different types of contention, formally defining priority inversion blocking, deriving worst-case analyses, and developing effective synchronization protocols to reduce the effect of priority inversion.

In Chapter 5, we take a step toward these goals with an analysis for parallel tasks scheduled under federated scheduling for which shared resources are protected by spin locks. We include in the following the work on this topic of which we are aware. Holenderski et al. [62] proposed a multi-resource model for synchronous parallel tasks that is an abstraction for both processor and non-processor resources (such as memory, network buffers). Each parallel task is specified with a set of resource requirements. They presented a scheduling algorithm, named Parallel SRP (PSRP), that is an extension of MSRP for this model. Since their resource model is general, PSRP can be applied to the resource sharing problem that we are considering. Jiang et al. [64] recently proposed a synchronization protocol, named the Limited Pending Protocol (LPP), for a similar system model as the one considered in our contribution presented in Chapter 5, i.e., DAG tasks scheduled under federated scheduling. The difference is that
they assume that resources are protected by semaphores instead of spin locks. LPP limits the maximal number of requests to a semaphore by a job to a system-wide, configurable parameter; thus reducing the blocking time a job may experience.  

2.4 DAG Task Model

In this dissertation, we are particularly interested in the DAG task model since it is a more general model compared to other previously proposed models, and can be used to represent a wide range of parallel applications in practice. We now formally define the DAG task model as follows.

We consider a task set of \( n \) real-time, sporadic tasks scheduled preemptively on a homogeneous multiprocessor platform consisting of \( m \) identical processors (or cores). Each task \( \tau_i \) is modeled by a tuple \((G_i, D_i, T_i)\), where \( D_i \) and \( T_i \) are the relative deadline and minimum inter-arrival time (i.e., period) of \( \tau_i \), respectively. Task \( \tau_i \) is represented by a directed acyclic graph (DAG) \( G_i = (V_i, E_i) \) in which \( V_i \triangleq \{v_{i,1}, v_{i,2}, ..., v_{i,n_i}\} \) is the set of vertices and \( E_i \subseteq (V_i \times V_i) \) is the set of directed edges of \( \tau_i \). Vertices are also called subtasks or nodes. Each vertex denotes a sequential execution unit of the task and each edge \((v_{i,p}, v_{i,q}) \in E_i\) denotes the precedence constraint between vertices \( v_{i,p} \) and \( v_{i,q} \)— \( v_{i,p} \) must finish before \( v_{i,q} \) may start its execution.

A vertex with no incoming edges is called a source vertex, and a vertex with no outgoing edges is called a sink vertex. A sequence of vertices \((v_{i,k_j}, v_{i,k_{j+1}}, ..., v_{i,k_t})\), where \( v_{i,k_j} \) is a sink vertex and \((v_{i,k_p}, v_{i,k_{p+1}}) \in E_i, \forall j \leq p < t\), is a path of the DAG of \( \tau_i \) starting from \( v_{i,k_j} \).

Task \( \tau_i \) may release an infinite number of instances (jobs) and the difference between the release times of any two consecutive jobs of \( \tau_i \) must be at least \( T_i \). We use \( J_i \) to denote an arbitrary job of \( \tau_i \) which has release time \( r_i \) and absolute deadline \( d_i \). We consider

\(^2\)Jiang et al. [66] are publishing new work on this topic. At the time of this writing, however, the paper is not yet available.
**Figure 2.8:** Example of DAG task considered in this dissertation.

Constrained-deadline tasks, i.e., \( D_i \leq T_i \) for all \( \tau_i, 1 \leq i \leq n \). The worst-case execution time (WCET) of subtask \( v_{i,j} \) is denoted by \( C_{i,j} \). The WCET of the whole DAG task \( \tau_i \), denoted by \( C_i \), is simply the sum over the WCETs of its subtasks: \( C_i = \sum_{v_{i,j} \in V_i} C_{i,j} \). The value \( C_i \) is also called the work of \( \tau_i \). A path with the greatest length of \( \tau_i \) is called a critical-path of the DAG. The length of a path is the sum of the WCETs of all subtasks along the path. The critical-path length (or span) of \( \tau_i \) is denoted by \( L_i \). The utilization and density of \( \tau_i \) are denoted by \( u_i = \frac{C_i}{T_i} \) and \( \sigma_i = \frac{C_i}{\min(D_i, T_i)} = \frac{C_i}{D_i} \), respectively. Task \( \tau_i \) is called a heavy task if \( \sigma_i > 1.0 \) and a light task if \( \sigma_i \leq 1.0 \). Let \( n^h \) and \( n^l \) denote the number of heavy and light tasks in a task set respectively, i.e., \( n^h + n^l = n \). The normalized utilization of a task set is denoted by \( U : U = \frac{\sum_{\tau_i \in T} u_i}{m} \). Similarly, the normalized utilization of heavy and light tasks in a task set are denoted by \( U^h \) and \( U^l \), respectively. Figure 2.8 shows an example DAG task with 10 vertices. The work, critical-path length, and relative deadline of the task are \( C_i = 122, L_i = 36, \) and \( D_i = 44 \), respectively.

In the subsequent chapters, we assume this DAG task model unless stated otherwise.
Chapter 3

Efficient Deterministic Federated Scheduling for Parallel Tasks

As discussed in Chapter 2, federated scheduling is a promising approach for scheduling parallel real-time tasks. Federated scheduling, however, may over-provision computational resources to tasks due to its dedicated allocation of processors. In this chapter, we address this issue and propose a novel scheduling algorithm for parallel tasks that, when used together with a federated scheduling, notably improves its efficiency.

3.1 Introduction

The development of multicore processors enables applications with high computational demand to be deployed in modern real-time systems. Applications such as motion planning in autonomous vehicles [71], real-time hybrid structural simulation [48], and computer vision [46] require multiple processors simultaneously to meet their deadlines. In contrast to sequential tasks which only allow inter-task parallelism, such parallel applications not only
allow inter-task parallelism but they also enable *intra-task parallelism*, i.e., each instance of a task can execute on more than one processor at the same time. With the widespread use of multicore processors and the available parallel programming languages and concurrency platforms such as OpenMP [23] and Cilk Plus [40], ever more parallel applications are being deployed in modern real-time systems.

As discussed in Chapter 2, there are two major approaches for scheduling parallel tasks on multiprocessors: global scheduling and federated scheduling. In global scheduling, threads of a parallel task can execute on any processor and are allowed to migrate freely among processors even within a single instance of the task. On the other hand, federated scheduling, originally proposed by Li et al. [76], can be thought of as a generalization of partitioned scheduling for parallel tasks. Federated scheduling classifies parallel tasks as *heavy tasks*, i.e., tasks that need to execute on more than one processor, and *light tasks*, i.e., tasks that can execute sequentially and still meet their deadlines. Each of the heavy tasks is allocated a number of processors exclusively, i.e., it does not share the allocated processors with any other tasks. Light tasks are scheduled as sequential tasks on the remaining processors. Federated scheduling has been shown to be a promising approach for scheduling parallel tasks [8, 11, 12, 76] due to its analytical properties and ease of implementation in practice [77]. However, federated scheduling may suffer from resource waste, as processors that are dedicated to a heavy task cannot be shared with any other tasks, even if not fully exploited.

Recent work has attempted to address this resource waste problem, by either potentially reducing the number of processors allocated to heavy tasks [11] or increasing the ability for heavy tasks to share their processors with other tasks [65, 96]. For the latter direction, two approaches were proposed, namely semi-federated scheduling by Jiang et al. [65] and reservation-based federated scheduling by Ueter et al. [96]. We discuss these approaches in detail in Sections 3.2 and 3.5. We address the resource waste problem by reducing the
number of processors exclusively allocated to heavy tasks. Light tasks are then scheduled on
the remaining processors as in [8, 11, 12, 76].

The contributions presented in this chapter are as follows.

- First, we propose a novel algorithm to compute a deterministic schedule for each
  heavy task based on the internal graph structure of the task. The proposed algorithm
  efficiently exploits processors that are already allocated to a heavy task and only adds
  processors incrementally to the task when necessary. Consequently, the computed
  schedule is tight for the task, i.e., it leaves as few unused processor cycles as possible.
  We show that, theoretically the proposed algorithm requires no more processors than
  the previous work by Li et al. [76] for each heavy task. A federated scheduling algorithm
  is presented based on the deterministic scheduling algorithm.

- Second, we conduct an extensive experimental evaluation to evaluate the performance
  of the proposed algorithm with randomly generated parallel tasks. Experimental
  results show that the proposed algorithm significantly reduces the number of processors
  allocated to heavy tasks, and that our new federated scheduling algorithm outperforms
  the original federated scheduling algorithm [76] and the other state-of-the-art algorithms
  [11, 65, 96], including the semi-federated scheduling and reservation-based federated
  scheduling approaches, often by a large margin.

This chapter is organized as follows. In Section 3.2 we present the new deterministic scheduling
algorithm for heavy tasks and discuss its theoretical properties. Section 3.3 discusses our
overall federated scheduling algorithm that utilizes the algorithm in Section 3.2 for scheduling
heavy tasks. In Sections 3.4 and 3.5, we present experimental evaluations of our deterministic
algorithm and federated algorithm, respectively: these algorithms are compared with the
recent advances in federated-based scheduling. Lastly, Section 3.6 summarizes our work and
discusses how our work can be combined with previous work by Brandenburg et al. [32] to provide a practical and efficient solution to scheduling parallel tasks.

3.2 A Deterministic Scheduling Algorithm for DAG Tasks

Most previous work on federated scheduling ignores the structure of their DAGs when scheduling heavy tasks [8, 11, 12, 76]. Specifically, in [76] the number of dedicated processors allocated to $\tau_i$ is $\lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ — that is, it only depends on the basic parameters of $\tau_i$. $\tau_i$ is then scheduled using any work-conserving algorithm on its processors. The advantage of this method is that it is simple and does not require us to know the internal structure of the DAGs of the tasks. Thus, it allows the DAG of each task to vary in different releases as long as it satisfies the work and critical-path length bounds. However, this also means that the algorithm must assume the worst-case DAG for every task — the one that requires the largest number of processors — for the given work and critical-path length of the task. Hence, it risks over-provisioning the task, which leads to resource waste.

In [8, 11], the authors take the DAGs of the tasks into account to some extent. In particular, they first allocate $\tau_i$ a minimal number of processors equal to $\lceil \sigma_i \rceil$ and add one additional processor to $\tau_i$ only if the makespan of a schedule returned by Graham’s list scheduling [57] on the current number of processors is greater than the task’s deadline. Otherwise, $\tau_i$ is schedulable on the processors currently allocated to it using list scheduling. In the other words, they indirectly take the DAG of $\tau_i$ into consideration through the use of list scheduling to test whether $\tau_i$ can be scheduled successfully in each step.
Graham’s list scheduling, however, was not designed for real-time systems originally. Instead, its objective is to schedule the task on a given number of processors so that the task’s completion time (i.e., the time when its last subtask completes) is minimized. There is no notion of deadline in list scheduling. In contrast, for real-time systems, as long as tasks finish by their deadlines, how early they complete does not affect the temporal correctness of the system. In this section, we propose a new algorithm for scheduling heavy tasks which takes into account all information from the task, including its DAG, to compute the schedule for the task. The objective of this algorithm is to exploit the processors allocated to heavy task as efficiently as possible, thus minimizing the number of processors allocated to it. Algorithm 1 shows the pseudocode for the proposed algorithm.

Algorithm 1: Scheduling Algorithm for Parallel DAG Tasks

1: procedure DagSched(τᵢ, m)
2:   Pre-process τᵢ’s DAG
3:   \[ mᵢ \leftarrow \left\lceil \frac{Cᵢ}{Dᵢ} \right\rceil \]  \hspace{1cm} ▷ Assuming \( mᵢ \leq m \)
4:   while ScheduleCore(τᵢ, mᵢ) = False do
5:     if \( mᵢ > m \) then Return False
6:     end if
7:   end while
8:   Return True and the schedule for τᵢ
9: end procedure \hspace{1cm} ▷ Continued on the next page

In Graham’s list scheduling, once a subtask is scheduled it is executed non-preemptively until it completes. Upon its completion, succeeding subtasks are enabled subject to the precedence constraints defined by the task’s DAG. In contrast to Graham’s list scheduling, Algorithm 1 allows subtasks to be preempted and resumed at appropriate times. We call each such execution portion of a subtask a fragment of that subtask. We denote the \( k^{th} \) fragment of the \( j^{th} \) subtask for task \( τᵢ \) as \( vᵢ^{k,j} \). Fragment \( vᵢ^{k,j} \) of subtask \( vᵢ,j \) has execution time \( Cᵢ^{k,j} \). If subtask \( vᵢ,j \) is scheduled as \( nfrags(vᵢ,j) \) fragments, then we have \( \sum_{k=1}^{nfrags(vᵢ,j)} Cᵢ^{k,j} = Cᵢ,j \). Algorithm 1
procedure ScheduleCore(τᵢ, mᵢ)

currTime ← 0

readyQ ← \{insert source fragments\}

while readyQ ̸= ∅ do

minCores ← ⌈remainWork / Dᵢ − currTime⌉

if minCores > mᵢ then

  mᵢ ← mᵢ + 1

  Return False

end if

S ← \{vᵢ,j | vᵢ,j ∈ readyQ ∧ len(λᵢ,j) = Dᵢ − currTime\}

if |S| > mᵢ then

  mᵢ ← mᵢ + 1

  Return False

end if

listFrags ← S

readyQ ← readyQ \ S

while readyQ ̸= ∅ ∧ |listFrags| < mᵢ do

  Find vᵢ,j ∈ readyQ with greatest work(vᵢ,j)

  listFrags ← listFrags ∪ \{vᵢ,j\}

  readyQ ← readyQ \ \{vᵢ,j\}

end while

execTime ← \min_{vᵢ,j ∈ listFrags} Cᵢ,j

if vᵢ,j ̸= ∅ then

  execTime ← \min\{execTime, Dᵢ − currTime − len(λᵢ,j)\}

end if

Find vᵢ,j with Cᵢ,j > execTime

ψ ← \{second parts of the split fragments\}

readyQ ← readyQ ∪ ψ

readyQ ← readyQ ∪ \{new enabled fragments\}

Run fragments in listFrags for execTime units

currTime ← currTime + execTime

end while

Return True

end procedure
receives a task $\tau_i$ as its input and returns a schedule for $\tau_i$. The returned schedule consists of a chain of segments, each comprising fragments from different subtasks of $\tau_i$ executing in parallel.

Before the proposed algorithm computes the schedule for $\tau_i$, it pre-processes $\tau_i$’s DAG (line 2). In this pre-processing step, for each subtask $v_{i,j}$ of $\tau_i$ it computes two parameters. First, it computes the length of a longest path originating from $v_{i,j}$, i.e., the sum of the WCETs of the subtasks along that longest path starting from $v_{i,j}$ and ending at a sink subtask. Let $\lambda_{i,j}$ denote such a longest path for $v_{i,j}$. We use $\text{len}(\lambda_{i,j})$ to denote the length of the path $\lambda_{i,j}$. Second, it computes the total work of the subgraph rooted at $v_{i,j}$, i.e., the sum of the WCETs for all subtasks belonging to that subgraph. We use $\text{dag}(v_{i,j})$ to denote the subgraph of $\tau_i$’s DAG rooted at $v_{i,j}$ and $\text{work}(v_{i,j})$ to denote the work of $\text{dag}(v_{i,j})$. In other words, $\lambda_{i,j}$ is a critical-path of the subgraph $\text{dag}(v_{i,j})$, and $\text{len}(\lambda_{i,j})$ is its critical-path length. For example, the subgraph for $v_{i,0}$ in Figure 2.8 includes subtasks $v_{i,0}$, $v_{i,3}$, $v_{i,7}$, $v_{i,9}$, and thus has a total work of 52 time units. The notation of subgraph, critical-path, and work of a subgraph are also extended for fragments of subtasks. In particular, $\text{dag}(v_{i,j}^k)$ denotes the subgraph of $\tau_i$ rooted at fragment $v_{i,j}^k$ and $\text{work}(v_{i,j}^k)$ is the work of that subgraph. Similarly, a longest path starting from $v_{i,j}^k$ is denoted by $\lambda_{i,j}^k$ and its length is $\text{len}(\lambda_{i,j}^k)$.

After completing the pre-processing, Algorithm 1 proceeds by allocating task $\tau_i$ a minimal number of dedicated processors based on its density: $m_i = \lceil \frac{C_i}{D_i} \rceil$ (line 3). Note that this is the smallest number of processors that can possibly schedule $\tau_i$ successfully. The algorithm then executes a loop which gradually increases the number of processors $m_i$ allocated to $\tau_i$ (line 4). In each iteration, it attempts to compute a schedule for $\tau_i$ with the given value of $m_i$. If it fails, it increases $m_i$ by 1 and re-computes a schedule for $\tau_i$ with the new $m_i$. The algorithm terminates when a satisfying schedule for $\tau_i$, i.e., a schedule in which $\tau_i$ completes by its deadline, is obtained. For a given $m_i$, the algorithm maintains a queue $\text{readyQ}$ of
read fragments, i.e., the fragments that have their dependencies resolved (line 12). At the beginning, fragments corresponding to the source subtasks are inserted into the queue. These fragments have execution times equal to the WCETs of their corresponding subtasks. In a loop, it incrementally constructs a schedule for $\tau_i$ (lines 13-47). For each iteration, it determines which fragments will be scheduled and for how long. All chosen fragments will be scheduled for the same duration in each iteration. The algorithm keeps a variable $currTime$ denoting the time of the schedule it has constructed so far during the loop (line 11). At the beginning, $currTime$ is set to 0, and it is updated after each iteration.

At the beginning of each iteration of the inner loop, Algorithm 1 increases the number of processors $m_i$ allocated to $\tau_i$ if one of the following two cases happens. First, it computes an estimate for the minimum number of processors required to successfully schedule the remaining work of $\tau_i$’s DAG: \[
\left\lceil \frac{\text{remainWork}}{D_i - currTime} \right\rceil,
\]
where $\text{remainWork}$ is the remaining work of $\tau_i$ that has not yet been scheduled and $D_i - currTime$ is the amount of time the algorithm has to schedule that remaining work (line 14). This value is the ceiling of the density of the remaining work of $\tau_i$. If this value is greater than the current number of processors $m_i$ allocated to $\tau_i$, it increases $m_i$ by 1 and re-computes the schedule for $\tau_i$ with the new value for $m_i$ (lines 15-18).

The second case for which the algorithm increases the number of processors allocated to $\tau_i$ is as follows. At the beginning of each iteration, the algorithm computes a set $S$ of ready fragments for which the lengths of their longest paths are equal to the time left until $\tau_i$’s deadline. That is, \[S \triangleq \{ v_{i,j}^k \in \text{readyQ} | \text{len}(\lambda_{i,j}^k) = D_i - currTime \}\] (line 19). Each such fragment needs to execute immediately: otherwise $\tau_i$ will miss its deadline since the path corresponding to a fragment that is not scheduled immediately will run past the deadline. If the number of the fragments in $S$ is greater than $m_i$, the algorithm increases $m_i$ by 1 and re-starts the schedule construction with this new value of $m_i$, as in the first case (lines 20-23).
This increment is mandatory, due to the fact that there is no way the algorithm can schedule \( \tau_i \) successfully with the current value of \( m_i \).

If neither of the above two cases happens, the algorithm keeps the current value for \( m_i \) and determines at most \( m_i \) ready fragments to be scheduled. Since all fragments in the set \( S \) must be scheduled immediately, the algorithm takes all of them to schedule in this iteration (line 24). We call the list of fragments chosen to be scheduled \( \text{listFrags} \). For \( m_i - |S| \) processors left, it takes at most \( m_i - |S| \) ready fragments with the largest work values of the associated subgraphs, which are not in the set \( S \), to add to \( \text{listFrags} \) (lines 26-30). The intuition for this decision is that we want to prioritize the fragments with the greatest amount of pending work, so that as much work will be enabled for the succeeding iterations as possible. This gives us a better chance of scheduling more work in parallel in the subsequent iterations.

![Figure 3.1: The deterministic schedule computed by Algorithm 1 for the DAG in Figure 2.8.](image)

After the fragments being scheduled have been determined, the algorithm calculates the duration for the execution of those fragments, denoted by \( \text{execTime} \). Note that if the computed \( \text{execTime} \) is less than a fragment \( v_{i,j}^k \)'s execution time \( C_{i,j}^k \), the fragment is split into two smaller fragments. The first fragment is still denoted \( v_{i,j}^k \) but now has \( C_{i,j}^k = \text{execTime} \); this fragment is scheduled in this iteration. The second fragment, \( v_{i,j}^{k+1} \), has the remaining execution time, and is inserted back into the ready queue for subsequent iterations. First
of all, the execution length execTime is determined by taking the minimum execution time among all chosen fragments (line 31). In addition, the algorithm takes into account two other factors to calculate execTime. The intuition for these two factors is that after the chosen fragments have been scheduled for some time, some of the unchosen ready fragments may have greater total work (of their subgraphs) than the scheduled ones. Similarly, some of them may have longest paths that need to execute immediately because their length is equal to the time left until the task’s deadline. The algorithm therefore needs to choose a new set of ready fragments to schedule when that happens.

To compute the first factor, it picks from among the unchosen ready fragments the one with greatest longest-path length (line 32). Let $v^p_{u,w}$ denote such a fragment. That means $v^p_{u,w}$ has greatest $\text{len}(\lambda^p_{u,w})$ among all unchosen ready fragments. The chosen fragments thus can execute for at most $D_i - \text{currTime} - \text{len}(\lambda^p_{u,w})$ time units before $v^p_{u,w}$ must be scheduled (line 34). If there is no such fragment $v^p_{u,w}$, the algorithm just ignores this factor. Similarly, for the second factor the algorithm considers an unchosen fragment $v^q_{a,b}$ with greatest subgraph work (line 36) and a chosen fragment $v^r_{x,y}$ with smallest subgraph work among all chosen fragments such that $v^r_{x,y}$ is not in the set $S$ (line 37). Then, the difference between $\text{work}(v^r_{x,y})$ and $\text{work}(v^q_{a,b})$ gives us the second factor (line 39). The reason is that after that amount of time, $v^q_{a,b}$ has its subgraph work equal to $v^r_{x,y}$’s and we should decide whether $v^q_{a,b}$ should be scheduled next. In particular, the second factor is computed by: $\text{work}(v^r_{x,y}) - \text{work}(v^q_{a,b}) + 1$. We add one to break ties in case there are multiple fragments with the same subgraph work. If either $v^q_{a,b}$ or $v^r_{x,y}$ does not exist, we ignore this factor.

After these two factors have been determined, execTime is computed by taking the minimum of the execution times of the chosen fragments and the first and the second factors. For fragments that are scheduled for less than their execution times, the algorithm splits them and inserts their second parts into the ready queue as described above (lines 41- 43). For
fragments that complete, their corresponding subtasks are also finished. The algorithm therefore inserts new fragments corresponding to the enabled subtasks whose dependencies have been resolved into the ready queue (line 44). The chosen fragments now are scheduled onto the $m_i$ processors (line 45) and the current time for the schedule, $currTime$ is advanced by $execTime$ time units (line 46).

Example for Algorithm 1. Figure 3.1 shows the schedule computed by Algorithm 1 for the DAG task in Figure 2.8. The algorithm starts with $m_i = \lceil \frac{C_i}{D_i} \rceil = 3$ processors. Then, it inserts the source fragments \{${v^0_{i,0}, v^0_{i,1}, v^0_{i,2}, v^0_{i,4}, v^0_{i,6}, v^0_{i,8}}$\} into the ready queue ($readyQ$), each of which has execution time equal to the WCET of its subtask. At line 14, $minCores$ is exactly $m_i$, so it does not increase $m_i$. It then computes a set $S$ of ready fragments with longest-path lengths equal to $D_i - currTime = 44$. Since there is no such fragment, $S$ is empty, and there is no need to update $m_i$.

The algorithm then picks $m_i = 3$ fragments with greatest subgraph work from $readyQ$. The fragments in $readyQ$ have the following subgraph work. $v^0_{i,0}$'s subgraph contains \{$v^0_{i,0}, v^0_{i,3}, v^0_{i,7}, v^0_{i,9}$\} with $work(v^0_{i,0}) = C^0_{i,0} + C_{i,3} + C_{i,7} + C_{i,9} = 52$. Similarly, $v^0_{i,1}$'s subgraph contains \{$v^0_{i,1}, v^0_{i,5}, v^0_{i,7}$\} with work of 45, $v^0_{i,2}$'s subgraph contains $v^0_{i,2}, v^0_{i,7}$ with work of 27, $v^0_{i,4}$'s subgraph contains $v^0_{i,4}, v^0_{i,7}$ also with work of 27, $v^0_{i,6}$'s subgraph contains $v^0_{i,6}, v^0_{i,7}$ with work of 36, and $v^0_{i,8}$'s subgraph contains $v^0_{i,8}, v^0_{i,9}$ with work of 14. Three fragments with greatest subgraph work are $v^0_{i,0}, v^0_{i,1}$, and $v^0_{i,6}$, and thus they are scheduled in this iteration.

Algorithm 1 now computes $execTime$ for these fragments. First, $execTime$ is at most $\min\{C^0_{i,0}, C^0_{i,1}, C^0_{i,6}\} = 9$. Second, among the ready fragments that are not being scheduled \{$v^0_{i,2}, v^0_{i,4}, v^0_{i,8}$\}, $v^0_{i,2}$ (and $v^0_{i,4}$) has a longest path with greatest length, i.e., $len(\lambda^0_{i,2})(= 27) \geq len(\lambda^0_{i,4})(= 27)$ and $len(\lambda^0_{i,2})(= 27) \geq len(\lambda^0_{i,8})(= 14)$. The execution length is updated at line 34: $execTime = \min\{execTime, D_i - currTime - len(\lambda^0_{i,2})\} = \min\{9, 44 -
Lastly, among the fragments that are being scheduled $v_{i,6}^0$ has smallest subgraph work ($work(v_{i,6}^0) = 36$); among the fragments that are not being scheduled $v_{i,2}^0$ has greatest subgraph work ($work(v_{i,2}^0) = 27$). $execTime$ is thus updated as $execTime = \min\{execTime, work(v_{i,6}^0) - work(v_{i,2}^0) + 1\} = 9$.

After $execTime$ has been computed, it executes $v_{i,0}^0$, $v_{i,1}^0$, and $v_{i,6}^0$ on $m_i = 3$ processors for 9 units. Since $v_{i,1}$ has completed, $v_{i,5}$ is enabled and inserted into the ready queue as its first fragment $v_{i,5}^0$. The remaining parts of $v_{i,0}$ and $v_{i,6}$ are also inserted into the ready queue as fragments $v_{i,0}^1$ and $v_{i,6}^1$ respectively. The algorithm updates the current time for the schedule to 9 before executing the next iteration. It terminates and returns the computed schedule for $\tau_i$ when the ready queue is empty, meaning that all subtasks of $\tau_i$ have been scheduled successfully. The algorithm returns failure if at any point $m_i$ is greater than the total number of processors $m$.

Algorithms 1 only needs 3 processors to schedule the task. If we instead use the formula $\lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ for computing the required processors as in [76], we would need 11 processors. Algorithm 1 thus saves 8 processors in this example.

**Correctness.** We now show that Algorithm 1 computes a valid schedule for task $\tau_i$ in the sense that any schedule returned by the algorithm satisfies $\tau_i$’s deadline. We first prove the following lemma.

**Lemma 2.** For every iteration of the while-loop at line 13 of Algorithm 1, all ready fragments have their longest-path lengths less than or equal to $D_i - currTime$.

*Proof.* We prove this lemma by induction. For the first iteration, $currTime = 0$. Since the critical-path length of $\tau_i$, $L_i \leq D_i$, there is no ready fragments with longest-path length greater than $D_i - currTime = D_i$. 

43
Suppose the lemma holds for iteration $k$. That is, at iteration $k$, all ready fragments have their longest-path lengths less than or equal to $D_i - currTime$. We will prove that the lemma also holds for iteration $(k+1)$. For convenience, let $currTime_k$ and $currTime_{k+1}$ denote the values of $currTime$ at the beginning of iterations $k$ and $(k+1)$, respectively. Similarly, let $execTime_k$ denote the execution length computed for the iteration $k$. At the beginning of iteration $(k+1)$, each fragment $v_{i,a}^p$ that was scheduled at iteration $k$ either has completed, or its remaining $v_{i,a}^{p+1}$ has been inserted into the ready queue. For the latter case, $\text{len}(\lambda_{i,a}^{p+1}) = \text{len}(\lambda_{i,a}^p) - \text{execTime}_k \leq D_i - currTime_k - \text{execTime}_k = D_i - currTime_{k+1}$, since $\text{len}(\lambda_{i,a}^p) \leq D_i - currTime_k$. In the former case, after $v_{i,a}^p$ has completed it may enable some other subtasks to execute. For any subtask $v_{i,b}$ enabled by the completion of $v_{i,a}^p$, its first fragment $v_{i,b}^0$ has longest-path length $\text{len}(\lambda_{i,b}^0) \leq \text{len}(\lambda_{i,a}^p) - \text{execTime}_k \leq D_i - currTime_k - \text{execTime}_k = D_i - currTime_{k+1}$. Hence the lemma holds in both cases.

Let $v_{i,x}^q$ denote a fragment with greatest longest-path length among all ready fragments that were not chosen to be scheduled at iteration $k$. If there is no such fragment, then all ready fragments in iteration $k$ were scheduled and the lemma holds as discussed above. Otherwise, we have $\text{execTime}_k \leq D_i - currTime_k - \text{len}(\lambda_{i,x}^q) \iff \text{len}(\lambda_{i,x}^q) \leq D_i - currTime_k - \text{execTime}_k \iff \text{len}(\lambda_{i,x}^q) \leq D_i - currTime_{k+1}$. Since every other fragment $v_{i,y}^r$ that was not scheduled at iteration $k$ has $\text{len}(\lambda_{i,y}^r) \leq \text{len}(\lambda_{i,x}^q)$, the lemma holds for all ready fragments that were not scheduled at iteration $k$.

We now can prove that if Algorithm 1 returns a schedule for $\tau_i$, then it is a valid schedule.

**Theorem 3.** If Algorithm 1 returns a schedule for $\tau_i$, then this schedule satisfies $\tau_i$’s deadline.

**Proof.** Suppose the algorithm returns a schedule that misses $\tau_i$’s deadline. Then for the last iteration of the while-loop at line 13 before the algorithm terminates, there must be at least a ready fragment $v_{i,j}^k$ such that $\text{len}(\lambda_{i,j}^k) > D_i - currTime$. This contradicts Lemma 2. \qed
Theorem 2 in [76] has proved that an implicit-deadline DAG task $\tau_i$ can be scheduled on $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ processors using any work-conserving scheduler. This also applies to constrained-deadline DAG tasks since $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ processors are sufficient to guarantee that it takes at most $D_i$ time units to complete $\tau_i$. Algorithm 1 is also a work-conserving algorithm since it does not leave any processor idle if there are some fragments ready to be executed. We thus can bound the number of processors required by Algorithm 1 in the following theorem.

**Theorem 4.** The number of processors required by Algorithm 1 for task $\tau_i$ is at most $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$.

**Proof.** Suppose that Algorithm 1 returns a schedule for $\tau_i$ that requires $m_i^* > \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ processors. The algorithm starts with $m_i = \left\lceil \frac{C_i}{D_i} \right\rceil < \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ processors. In each subsequent call at line 4, it increases $m_i$ by 1. Thus at some point, $m_i$ is set to $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$. Since Algorithm 1 is work-conserving, it would have scheduled $\tau_i$ successfully using that number of processors and the returned $m_i$ would have been $\left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil < m_i^*$. This contradicts the hypothesis. \qed

Even though Theorem 4 does not show a theoretical improvement of Algorithm 1 in terms of the bound on the number of processors required, this bound is loose and can be improved.

We show in our experimental evaluation (Sections 3.4 and 3.5) that Algorithm 1 can perform much better than existing federated scheduling algorithms. Before we present our evaluation, we discuss the time complexity of Algorithm 1.

**Time Complexity.** In the pre-processing step, to compute the longest-path length of a subtask, we can apply a variant of the shortest paths algorithm for DAGs, presented in [39] (Section 24.2). This algorithm has a time complexity of $O(|V_i| + |E_i|)$. To compute the subgraph work of a subtask $v_{i,j}$, we can use breadth-first search or depth-first search to find all subtasks reachable from $v_{i,j}$ and take the sum of their WCETs. This also takes $O(|V_i| + |E_i|)$ for each subtask. Thus the total time complexity for pre-processing is $O(|V_i|(|V_i| + |E_i|))$. 

45
The while-loop at line 4 runs $O(m)$ times. In the worst case, each iteration of the while-loop at line 13 advances $\text{currTime}$ 1 time unit. Thus this loop executes $O(D_i)$ iterations. We can implement the ready queue ($\text{readyQ}$) as a priority-queue with keys corresponding to the subgraph work of each ready fragment. At line 12, $\text{readyQ}$ is constructed with at most $|V_i|$ source subtasks, and thus it takes $O(|V_i|)$ time.

To compute the set $\mathcal{S}$ at line 19, we can iteratively pop each fragment from $\text{readyQ}$, add it to $\mathcal{S}$ if its longest-path length is equal to $D_i - \text{currTime}$, or add it to a temporary set if its longest-path length is less than $D_i - \text{currTime}$. After all ready fragments have been examined, the fragments in the temporary set are inserted back into $\text{readyQ}$. This step takes $O(|V_i|\log|V_i|)$ time. Similarly, the while-loop at lines 26 - 30 takes $O(m\log|V_i|)$ time. Line 32 takes $O(|V_i|)$ time by simply examining the ready but unscheduled fragments and computing the maximum longest-path length for them. Line 36 takes $O(1)$ time by accessing the top fragment of $\text{readyQ}$. The computation for line 37 can be embedded in the while-loop at lines 26 - 30 and thus does not take additional time. Lines 41 - 44 take $O(|V_i|\log|V_i|)$ time since there are at most $|V_i|$ ready fragments inserted to $\text{readyQ}$. In total, Algorithm 1 takes $O(D_i m\log|V_i||V_i| + m + |V_i|^2 + |V_i||E_i|)$ time. That is, the algorithm has pseudo-polynomial time in the deadline of $\tau_i$, which is of similar complexity to response-time analysis.

Figure 3.2: A schedule computed by the algorithm in [11] for the DAG in Fig. 2.8.
Optimization. We can reduce unnecessary migrations for fragments of the same subtasks by a small adjustment to Algorithm 1. After the fragments being scheduled for an iteration have been determined, we can assign them to the processors so that for each fragment $v^{k}_{i,j}$ being scheduled, if its preceding fragment $v^{k-1}_{i,j}$ was scheduled in the immediately preceding iteration, then $v^{k}_{i,j}$ is scheduled on the same processor as $v^{k-1}_{i,j}$’s. Consecutive fragments for the same subtask are now executed on the same processor and thus can be merged into a larger fragment. At runtime, these merged fragments are scheduled instead of the smaller ones. For instance, in Figure 3.1 all fragments of $v^{0}_{i,0}$ ($v^{0}_{i,0}, v^{1}_{i,0}, v^{2}_{i,0}, v^{3}_{i,0}, v^{4}_{i,0}, v^{5}_{i,0}, v^{6}_{i,0}, v^{7}_{i,0}$) are scheduled consecutively on processor P2 from time $t = 0$ to time $t = 18$. These fragments are then merged into a single fragment with WCET of 18, i.e, the WCET of $v^{0}_{i,0}$. Thus, at runtime subtask $v^{0}_{i,0}$ is scheduled continuously on processor P2 without migrations.

3.3 A Federated Scheduling Algorithm

The pseudocode for our federated scheduling algorithm is shown in Algorithm 2. The heavy tasks are scheduled by Algorithm 1 on their dedicated processors. The light tasks are scheduled as sequential tasks on the remaining processors using any existing multiprocessor scheduling algorithm such as P-EDF, P-FP, or G-EDF. The algorithm returns failure if Algorithm 1 fails to schedule any heavy task (line 4) or if the light tasks cannot be scheduled on the remaining processors (line 9). Otherwise the task set is scheduled successfully by the algorithm.

In Section 3.5, we evaluate Algorithm 2 and compare it with the state-of-the-art federated scheduling algorithms, including semi-federated scheduling [65] and reservation-based federated scheduling [96]. Before that, we evaluate the performance of Algorithm 1 for scheduling heavy DAG tasks in Section 3.4.
Algorithm 2 Federated Scheduling for DAG Tasks

1: procedure FedSched(τ, m)
2: $m_r \leftarrow m$
3: for Each heavy task $\tau_i$ do
4: if DAGSCHED($\tau_i$, $m_r$) = False then ▷ Algo. 1
5: Return Failure
6: end if
7: $m_r \leftarrow m_r - m_i$ ▷ $\tau_i$ is allocated $m_i$ processors
8: end for
9: if Scheduling light tasks on $m_r$ processors fails then
10: Return Failure
11: end if
12: Return Success
13: end procedure

3.4 Evaluation for Heavy Tasks

In Section 3.2 we described an improvement to the processor allocation algorithms for heavy tasks proposed by Baruah [8, 11]. Figure 3.2 shows a possible schedule returned by the algorithm in [11] for the task in Figure 2.8. This algorithm requires 4 processors to schedule the task compared to 3 processors needed by our algorithm. In general, the schedule returned by this algorithm, and thus the number of processors required, depends on the specific order of subtasks in list scheduling. However, for this example 4 is the smallest number of processors the algorithm in [11] can get. It turns out that this algorithm for scheduling heavy tasks works quite well in practice, as we show in Section 3.5. In this section, we compare our proposed algorithm, denoted by PRO, with the algorithm in [11], denoted by BAR, for scheduling heavy DAG tasks. The goal is to observe experimentally how much Algorithm 1 improves in terms of the number of processors allocated to heavy task. We denote the method by Li et al. [76] for allocating processors to heavy tasks, i.e., task $\tau_i$ is allocated $\lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ processors, as LI.
Table 3.1: Comparison to BAR and LI for Heavy DAG Tasks

<table>
<thead>
<tr>
<th>Edge Probability Threshold $p$</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Against BAR [11]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fewer</td>
<td>43.55%</td>
<td>42.81%</td>
<td>40.6%</td>
<td>44.47%</td>
<td>48.28%</td>
</tr>
<tr>
<td>More</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Against LI [76]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fewer</td>
<td>83.93%</td>
<td>82.03%</td>
<td>80.25%</td>
<td>76.91%</td>
<td>75.21%</td>
</tr>
<tr>
<td>More</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge Probability Threshold $p$</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Against BAR [11]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fewer</td>
<td>39.67%</td>
<td>28.35%</td>
<td>21.73%</td>
<td>22.14%</td>
</tr>
<tr>
<td>More</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Against LI [76]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fewer</td>
<td>74.45%</td>
<td>76.26%</td>
<td>92.04%</td>
<td>84.84%</td>
</tr>
<tr>
<td>More</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

We used the Erdős-Rényi $G(n_i, p)$ method for generating DAGs [38]. In this method, $n_i$ is the number of vertices and $p$ is a probability threshold used to determine whether a directed edge between a pair of vertices is added. For each pair of vertices, a real number is drawn uniformly at random from the range $[0, 1]$ and if the number is less than $p$, an edge is added between the two vertices. The obtained DAG may not be connected; in that case, we added a minimal number of edges to make it weakly connected. For each task, the number of vertices was chosen uniformly at random in the range $[50, 250]$. The worst-case execution time of each vertex was also drawn uniformly at random in the range $[50, 100]$. The relative deadline for $\tau_i$ was generated uniformly at random in the range $[L_i, C_i)$. We varied the probability threshold $p$ in the range $[0.1, 0.9]$ with a step of 0.1. For each setting, we generated 10,000 tasks and scheduled them with our algorithm and the algorithm in [11]. The numbers of processors required by PRO, BAR, and LI were recorded.
Table 3.1 shows the percentages of the generated heavy tasks for which PRO needed fewer or more processors than BAR and LI. As we can see, PRO required fewer processors than BAR for $21\% - 48\%$ of the generated tasks. As $p$ increases, the generated tasks become more sequential and thus there is less room for the proposed algorithm to improve. There were still over $22\%$ of the tasks for which at least 1 processor was saved when $p = 0.9$. Notably, there was no task for which PRO required more processors than BAR. Since allocating processors using the formula $\lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ is more susceptible to over-provisioning, it is unsurprising that PRO significantly outperforms LI as $74\% - 92\%$ of the generated tasks needed fewer processors.
under PRO. Similar to the comparison with BAR, there was no task for which PRO required more processors than LI.

Figure 3.3 takes a deeper look at the extent of improvement of PRO compared to BAR. For each value of \( p \), a box plot is drawn for all tasks for which PRO required fewer processors than BAR. Each data point demonstrates the ratio of the numbers of processors returned by PRO and BAR; a smaller ratio means a greater extent of improvement by PRO. The median ratios were 0.8, 0.75, 0.67. That is 50% of the considered tasks were saved one fifth, one quarter, or one third (or more) processors respectively, compared to when BAR is used.

Figure 3.4: Ratio of schedulable task sets for varying total utilization and number of processors.

(a) \( m = 16, n = n^h = 5 \).

(b) \( m = 32, n = n^h = 10 \).

(c) \( m = 64, n = n^h = 15 \).
We also conducted experiments with task sets consisting of heavy tasks for systems with \( m = \{16, 32, 64\} \) processors. For each value of \( m \), we varied the normalized total utilization \( U \) in the range \([0.2, 1.0]\) with a step of 0.05. For \( m = \{16, 32, 64\} \), each task set consisted of \( n = \{5, 10, 15\} \) tasks respectively, except when \( U \) was too small to generate \( n \) heavy tasks. In those cases, we generated smaller numbers of tasks. For instance, when \( m = 16 \) and \( U = 0.2 \), we generated 2 tasks per task set. DAG tasks were generated using the Erdős-Rényi method as described above with \( p \) set to 0.2. For each task set, we used the RandFixedSum algorithm [93] to generate individual utilizations for the tasks uniformly in the range \([1.1, U \times m]\). For each value of \( U \), we generated 100 task sets and recorded the ratios of task sets that were schedulable under PRO, BAR, and LI. Figure 3.4 shows the result for this experiment. We can see that PRO dominates both BAR and LI. As \( m \) and \( n \) increase, the differences in performance between PRO and BAR and LI increase. For \( m = 64 \), there was a big gap between PRO and BAR (and LI). This is because as \( m \) and \( n \) increase, the chance that PRO can save processors for some tasks in each task set, and thus the chance that all tasks get sufficient processors, increases. Again there was no task for which PRO required more processors than BAR or LI.

In this section, we have focused on comparing PRO with BAR and LI for heavy DAG tasks and showed that PRO outperforms both BAR and LI by a large margin. In Section 3.5, we discuss the performance of our federated scheduling algorithm (Algorithm 2) that uses PRO and compare it with the state-of-the-art federated-based scheduling algorithms when both heavy and light tasks are involved.
3.5 Evaluation Versus the State-of-the-Art

3.5.1 The State-of-the-Art Federated-Based Scheduling

Jiang et al. [65] and Ueter et al. [96] proposed approaches to address resource waste in the federated scheduling paradigm. In [65] the authors introduced the semi-federated scheduling algorithm. Instead of allocating task $\tau_i$ with processing capacity requirement of $x + \epsilon = \frac{C_i - L_i}{D_i - L_i}$ (where $x = \lfloor \frac{C_i - L_i}{D_i - L_i} \rfloor$ and $0 \leq \epsilon = \frac{C_i - L_i}{D_i - L_i} - x < 1$) $\lceil x + \epsilon \rceil$ processors as in [76], they allocate $\tau_i x$ processors. The remaining fraction $\epsilon$ of the processing capacity requirement is scheduled together with the light tasks on the remaining processors.

To realize this idea, the fractional part $\epsilon$ is serviced by a container task with load bound equal to $\epsilon$. A runtime dispatcher for each DAG determines when a subtask of the DAG is serviced by the container task and for how long. They proposed two variants for the semi-federated approach. In the first variant, there is one container task for a heavy task (if $\epsilon > 0$) with load bound equal to $\epsilon$. The container tasks for the heavy tasks and the light tasks are scheduled on the remaining processors using P-EDF with a Worst-Fit bin packing heuristic [68]. For a given processor and a container task or a light task, the task can be assigned to the processor if the sum of the total load of all tasks that are already assigned to the processor and the load of the considered task $\leq 1.0$ (the load for a light task is equal to its density). In the second variant, there are two container tasks for a heavy task with the sum of their load bounds equal to $\epsilon$. As the load bounds for container tasks get smaller, this may improve schedulability for the container tasks and light tasks.

In [96] the authors proposed a reservation-based federated scheduling approach that generalizes the federated scheduling paradigm. Instead of allocating heavy task $\tau_i$ a set of dedicated
processors, they allocate $\tau_i$ a set of dedicated reservation servers. Each light task is assigned one reservation server with a budget equal to the task's work. The reservation servers for the tasks are scheduled using an existing multiprocessor scheduling algorithm for sequential tasks, such as P-EDF, P-DM or G-EDF. They derived a condition for the total budget of all reservation servers assigned to a DAG task that guarantees its feasibility. As long as the total budget for $\tau_i$ is sufficient and its reservation servers are scheduled to meet their deadlines under the applied multiprocessor scheduling algorithm, $\tau_i$ is guaranteed to meet its deadline.

The authors proposed two algorithms for assigning reservation servers to tasks. In the first algorithm, R-MIN, each light task is assigned one reservation server with a budget equal to the task’s work; each heavy task is assigned $m_i = \lceil \frac{C_i - L_i}{D_i - L_i} \rceil$ reservation servers, each with an identical budget of $L_i + \frac{C_i - L_i}{m_i}$. In the second algorithm, R-EQUAL, tasks are classified as heavy or light tasks depending on whether or not $C_i > \gamma L_i$, where $\gamma = \min_{\tau_i} \frac{D_i}{L_i}$ is a common stretch ratio for all tasks. Based on this classification, each light task is then assigned one reservation server with a budget identical to the work of the task. Each heavy task is assigned $m_i = \lceil \frac{C_i - L_i}{L_i(\gamma - 1)} \rceil$ reservation servers, each with a budget $\gamma L_i$. The authors introduced an algorithm that dynamically adapts the number of reservation servers assigned to a heavy task if one of its current servers fails to be partitioned. Specifically, if a reservation server for $\tau_i$ fails to be partitioned, the number of servers assigned to $\tau_i$ is increased by 1 and the budget of each server is reduced accordingly. Since the smaller reservation servers are easier to partition, this improves the schedulability of the system.

In this section, we evaluate our federated scheduling algorithm in comparison to the semi-federated scheduling [65] and reservation-based federated scheduling [96] approaches. We also include the algorithm proposed by Baruah [11], as Section 3.4 suggests that it is also a competitive alternative.
(a) $m = 16$, $n^h \in [1, 4]$, $U^h = 0.5U$, $n^l = 20$.
(b) $m = 32$, $n^h \in [2, 7]$, $U^h = 0.5U$, $n^l = 40$.
(c) $m = 64$, $n^h \in [4, 14]$, $U^h = 0.5U$, $n^l = 80$.

Figure 3.5: Ratio of schedulable task sets for varying total utilization $U$.

3.5.2 Experimental Evaluation

For all federated-based scheduling approaches being considered, we used P-EDF for scheduling sequential tasks including light tasks, container tasks (for semi-federated scheduling), and reservation servers (for reservation-based federated scheduling). We considered three bin packing heuristics for partitioning: Worst-Fit (WF), Best-Fit (BF), and First-Fit (FF). For testing whether a sequential task can be assigned to a processor, two uniprocessor EDF schedulability tests were adopted. The first test computes the sum of the loads, i.e., densities
for light tasks and reservation servers and load bounds for container tasks, from all tasks already assigned to this processor and the load of the considered task. If the sum is ≤ 1.0 then the task can be assigned to this processor. We denote this test DEN. The second schedulability test, introduced by Baruah et al. [15], is based on an approximation to the demand bound function [13] which captures the maximum cumulative execution requirement generated by a task in a given time interval. We denote this test DBF. In general, Best-Fit bin packing combined with the DBF test produced the best results in our experiments.
In [96] the authors show that their approach performs best when R-MIN is used for assigning reservation servers together with the Best-Fit bin packing heuristic and the DBF schedulability test. We hence included this variant of reservation-based federated scheduling in our figures and denote it RESV. For semi-federated scheduling, denoted by SEMI, we used Worst-Fit bin packing together with DEN for a schedulability test that is similar to one in [65]. For the federated scheduling algorithm proposed by Baruah [11] (denoted by BAR), the results for Best-Fit bin packing with the DBF test are reported since they performed the best for BAR. For our algorithm, we include two variants: one for Worst-Fit bin packing with the DEN
test, and the other for Best-Fit bin packing with the DBF test. These variants are denoted as PRO-WF-DEN and PRO-BF-DBF, respectively.

Similar to the methods described in Section 3.4, we applied the Erdős-Rényi method [38] to generate DAGs. For each task, the number of subtasks was uniformly chosen in the range $[50, 250]$. Each subtask has a WCET picked uniformly in the range $[50, 100]$ and the probability threshold for adding edges $p$ was set to 0.2. The following parameters were controlled: the normalized utilization $U$, the normalized utilization of heavy tasks $U^h$, the numbers of heavy tasks $n^h$ and light tasks $n^l$. For given values of $m, U, U^h, n^h, n^l$, we used the RandFixedSum algorithm [93] to generate individual utilizations uniformly in the ranges $[1.1, U^h \times m]$ for heavy tasks and $[0.01, 0.9]$ for light tasks, respectively. For each data point, 500 task sets were generated and the ratios of schedulable task sets were recorded. The experiments were conducted for systems with $m = \{16, 32, 64\}$ processors.

In Figure 3.5 we varied $U$ in the range $[0.2, 1.0]$ with a step of 0.05. The ratio of $\frac{U^h}{U}$ was set to 0.5, i.e., heavy tasks account for half of the total utilization in each task set. For $m = \{16, 32, 64\}$, we generated maximum of $\{4, 7, 14\}$ heavy tasks, respectively (for small $U$ we generated smaller number of heavy tasks accordingly). For light tasks, $n^l = \{20, 40, 80\}$ respectively. We saw from this experiment that PRO-BF-DBF dominated all other approaches. Especially, it outperformed semi-federated and reservation-based federated scheduling by a large margin. The reason for PRO dominating SEMI is because for each task with processing capacity of $x + \epsilon$, SEMI saves at most 1 processor regardless of the value of $x + \epsilon$. For task sets with many heavy tasks and/or large heavy tasks, the effectiveness of SEMI is reduced. In RESV, the adoption of reservation servers allows heavy tasks and light tasks to share processors. However, to enable that flexibility the total budget of the servers assigned to a task is over-provisioned. In particular, the total budget for $\tau_i$ must be $\geq L_i \cdot (m_i - 1) + C_i > C_i$, 58
where $m_i$ is the number of reservation servers of $\tau_i$ (Equation 1 in [96]). Similar to [96], we observe that RESV outperforms SEMI in our experiments.

BAR also outperforms RESV and SEMI significantly. This surprising result shows that federated scheduling is still very competitive if the scheduling for heavy tasks is designed carefully. As $m$ (and $n$) increases the gap between the group (PRO-BF-DBF, PRO-WF-DEN, BAR) and the group (RESV, SEMI) gets bigger. This is because as $m$ increases, the number of processors saved by PRO-BF-DBF, PRO-WF-DEN, and BAR increases while the effectiveness of RESV and SEMI is reduced due to their over-provisioning. The gap between PRO-BF-DBF and BAR also increases as $m$ increases for the same reason as was discussed in Section 3.4. Best-Fit bin packing combined with the DBF schedulability test performs much better than the combination of Worst-Fit with the DEN test. This is manifested in the difference between the performances of PRO-BF-DBF and PRO-WF-DEN.

In Figure 3.6, for each $m$ we kept $U$ and varied the proportion of heavy tasks $\frac{n^h}{U}$ in the range $[0.0, 1.0]$ with a step of 0.1. The maximum value of $n^h$ in each task set is $\{4, 8, 14\}$ and $n^l$ is $\{20, 40, 80\}$ for $m = \{16, 32, 64\}$, respectively. For $U^h = 0$, all tasks are light tasks and for $U^h = 1.0$, all tasks are heavy tasks. We observe similar trends between different approaches as discussed above. The fluctuation of PRO-BF-DBF, PRO-WF-DEN, and BAR is due to the integral processor allocation for heavy tasks. Consequently, there are cases when $U^h$ increases but the heavy tasks do not require additional processors while $U^l$ reduces and the light tasks become easier to schedule. In addition, as the proportion of heavy tasks increases, the task sets become harder to schedule and the performances of all approaches decrease. However, RESV and SEMI have much higher rates of reduction compared to PRO-BF-DBF, PRO-WF-DEN, and BAR; PRO-BF-DBF is the most stable one among them. Also, as $\frac{U^h}{U}$ increases, the gap of performance between PRO-BF-DBF and BAR gets larger. This shows
the efficiency of Algorithm 1 for scheduling heavy tasks. When \( \frac{v^h}{U} = 1 \), PRO-WF-DEN is the same as PRO-BF-DBF since there is no light task.

Figure 3.7 shows the result for varying \( \frac{D_i}{T_i} \). We generated 500 task sets with normalized density fixed to 0.9 for all values of \( m \). In each task set, half of the total density was given to heavy tasks. For \( m = \{16, 32, 64\} \), \( n^h = \{4, 7, 14\} \) and \( n^l = \{20, 40, 80\} \) respectively. We used the RandFixedSum method to generate individual densities for all tasks. The range for \( \frac{D_i}{T_i} \) was varied in the set \( \{[0.01, 0.05), [0.05, 0.1), \ldots, [0.95, 1.0)\} \). For a given range, \( \frac{D_i}{T_i} \) was uniformly chosen in that range and \( T_i \) was computed accordingly based on \( D_i \). As \( \frac{D_i}{T_i} \) increases, the total utilization also increases and the task sets become harder to schedule. Again, PRO-BF-DBF outperforms other methods. RESV performs well for small \( \frac{D_i}{T_i} \) but declines quickly when \( \frac{D_i}{T_i} \) gets larger. The acceptance ratios of PRO-WF-DEN and SEMI do not change as \( \frac{D_i}{T_i} \) increases since they use the DEN test for light tasks, which only depends on the densities of the tasks, and for heavy tasks the number of processors allocated to them is not affected.

### 3.6 Summary

We have proposed a novel algorithm to compute a deterministic schedule for each heavy DAG task. The algorithm takes into account all information about the DAG task, including its internal graph structure and its deadline to compute its schedule. It efficiently exploits all dedicated processors allocated to each heavy task. Consequently, the number of processors allocated to each heavy task reduces significantly compared to the state-of-the-art alternatives. A federated scheduling algorithm for parallel DAG tasks was then proposed that employs the deterministic algorithm for scheduling heavy tasks and schedules all light tasks on the remaining processors using an existing multiprocessor scheduling algorithm.
Experimental evaluation results show that our new federated scheduling algorithm outperforms semi-federated scheduling and reservation-based federated scheduling by a large margin. Surprisingly, a simpler federated scheduling algorithm proposed by Baruah [11], even though it is dominated by our new algorithm, also outperforms the more sophisticated approaches above. Another advantage of our federated scheduling algorithm (and the one in [11]) is that it can be implemented easily and efficiently in practice due to its simplicity.

The proposed algorithms can be combined with the work previously published by Brandenburg et al. [32] as follows. Heavy tasks are scheduled by Algorithm 1 on their dedicated sets of processors; light tasks are scheduled on the remaining processors using the techniques presented in [32]. In [32], the authors show that they can achieve near optimal schedulable utilization (over 99%) for sequential tasks on multiprocessors using known techniques, including semi-partitioned scheduling, reservations, period transformation and new heuristics for task placement. Since both approaches are experimentally shown to be effective for their targeted tasks, this combination appears to be a promising strategy for scheduling parallel real-time tasks.
Chapter 4

Analysis of Global Fixed-Priority for Parallel Tasks

In Chapter 3, we have presented a deterministic algorithm for scheduling DAG tasks that can be used together with the federated scheduling approach. In this chapter, we consider the global scheduling approach for DAG tasks. In particular, we present an analysis of Global Fixed-Priority (G-FP) for the same DAG task model as assumed in Chapter 3.

4.1 Introduction

As discussed in Chapter 2, much effort has been made to develop analysis techniques and schedulability tests for scheduling parallel real-time tasks under global scheduling algorithms such as G-EDF or G-DM. Schedulability analysis for parallel tasks is inherently more complex than for conventional sequential tasks. This is because intra-task parallelism is allowed within

\[ \text{Contents of this chapter previously appeared in the following paper:} \]
individual tasks, which enables each individual task to execute simultaneously upon multiple processors. The parallelism of a task can also vary during its execution, as it depends on the precedence constraints imposed on the task. Consequently, this raises questions of how to account for inter-task interference caused by other tasks on a task and intra-task interference caused by different threads of the same task.

In this chapter, we consider parallel tasks scheduled under Global Fixed-Priority (G-FP); each task is represented by a Directed Acyclic Graph (DAG) (see Section 2.4). Our analysis is based on the concepts of critical interference and critical chain [36, 37, 81], which allow the analysis to focus on a special chain of sequential segments of each task, and hence enable us to use techniques similar to the ones developed for sequential tasks [6, 17, 18, 19].

The contributions of this chapter are as follows.

- We summarize the state-of-the-art analyses for G-FP and highlight their limitations, specifically for the calculation of interference of carry-in and carry-out jobs.
- We propose a new technique for computing upper-bounds on carry-out workloads, by transforming the problem into an optimization problem that can be solved by modern optimization solvers.
- We present a response-time analysis, using the workload bound computed with the new technique. Experimental results for randomly generated DAG tasks confirm that our technique dominates existing analyses for G-FP.

The rest of this chapter is organized as follows. Section 4.2 reviews the concepts of critical interference and critical chain and discusses a general framework to bound the response-time of a task. Section 4.3 summarizes the most recent analyses for G-FP, and highlights limitations of those analyses. In Section 4.4 we propose a new technique to bound carry-out workload.
A response-time analysis and a discussion of the complexity of our method are given in Section 4.5. Section 4.6 presents the evaluation of our method for randomly generated DAG tasks. Lastly, we summarize the contribution presented in this chapter, in Section 4.7.

4.2 Background

In this section we discuss the concept of critical interference that our work is based on, and present a general framework to bound response-times of DAG tasks scheduled under G-FP. In the next section, we summarize the state-of-the-art analyses for G-FP and give an overview of our method.

4.2.1 Assumptions and Notations

We assume the DAG task model as presented in Section 2.4. In addition, the following notation is used. A sequence of subtasks \( (v_{i,u_1}, v_{i,u_2}, \ldots, v_{i,u_t}) \) of \( \tau_i \), in which \( (v_{i,u_j}, v_{i,u_{j+1}}) \in E_i, \forall 1 \leq j \leq t-1 \), is called a chain of \( \tau_i \) and is denoted by \( \lambda_i \). The length of a chain \( \lambda_i \) is the sum of the WCETs of subtasks in \( \lambda_i \) and is denoted by \( \text{len}(\lambda_i) \), i.e., \( \text{len}(\lambda_i) = \sum_{v_{i,u_j} \in \lambda_i} C_{i,u_j} \).

A chain of \( \tau_i \) which has the longest length is a critical path of the task. In this chapter, we assume that tasks are scheduled using a preemptive, global fixed-priority algorithm where each task is assigned a fixed task-level priority. All subtasks of a task have the same priority as the task. Without loss of generality, we assume that tasks have distinct priorities, and \( \tau_i \) has higher priority than \( \tau_k \) if \( i < k \). Figure 4.1 shows a task example that we use to demonstrate the analysis throughout this chapter.
4.2.2 Critical Chain and Critical Interference

The notions of critical chain and critical interference were introduced by Chwa et al. [36, 37] for analyzing parallel tasks scheduled with G-EDF. Unlike sequential tasks, analysis of DAG tasks with internal parallelism is inherently more complicated: (i) some subtasks of a task can be interfered with by other subtasks of the same task (i.e., intra-task interference); (ii) subtasks of a task can be interfered with by subtasks of higher-priority tasks (i.e., inter-task interference); and (iii) the parallelism of a DAG task may vary during execution, subject to the precedence constraints imposed by its graph. The critical chain and critical interference concepts alleviate the complexity of the analysis by focusing on a special chain of subtasks of a task which accounts for its response time, thus bringing the problem closer to a more familiar analysis technique for sequential tasks. Although they were originally proposed for analysis of G-EDF [36, 37], these concepts are also useful for analyzing G-FP. We therefore use them in our analysis and include a discussion of them in this section.

Consider any job $J_k$ of a task $\tau_k$ and its corresponding schedule. A last-completing subtask of $J_k$ is a subtask that completes last among all subtasks in the schedule of $J_k$. A last-completing predecessor of a subtask $v_{k,a}$ is a predecessor that completes last among all predecessors of
Note that a subtask can only be ready after a last-completing predecessor finishes, since only then are all the precedence constraints for the subtask satisfied. Starting from a last-completing subtask of $J_k$, we can recursively trace back through all last-completing predecessors until we reach a subtask with no predecessors. If during that process, a subtask has more than one last-completing predecessor, we arbitrarily pick one. The chain that is reconstructed by appending those last-completing predecessors and the last-completing subtask is called a critical chain of job $J_k$. We call the subtasks that belong to a critical chain critical subtasks.

![Critical chain and critical interference of $J_k$.](image)

**Example:** Figure 4.2 presents an example of a critical chain of a job $J_k$ of task $\tau_k$, which has the same DAG as shown in Figure 4.1. In Figure 4.2, boxes with bold, solid borders denote the execution of critical subtasks of $J_k$; boxes with bold, dashed borders denote the execution of the other subtasks of $J_k$. The other boxes are for jobs of other tasks. Subtask $v_{k,a}$ in the schedule of $J_k$. Note that a subtask can only be ready after a last-completing predecessor finishes, since only then are all the precedence constraints for the subtask satisfied. Starting from a last-completing subtask of $J_k$, we can recursively trace back through all last-completing predecessors until we reach a subtask with no predecessors. If during that process, a subtask has more than one last-completing predecessor, we arbitrarily pick one. The chain that is reconstructed by appending those last-completing predecessors and the last-completing subtask is called a critical chain of job $J_k$. We call the subtasks that belong to a critical chain critical subtasks.

![Critical chain and critical interference of $J_k$.](image)

**Example:** Figure 4.2 presents an example of a critical chain of a job $J_k$ of task $\tau_k$, which has the same DAG as shown in Figure 4.1. In Figure 4.2, boxes with bold, solid borders denote the execution of critical subtasks of $J_k$; boxes with bold, dashed borders denote the execution of the other subtasks of $J_k$. The other boxes are for jobs of other tasks. Subtask $v_{k,a}$ in the schedule of $J_k$. Note that a subtask can only be ready after a last-completing predecessor finishes, since only then are all the precedence constraints for the subtask satisfied. Starting from a last-completing subtask of $J_k$, we can recursively trace back through all last-completing predecessors until we reach a subtask with no predecessors. If during that process, a subtask has more than one last-completing predecessor, we arbitrarily pick one. The chain that is reconstructed by appending those last-completing predecessors and the last-completing subtask is called a critical chain of job $J_k$. We call the subtasks that belong to a critical chain critical subtasks.
$v_{k,6}$ is a last-completing subtask. A last-completing predecessor of $v_{k,6}$ is $v_{k,5}$. Similarly, a last-completing predecessor of $v_{k,5}$ is $v_{k,3}$, and a last-completing predecessor of $v_{k,3}$ is $v_{k,1}$. Hence a critical chain of $J_k$ is $(v_{k,1}, v_{k,3}, v_{k,5}, v_{k,6})$.

The critical chain concept has a few properties that make it useful for schedulability analysis of parallel DAG tasks. First, the first subtask of any critical chain of a job is ready to execute as soon as the job is released, since it does not have any predecessor. Second, when the last subtask of a critical chain completes, the corresponding job finishes — this is from the construction of the critical chain. Thus the scheduling window of a critical chain of $J_k$ — i.e., from the release time of its first subtask to the completion time of its last subtask — is also the scheduling window of job $J_k$ — i.e., from the job’s release time to its completion time. Third, consider a critical chain $\lambda_k$ of $J_k$: at any instant during the scheduling window of $J_k$, either a critical subtask of $\lambda_k$ is executed or a critical subtask of $\lambda_k$ is ready but not executed because all $m$ processors are busy executing subtasks not belonging to $\lambda_k$, including non-critical subtasks of job $J_k$ and subtasks from other tasks (see Figure 4.2). Therefore, the response-time of a critical chain of $J_k$ is also the response-time of $J_k$. Hence if we can upper-bound the response-time of a critical chain for any job $J_k$ of $\tau_k$, that bound also serves as an upper-bound for the response-time of $\tau_k$.

The third property of the critical chain suggests that we can partition the scheduling window of a job $J_k$ into two sets of intervals. One includes all intervals during which critical subtasks of $J_k$ are executed and the other includes all intervals during which a critical subtask of $J_k$ is ready but not executed. The total length of the intervals in the second set is called the critical interference of $J_k$. We include definitions for critical interference and interference caused by an individual task on $\tau_k$ as follows.

**Definition 1.** Critical interference $I_k(a, b)$ on a job $J_k$ of task $\tau_k$ is the aggregated length of all intervals in $(a, b)$ during which a critical subtask of $J_k$ is ready but not executed.
Definition 2. Critical interference $I_{i,k}(a,b)$ on a job $J_k$ of task $\tau_k$ due to task $\tau_i$ is the aggregated processor time from all intervals in $[a, b)$ during which one or more subtasks of $\tau_i$ are executed and a critical subtask of $J_k$ is ready but not executed.

In Figure 4.2, the critical interference $I_k(0,14)$ of $J_k$ is the sum of the lengths of intervals $[0,2)$, $[4,5)$, $[7,9)$, and $[11,13)$ which is 7. The critical interference $I_{i,k}(0,14)$ caused by a task $\tau_i$ is the total processor time of $\tau_i$ in those four intervals. Note that $\tau_i$ may execute simultaneously on multiple processors, and we must sum its processor time on all processors. From the definition of critical interference, we have:

$$I_k(a,b) = \frac{1}{m} \sum_{\tau_i \in \tau} I_{i,k}(a,b). \quad (4.1)$$

4.2.3 A General Method for Bounding Response-Time

![Diagram](image)

Figure 4.3: Workload generated by an interfering task $\tau_i$ in an interval of length $\Delta$.

We now discuss a general framework for bounding response-time in G-FP that is used in this work and was also employed by the state-of-the-art analyses [51, 81]. Based on the definitions
of critical chain and critical interference, the response-time \( R_k \) of \( J_k \) is:

\[
R_k = \text{len}(\lambda_k) + I_k(r_k, r_k + R_k),
\]

where \( \lambda_k \) is a critical chain of \( J_k \) and \( \text{len}(\lambda_k) \) is its length (see Figure 4.2 for example). Applying Equation 4.1 we have:

\[
R_k = \left( \text{len}(\lambda_k) + \frac{1}{m} I_{k,k}(r_k, r_k + R_k) \right) + \frac{1}{m} \sum_{\tau_i \in h\!p(\tau_k)} I_{i,k}(r_k, r_k + R_k), \quad (4.2)
\]

where \( h\!p(\tau_k) \) is the set of tasks with higher priorities than \( \tau_k \)'s. Thus if we can bound the right-hand side of Equation 4.2, we can bound the response-time of \( \tau_k \). To do so, we bound the contributions to \( J_k \)'s response-time caused by subtasks of \( J_k \) itself and by jobs of higher-priority tasks separately.

**Intra-Task Interference**

The sum \( \left( \text{len}(\lambda_k) + \frac{1}{m} I_{k,k}(r_k, r_k + R_k) \right) \), which includes the intra-task interference on the critical chain of \( J_k \) caused by non-critical subtasks of \( J_k \), is bounded by Lemma V.3 in [81]. We include the bound below.

**Lemma 5.** The following inequality holds for any task \( \tau_k \) scheduled by any work-conserving algorithm:

\[
\text{len}(\lambda_k) + \frac{1}{m} I_{k,k}(r_k, r_k + R_k) \leq L_k + \frac{1}{m}(C_k - L_k)
\]
Inter-Task Interference

Now we need to bound the inter-task interference on the right-hand side of Equation 4.2. Since the interference caused by a task in an interval is at most the workload generated by the task during that interval, we can bound \( I_{i,k}(a,b) \), \( \forall \tau_i \in hp(\tau_k) \) using the bound for the workload generated by \( \tau_i \) in the interval \([a,b]\). Let \( W_i(a,b) \) denote the maximum workload generated by \( \tau_i \) in the interval \([a,b]\). Let \( W_i(\Delta) \) denote the maximum workload generated by \( \tau_i \) in any interval of length \( \Delta \). The following inequality holds for any \( \tau_i \):

\[
I_{i,k}(r_k, r_k + R_k) \leq W_i(r_k, r_k + R_k) \leq W_i(R_k).
\] (4.3)

Let the problem window be the interval of interest with length \( \Delta \). The jobs of \( \tau_i \) that may generate workload within the problem window are classified into three types: (i) a carry-in job is released strictly before the problem window and has a deadline within it, (ii) a carry-out job is released within the problem window and has its deadline strictly after it, and (iii) a body job has both release time and deadline within the problem window. Similar to analyses for sequential tasks (e.g., Bertogna et al. [17]), the maximum workload generated by \( \tau_i \) in the problem window can be attained with a release pattern in which (i) jobs of \( \tau_i \) are released as quickly as possible, meaning that the gap between any two consecutive releases is exactly the period \( T_i \), (ii) the carry-in job finishes as late as its worst-case finishing time, and (iii) the body jobs and the carry-out job start executing as soon as they are released. Figure 4.3 shows an example of such a job-release pattern of an interfering task \( \tau_i \) with the DAG structure shown in Figure 4.1.

However, unlike sequential tasks, analysis for parallel DAG tasks is more challenging in two aspects. First, it is not obvious which schedule for the subtasks of the carry-in (carry-out) job would generate maximum carry-in (carry-out) workload. This is because the parallelism of a
DAG task can vary depending on its internal graph structure. Second, for the same reason, aligning the problem window’s start time with the start time of the carry-in job of \( \tau_i \) may not correspond to the maximum workload generated by \( \tau_i \). For instance, in Figure 4.3 if we shift the problem window to the right 2 time units, the carry-in job’s workload loses 2 time units but the carry-out job’s workload gains 5 time units. The total workload thus increases 3 time units. Therefore in order to compute the maximum workload generated by \( \tau_i \) we must slide the problem window to find a position that corresponds to the maximum sum of the carry-in workload and carry-out workload. We discuss an existing method for computing carry-in workload in Section 4.3 and our technique for computing carry-out workload in Section 4.4. In Section 4.5, we combine those two bounds in a response-time analysis and explain how we slide problem windows to compute maximum workloads.

We note that the maximum workload generated by each body job does not depend on the schedule of its subtasks and is simply its total work. Furthermore, regardless of the position of the problem window, the workload contributed by the body jobs, denoted by \( W_i^{BO}(\Delta) \), is bounded as follows.

**Lemma 6.** The workload generated by the body jobs of task \( \tau_i \) in a problem window with length \( \Delta \) is upper-bounded by

\[
W_i^{BO}(\Delta) = \max \left\{ \left( \left\lfloor \frac{\Delta - L_i + R_i}{T_i} \right\rfloor - 1 \right) C_i, 0 \right\}.
\]

**Proof.** Consider the case where the start of the problem window is aligned with the starting time of the carry-in job, as shown in Figure 4.3. The number of body jobs is at most \( \max \left\{ \left\lfloor \frac{\Delta - L_i + R_i}{T_i} \right\rfloor - 1, 0 \right\} \). Thus for this case the workload of the body jobs is at most \( \max \left\{ \left( \left\lfloor \frac{\Delta - L_i + R_i}{T_i} \right\rfloor - 1 \right) C_i, 0 \right\} \).
Shifting the problem window to the left or right can change the workload contributed by the carry-in and carry-out jobs but does not increase the maximum number of body jobs or their workload. The bound thus follows.

Let the carry-in window and carry-out window be the intervals within the problem window during which the carry-in job and the carry-out job are executed, respectively. Intuitively, the carry-in window spans from the start of the problem window to the completion time of the carry-in job; the carry-out window spans from the starting time of the carry-out job to the end of the problem window. We denote the lengths of the carry-in window and carry-out window for task \( \tau_i \) by \( \Delta_i^{CI} \) and \( \Delta_i^{CO} \) respectively. The sum of \( \Delta_i^{CI} \) and \( \Delta_i^{CO} \) is:

\[
\Delta_i^{CI} + \Delta_i^{CO} = L_i + (\Delta - L_i + R_i) \mod T_i \quad (4.4)
\]

Let \( W_i^{CI}(\Delta_i^{CI}) \) be the maximum carry-in workload of \( \tau_i \) for a carry-in window of length \( \Delta_i^{CI} \). Similarly, let \( W_i^{CO}(\Delta_i^{CO}) \) be the maximum carry-out workload of \( \tau_i \) for a carry-out window of length \( \Delta_i^{CO} \). The maximum workload generated by \( \tau_i \) in any problem window of length \( \Delta \) can be computed by taking the maximum over all \( \Delta_i^{CI} \) and \( \Delta_i^{CO} \) that satisfy Equation 4.4:

\[
W_i(\Delta) = W_i^{BO}(\Delta) + \max_{\Delta_i^{CI}, \Delta_i^{CO} \text{satify Eq. 4.4}} \left\{ W_i^{CI}(\Delta_i^{CI}) + W_i^{CO}(\Delta_i^{CO}) \right\} \quad (4.5)
\]

Therefore if we can bound \( W_i^{CI}(\Delta_i^{CI}) \) and \( W_i^{CO}(\Delta_i^{CO}) \), we can bound the inter-task interference of \( \tau_i \) on \( \tau_k \) and thus the response-time of \( \tau_k \).

### 4.3 The State-of-the-Art Analysis for G-FP

Melani et al. [81] proposed a response-time analysis for G-FP scheduling of conditional DAG tasks that may contain conditional vertices, for modeling conditional constructs such as
if-then-else statements. They bounded the interfering workload by assuming that jobs of the interfering task execute perfectly in parallel on all \( m \) processors. Their bound for the interfering workload is computed as follows.

\[
W_i(\Delta) = \left\lfloor \frac{\Delta + R_i - C_i/m}{T_i} \right\rfloor C_i + \min \left\{ C_i, m((\Delta + R_i - C_i/m) \mod T_i) \right\}.
\]

Figure 4.4 illustrates the workload computation for an interfering task \( \tau_i \) given in [81]. As shown in this figure, both carry-in and carry-out jobs are assumed to execute with perfect parallelism upon \( m \) processors. Thus their workload contributions in the considered window are maximized. This assumption simplifies the workload computation as it ignores the internal DAG structures of the interfering tasks. However, assuming that DAG tasks have such abundant parallelism is likely unrealistic and thus makes the analysis pessimistic.

Fonseca et al. [51] later considered a task model similar to the one assumed in this chapter and proposed a method to improve the bounds for carry-in and carry-out workloads by explicitly considering the DAGs. The carry-in workload was bounded using a hypothetical schedule for the carry-in job, in which the carry-in job can use as many processors as it needs to fully exploit its parallelism. They proved that the carry-in workload of the hypothetical schedule is maximized when: (i) the hypothetical schedule’s completion time is aligned with
the worst-case completion time of the interfering task, (ii) every subtask in the hypothetical schedule starts executing as soon as all of its predecessors finish, and (iii) every subtask in the hypothetical schedule executes for its full WCET. Figure 4.3 shows the hypothetical schedule of the carry-in job for the task in Figure 4.1. We adopt their method for computing carry-in workload. In particular, the carry-in workload of task $\tau_i$ with a carry-in window of length $\Delta_i^{CI}$, i.e., from the start of the problem window to the completion time of the carry-in job (see Figure 4.3), is computed as follows.

$$W_{i}^{CI}(\Delta_i^{CI}) = \sum_{v_{i,k} \in V_i} \max \{C_{i,k} - \max(L_i - S_{i,k} - \Delta_i^{CI}, 0), 0\}.$$  (4.6)

In Equation 4.6, $S_{i,k}$ is the start time of subtask $v_{i,k}$ in the hypothetical schedule for the carry-in job described above. It can be computed by taking a longest path among all paths from source subtasks to $v_{i,k}$ and adding up the WCETs of the subtasks along that path excluding $v_{i,k}$ itself.

For the carry-out workload, [51] considered a subset of generalized DAG tasks, namely nested fork-join DAG (NFJ-DAG) tasks. A NFJ-DAG is constructed recursively from smaller NFJ-DAGs using two operations: series composition and parallel composition. Figure 4.5b shows an example NFJ-DAG task. Figure 4.5a shows a similar DAG with one more edge $(v_{i,7}, v_{i,8})$. The DAG in Figure 4.5a is not a NFJ-DAG due to a single cross edge $(v_{i,7}, v_{i,8})$. To deal with a non NFJ-DAG, [51] first transforms the original DAG to a NFJ-DAG by removing the conflicting edges, such as $(v_{i,7}, v_{i,8})$ in Figure 4.5. Then they compute the upper-bound for the carry-out workload using the obtained NFJ-DAG. The computed bound is proved to be an upper-bound for the carry-out workload. We note that the transformation removes some precedence constraints from the original DAG, and thus the resulting NFJ-DAG may have higher parallelism than the original DAG. Hence, computing the carry-out workload of
Figure 4.5: Example for a general DAG task and a nested fork-join DAG task.

(a) A non-nested-fork-join DAG task.
(b) A nested fork-join DAG task.

ea generalized DAG task via its transformed NFG-DAG may be pessimistic, especially for a complex DAG, as the transformation may remove many edges from the original DAG.

In this chapter, we propose a new technique to directly compute an upper-bound for the carry-out workload of generalized DAG task. The high level idea is to frame the problem of finding the bound as an optimization problem, which can be solved effectively by solvers such as the CPLEX [63], Gurobi [60], or SCIP [91]. The solution of the optimization problem then serves as a safe and tight upper-bound for the carry-out workload. In the next section we present our method in detail.
4.4 Bound for Carry-Out Workload

In this section we propose a method to bound the carry-out workload that can be generated by a job of task $\tau_i$ by constructing an integer linear program (ILP) for which the optimal solution value is an upper-bound of the carry-out workload.

Consider a carry-out job of task $\tau_i$, which is scheduled with an unrestricted number of processors, meaning that it can use as many processors as it requires to fully exploit its parallelism. Each subtask of the carry-out job executes as soon as it is ready, i.e., immediately after all of its predecessors have finished. We label such a schedule for the carry-out job $\text{SCH}\mathcal{E}^{CO}(\tau_i)$. We prove in the following lemma that the workload generated by $\text{SCH}\mathcal{E}^{CO}(\tau_i)$ is an upper-bound for the carry-out workload.

**Lemma 7.** For specific values of the execution times for the subtasks of $\tau_i$, workload generated by $\text{SCH}\mathcal{E}^{CO}(\tau_i)$ in a carry-out window of length $\Delta^{CO}_i$ is an upper-bound for the carry-out workload generated by $\tau_i$ with the given subtasks’s execution times.

Proof. We prove by contradiction. Consider a schedule $\text{SCH}\mathcal{E}^*$ for the carry-out job in which subtasks execute for the same lengths as in $\text{SCH}\mathcal{E}^{CO}(\tau_i)$. Suppose subtask $v_{i,k}$ is the first subtask in time order that produces more workload in $\text{SCH}\mathcal{E}^*$ than it does in $\text{SCH}\mathcal{E}^{CO}(\tau_i)$. This means $v_{i,k}$ must have started executing earlier in $\text{SCH}\mathcal{E}^*$ than it have in $\text{SCH}\mathcal{E}^{CO}(\tau_i)$.

Hence, $v_{i,k}$ must have started its execution before all of its predecessors have finished in $\text{SCH}\mathcal{E}^*$. This is impossible and the lemma follows. 

Unlike the carry-in workload, the carry-out workload generated when all subtasks execute for their full WCETs is not guaranteed to be the maximum. Consider an interfering task $\tau_i$ shown in Figure 4.1 and a carry-out window of length 3 time units. If all subtasks of the
carry-out job of $\tau_i$ execute for their WCETs, the carry-out workload would be 4 time units, as shown in Figure 4.6a. However, if subtask $v_{i,1}$ finishes immediately, i.e., executes for 0 time units, the carry-out workload would be 7 time units, as shown in Figure 4.6b. From Lemma 7 and the discussion above, to compute an upper-bound for carry-out workload we must consider all possible execution times of the subtasks, and subtasks must execute as soon as they are ready.

For each subtask $v_{i,a}$ of the carry-out job of an interfering task $\tau_i$, we define two non-negative integer variables $X_{i,a} \geq 0$ and $W_{i,a} \geq 0$. $X_{i,a}$ represents the actual execution time of subtask $v_{i,a}$ in the carry-out job and $W_{i,a}$ denotes the contribution of subtask $v_{i,a}$ to the carry-out workload. Let $\Delta^{CO}$ be an integer constant denoting the length of the carry-out window. Then the carry-out workload is the sum of the contributions of all subtasks in $SCH^{CO}(\tau_i)$, which is upper-bounded by the maximum of the following optimization objective function:

$$\text{obj}(\tau_i, \Delta^{CO}) \triangleq \sum_{v_{i,a} \in V_i} W_{i,a}. \quad (4.7)$$
The optimal value for the above objective function gives the actual maximum workload generated by the carry-out job with unrestricted number of processors. We now construct a set of constraints on the contribution of each subtask in $\text{SCH}^{CO}(\tau_i)$ to the carry-out workload. From the definitions of $X_{i,a}$ and $W_{i,a}$, we have the following bounds for them.

**Constraint 1.** For any interfering task $\tau_i$:

$$\forall v_{i,a} \in V_i : 0 \leq X_{i,a} \leq C_{i,a}.$$  

**Constraint 2.** For any interfering task $\tau_i$:

$$\forall v_{i,a} \in V_i : 0 \leq W_{i,a} \leq X_{i,a}.$$  

These two constraints come from the fact that the actual execution time of subtask $v_{i,a}$ cannot exceed its WCET, and each subtask can contribute at most its whole execution time to the carry-out workload. Let $S_{i,a}$ be the starting time of $v_{i,a}$ in $\text{SCH}^{CO}(\tau_i)$ assuming that the carry-out job starts at time instant $0$. For simplicity of exposition, we assume that the DAG $G_i$ has exactly one source vertex and one sink vertex. If this is not the case, we can always add a couple of dummy vertices, $v_{i,\text{source}}$ and $v_{i,\text{sink}}$, with zero WCETs for source and sink vertices, respectively. Then we add edges from $v_{i,\text{source}}$ to all vertices with no predecessors in the original DAG $G_i$, and edges from all vertices with no successors in $G_i$ to $v_{i,\text{sink}}$. Without loss of generality, we assume that $v_{i,1}$ and $v_{i,n_i}$ are the source vertex and sink vertex of $G_i$, respectively. Let $\sigma_{i,a}^p$ denote a path from the source $v_{i,1}$ to $v_{i,a}$: $\sigma_{i,a}^p \triangleq (v_{i,j_1}, \ldots, v_{i,j_p})$, where $j_1 = 1$, $j_p = a$, and $(v_{i,j_x}, v_{i,j_{x+1}})$ is an edge in $G_i$ for $1 \leq x < p$. Let $\mathcal{P}(v_{i,a})$ denote the set of all paths from $v_{i,1}$ to $v_{i,a}$ in $G_i$: $\mathcal{P}(v_{i,a}) \triangleq \{ \sigma_{i,a}^p \}$. $\mathcal{P}(v_{i,a})$ for all subtasks can be constructed by a graph traversal algorithm. For instance, a simple modification of depth-first search would accomplish this.
For a particular path $\sigma_{i,a}^p$, the sum of execution times of all subtasks in this path, excluding $v_{i,a}$, is called the *distance* to $v_{i,a}$ with respect to this path. We let $D_{i,a}^p$ be a variable denoting the distance to $v_{i,a}$ in path $\sigma_{i,a}^p$. We impose the following two straightforward constraints on $D_{i,a}^p$ based on its definition.

**Constraint 3.** For any interfering task $\tau_i$:

$$\forall v_{i,a} \in V, \forall \sigma_{i,a}^p \in \mathcal{P}(v_{i,a}) : D_{i,a}^p \leq \sum_{v_{i,j,x} \in \{\sigma_{i,a}^p \setminus v_{i,a}\}} X_{i,j,x}.$$ 

**Constraint 4.** For any interfering task $\tau_i$:

$$\forall v_{i,a} \in V, \forall \sigma_{i,a}^p \in \mathcal{P}(v_{i,a}) : D_{i,a}^p \geq \sum_{v_{i,j,x} \in \{\sigma_{i,a}^p \setminus v_{i,a}\}} X_{i,j,x}.$$ 

In the schedule $\text{SCH}E^{CO}(\tau_i)$, the starting time $S_{i,a}$ of a subtask $v_{i,a}$ cannot be smaller than the distance to $v_{i,a}$ in any path $\sigma_{i,a}^p$. We prove this as follows.

**Lemma 8.** In the schedule $\text{SCH}E^{CO}(\tau_i)$ of any interfering task $\tau_i$:

$$\forall v_{i,a} \in V, \forall \sigma_{i,a}^p \in \mathcal{P}(v_{i,a}) : S_{i,a} \geq D_{i,a}^p.$$ 

**Proof.** We prove by contradiction. Let $\sigma_{i,a}^{p*}$ be a path so that the starting time $S_{i,a}$ is smaller than $D_{i,a}^p$. Subtask $v_{i,a}$ must be ready to start execution, meaning all of its predecessors must finish, at time $S_{i,a}$. Since $S_{i,a} < D_{i,a}^p$, there must be a subtask $v_{i,j,x} \in \{\sigma_{i,a}^{p*} \setminus v_{i,a}\}$ executing (and thus not finished) at time $S_{i,a}$. Then $v_{i,a}$ cannot be ready at time $S_{i,a}$ since it depends on $v_{i,j,x}$. This contradicts the assumption that $v_{i,a}$ is ready at $S_{i,a}$ and the lemma follows.

In fact, in the schedule $\text{SCH}E^{CO}(\tau_i)$ the starting time $S_{i,a}$ of $v_{i,a}$ is equal to the longest distance among all paths to it.
Lemma 9. In the schedule $\text{SCHEC}^{CO}(\tau_i)$ of any interfering task $\tau_i$:

$$\forall v_{i,a} \in V_i : S_{i,a} = \max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p.$$ 

Proof. Consider a path $\sigma_{i,a}^{p^*}$ constructed as follows. First we take a last-completing predecessor of $v_{i,a}$, say $v_{i,jx}$. Since $v_{i,a}$ executes as soon as it is ready, it executes immediately after $v_{i,jx}$ finishes. We recursively trace back through the last-completing predecessors in that way until we reach the source vertex $v_{i,1}$. Path $\sigma_{i,a}^{p^*}$ is then constructed by chaining the last-completing predecessors together with $v_{i,a}$. We note that any subtask $v_{i,jx}$ in $\sigma_{i,a}^{p^*}$ executes as soon as its immediately preceding subtask finishes, since no other predecessors of $v_{i,jx}$ finish later than it does. Therefore, $S_{i,a} = D_{i,a}^{p^*}$. From Lemma 8, $\sigma_{i,a}^{p^*}$ must have the longest distance to $v_{i,a}$ among all paths in $\mathcal{P}(v_{i,a})$. Thus the lemma follows.

Based on Lemmas 8 and 9, we have the following constraint for the starting time of $v_{i,a}$.

Constraint 5. For any interfering task $\tau_i$:

$$\forall v_{i,a} \in V_i, \forall \sigma_{i,a}^p \in \mathcal{P}(v_{i,a}) : S_{i,a} \geq D_{i,a}^p.$$

Proof. We prove that this constraint requires that $S_{i,a}$ of every subtask $v_{i,a}$ for which $\max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p < \Delta^{CO}$ satisfies Lemma 9, that is $S_{i,a} = \max_{\sigma_{i,a}^p \in \mathcal{P}(v_{i,a})} D_{i,a}^p$. (Recall that $\Delta^{CO}$ is a constant denoting the carry-out window’s length.) In other words, we prove that it requires that every subtask $v_{i,a}$, which would start executing within the carry-out window in an unrestricted-processor schedule $\text{SCHEC}^{CO}(\tau_i)$, gets exactly the same starting time from the solution to the optimization problem. Let $Q_i$ denote the collection of such subtasks — the ones that would start executing within the carry-out window in $\text{SCHEC}^{CO}(\tau_i)$. 

80
Let $\pi^*$ be the solution to the optimization problem and $S^*_{i,a}$ be the corresponding value for the starting time of any subtask $v_{i,a} \in Q_i$ in the solution $\pi^*$. Obviously $S^*_{i,a} \geq \max_{\sigma_{i,a} \in P(v_{i,a})} D^p_{i,a}$ for any $v_{i,a}$ since any solution to the optimization problem satisfies this constraint. If $S^*_{i,a} = \max_{\sigma_{i,a} \in P(v_{i,a})} D^p_{i,a}$ for any $v_{i,a} \in Q_i$, then we are done. Suppose instead that $S^*_{i,a} = \max_{\sigma_{i,a} \in P(v_{i,a})} D^p_{i,a} + \epsilon_{i,a}, \epsilon_{i,a} > 0$ for some $v_{i,a} \in Q_i$. Let $Q'_i$ denote the set of such subtasks. We construct a solution $\pi'$ to the optimization problem from $\pi^*$ as follows. Consider a first subtask $v_{i,a} \in Q'_i$ in time. We reduce its starting time by $\epsilon_{i,a}$: $S'_{i,a} = S^*_{i,a} - \epsilon_{i,a}$. Since $v_{i,a}$ is the first delayed subtask, doing this does not violate the precedence constraints for other subtasks. We iteratively perform that operation for other subtasks in $Q'_i$ in increasing time order. The solution $\pi'$ constructed in this way yields a larger carry-out workload since more workload from individual subtasks can fit in the carry-out window. Therefore $\pi'$ is a better solution, which contradicts the assumption that $\pi^*$ is an optimal solution.

The workload contributed by a subtask $v_{i,a}$ is:

$W_{i,a} = \min\left\{ \max\{\Delta_{CO} - S_{i,a}, 0\}, X_{i,a} \right\}$. The second part of the outer minimization has been taken care of by Constraint 2. We now construct constraints to impose the first part of the minimization. Let $M_{i,a}$ be an integer variable representing the expression $\max\{\Delta_{CO} - S_{i,a}, 0\}$. Let $A_{i,a}$ be a binary variable which takes value either 0 or 1. We have the following constraints.

**Constraint 6.** For any interfering task $\tau_i$:

$$\forall v_{i,a} \in V_i : W_{i,a} \leq M_{i,a}.$$ 

**Constraint 7.** For any interfering task $\tau_i$:

$$\forall v_{i,a} \in V_i : M_{i,a} \geq 0.$$
Constraint 8. For any interfering task $\tau_i$:

$$\forall v_{i,a} \in V_i : M_{i,a} \leq (\Delta^{CO} - S_{i,a})A_{i,a}.$$  

Constraints 7 and 8 bound the value for $M_{i,a}$ and Constraint 6 enforces another upper bound for the workload $W_{i,a}$. If $\Delta^{CO} < S_{i,a}$, $A_{i,a}$ can only be 0 in order to satisfy both Contraints 7 and 8. If $\Delta^{CO} = S_{i,a}$, the value of $A_{i,a}$ does not matter. In both cases, these three constraints together with Constraint 2 bound $W_{i,a}$ to zero contribution of $v_{i,a}$ to the carry-out workload. If $\Delta^{CO} > S_{i,a}$, the maximizing process enforces that $A_{i,a}$ takes value 1. Therefore in any case Constraints 2, 6, 7, and 8 enforce a correct value for the workload contribution $W_{i,a}$ of $v_{i,a}$.

We have constructed an ILP with a quadratic constraint (Constraint 8) for each $v_{i,a}$, for which the optimal solution value is an upper bound for the carry-out workload. The carry-out workload of $\tau_i$ in a carry-out window of length $\Delta^{CO}$ can also be upper-bounded by the following straightforward lemma.

**Lemma 10.** The carry-out workload of an interfering task $\tau_i$ scheduled by G-FP in a carry-out window of length $\Delta^{CO}$ is upper-bounded by $m\Delta^{CO}$.

Lemma 10 follows directly from the fact that the carry-out job can execute at most on all $m$ processors of the system during the carry-out window. Since the carry-out workload of $\tau_i$ is upper-bounded by both the maximum value returned for the optimization problem and Lemma 10, it is upper-bounded by the minimum of the two quantities.

**Theorem 11.** The carry-out workload of an interfering task $\tau_i$ scheduled by G-FP in a carry-out window of length $\Delta^{CO}$ is upper-bounded by: $\min \{ OBJ, m\Delta^{CO} \}$, where $OBJ$ is the maximum value returned for the maximization problem (Equation 4.7).
As discussed in Section 4.3, the technique proposed by Fonseca et al. [51] can be applied directly for NFJ-DAGs but not for general DAGs. For a general DAG, the procedure to transform the general DAG to an NFJ-DAG will likely inflate the carry-out workload bound as it removes some precedence constraints between subtasks and enables a higher parallelism (and thus a greater interfering workload) for the carry-out job. In contrast, our method directly bounds the carry-out workload for any DAG and the optimal value obtained is the actual maximum carry-out workload. Hence, our method theoretically yields better schedulability than [51]'s for general DAGs. The cost of our method is higher time complexity for computing carry-out workload due to the hardness of the ILP problem. However, it can be implemented and works effectively with modern optimization solvers, as we show in our experiments (Section 4.6).

4.5 Response-Time Analysis

From the above calculations for the bounds of intra-task interference and inter-task interference on $\tau_k$, we have the following theorem for the response-time bound of $\tau_k$.

**Theorem 12.** A constrained-deadline task $\tau_k$ scheduled by a global fixed-priority algorithm has response-time upper-bounded by the smallest integer $R_{ub}^k$ that satisfies the following fixed-point iteration:

$$R_{ub}^k \leftarrow L_k + \frac{1}{m}(C_k - L_k) + \frac{1}{m} \sum_{\tau_i \in hp(\tau_k)} W_i(R_{ub}^k).$$

**Proof.** This follows from Equation 4.2, Lemma 5 and the fact that the inter-task interference of $\tau_i$ on $\tau_k$ is bounded by the workload generated by $\tau_i$ (Equation 4.3).
Algorithm 3 Response-Time Analysis for G-FP

1: procedure SchedulabilityTest$(\tau)$ \triangleright Without loss of generality, assuming tasks are sorted in decreasing order of priority
2: \hspace{1em} for Each $\tau_k \in \tau$ do \hspace{1em} \triangleright Initialize the values for response-time bounds
3: \hspace{2em} $R_{ub}^k \leftarrow L_k + \frac{1}{m}(C_k - L_k)$
4: \hspace{2em} if Any $R_{ub}^k > D_k$ then
5: \hspace{3em} Return Unschedulable
6: \hspace{2em} end if
7: \hspace{1em} end for
8: \hspace{1em} for $\tau_k$ from $\tau_2$ to $\tau_n$ do
9: \hspace{2em} Calculate $R_{ub}^k$ in Theorem 12
10: \hspace{2em} if $R_{ub}^k > D_k$ then
11: \hspace{3em} Return Unschedulable
12: \hspace{2em} end if
13: \hspace{1em} end for
14: \hspace{1em} Return Schedulable
15: end procedure

In Theorem 12, $W_i(R_{ub}^k)$ is computed using Equation 4.5 for all carry-in and carry-out windows that satisfy Equation 4.4. For specific carry-in and carry-out window lengths, the carry-in workload is bounded using Equation 4.6 and the carry-out workload is bounded as discussed in Section 4.4. The lengths for carry-in window $\Delta_i^{CI}$ and carry-out window $\Delta_i^{CO}$ are varied as follows. Let $\Gamma$ denote the right-hand side of Equation 4.4. First $\Delta_i^{CI}$ takes its largest value: $\Delta_i^{CI} \leftarrow \min\{\Gamma, L_i\}$, and $\Delta_i^{CO}$ takes the remaining sum: $\Delta_i^{CO} \leftarrow \min\{\Gamma - \Delta_i^{CI}, L_i\}$. Then in each subsequent step, $\Delta_i^{CI}$ is decreased and $\Delta_i^{CO}$ is increased until $\Delta_i^{CO}$ takes its largest value and $\Delta_i^{CI}$ takes the remaining value. We note that if at the first step both $\Delta_i^{CI}$ and $\Delta_i^{CO}$ are greater than or equal to $L_i$, the carry-in workload and carry-out workload are bounded by $\min(C_i, m\Delta_i^{CI})$ and $\min(C_i, m\Delta_i^{CO})$, respectively. Similarly, if the sum of $\Delta_i^{CI}$ and $\Delta_i^{CO}$ is 0 in Equation 4.4, both the carry-in workload and the carry-out workload are 0. We also note that for the highest priority task, there is no interference from any other task, and thus its response-time bound can be computed simply by: $R_{ub}^k \leftarrow (L_k + \frac{1}{m}(C_k - L_k))$. 

84
Using the above response-time bound, we derive a schedulability test, shown in Algorithm 4. First we initialize the response-times for the tasks to be \( (L_k + \frac{C_k - L_k}{m}) \) for all tasks \( \tau_k \). If for any task, the initial response-time is larger than its relative deadline, then the task set is deemed unschedulable (lines 2-7). Otherwise, we repeatedly compute the response-time bound for each task in descending order of priority using the fixed-point iteration in Theorem 12 (line 9). After the computation for each task finishes, we check whether the response-time bound is larger than its deadline. If it is, then the task set is deemed unschedulable (lines 10-12). Otherwise, the task set is deemed schedulable after all tasks have been checked (line 14).

As expected for response-time analysis, for each task \( \tau_i \) the number of iterations in the fixed-point equation (Theorem 12) is pseudo-polynomial in the task’s deadline \( D_i \) (line 9). In each iteration of the fixed-point equation and for each interfering task, we consider all combinations of carry-in and carry-out window lengths that satisfy Equation 4.4 to compute the maximum interfering workload. There are \( O(L_i) \) such combinations, and thus the ILP for the carry-out workload is solved \( O(L_i) \) times. The maximum workload over all combinations of carry-in and carry-out window lengths gives an upper-bound for the interfering workload generated by the given interfering task.

### 4.6 Experimental Evaluation

As we discussed in Sections 4.3 and 4.4, we apply a similar, high-level framework for analyzing schedulability of G-FP scheduling to the one used by Fonseca et al. [51] — i.e., accounting for the interfering workloads caused by the body jobs, the carry-in and carry-out jobs separately, and maximizing the interference by sliding the problem window. However, unlike [51] our technique for bounding carry-out workload works directly for general DAGs and does not introduce pessimism due to the removal of precedence constraints between subtasks, as
Figure 4.7: Ratio of schedulable task sets for varying total utilization and varying number of processors.

(a) Result for $m = 16$, minimum task utilization $\beta = 0.2$, and varying total utilization.

(b) Result for $m = 16$, minimum task utilization $\beta = 0.4$, and varying total utilization.

(c) Result for total utilization $U = 0.5m$, minimum utilization $\beta = 0.1$, and varying $m$.

(d) Result for total utilization $U = 0.7m$, minimum utilization $\beta = 0.1$, and varying $m$.

presented in [51], though for carry-in workload, we reuse the result from [51]. Hence, we consider our work as a generalization/extension of [51] that can be applied for general sporadic DAG tasks. The performance of our method in terms of the schedulability ratio is compatible with [51]'s — it theoretically is at least as good as [51] for NFJ-DAGs and is better than [51] for non NFJ-DAGs. We thus focus on measuring the performance of our method and use the work by Melani et al. [81] as a reference for evaluating the improvement of our method upon their simple one.
We applied the Erdős-Rényi $G(n,p)$ method, described in [38], to generate DAG tasks. In this method the number of subtasks, given by parameter $n$ in $G(n,p)$, is first fixed. Then, directed edges between pairs of vertices are added with probability $p$. Since the obtained DAG may not necessarily be connected, we added a minimum number of edges to make it weakly connected. In our experiments, the probability for a directed edge to be added is $p = 0.2$. We chose the number of subtasks uniformly in the range $[10, 20]$. Other parameters for each DAG task $\tau_i$ were generated similarly to [81]. In particular, the WCETs of subtasks of $\tau_i$ were generated uniformly in the range $[1, 100]$. After that, the work $C_i$ and span $L_i$ were calculated. $\tau_i$’s utilization was generated uniformly in the range $[\beta, C_i/L_i]$, where $\beta \leq 1$ is a parameter to control the minimum task’s utilization and $C_i/L_i$ represents the degree of parallelism of task $\tau_i$. $\tau_i$’s deadline $D_i$ was generated using a normal distribution with mean equal to $(T_i + L_i)/2$ and standard deviation equal to $(T_i - L_i)/4$. We kept generating the relative deadline until a value in the range $[L_i, T_i]$ was obtained.

To generate a task set for a given total utilization, we repeatedly add DAG tasks to the task set until the desired utilization is reached. The utilization (and period) of the last task may need to be adjusted to match the total utilization. We used the SCIP solver [91] with CPLEX [63] as its underlying LP-solver to compute the bound for carry-out workload. For our experiments, we set the default minimum utilization of individual tasks $\beta$ to 0.1. For each configuration we generated 500 task sets and recorded the ratios of task sets that were deemed schedulable. We compare our response-time analysis, denoted by DGA-RTA, with the response-time analysis introduced in [81], denoted by MBB-RTA. For all generated task sets, priorities were assigned in Deadline Monotonic order — studying an efficient priority assignment scheme for G-FP is beyond the scope of this dissertation.

Figures 4.7a, 4.7b, 4.7c, and 4.7d show representative results for our experiments. In Figure 4.7a and 4.7b, we fixed the total number of processors $m = 16$ and varied the total
utilization from 1.0 to 14.0. The minimum task utilization $\beta$ was set to 0.2 and 0.4 in these two experiments, respectively. Unsurprisingly, DGA-RTA dominates MBB-RTA, as was also observed in [51]. Notably, its schedulability ratios for some configurations are two times or more greater than for MBB-RTA, e.g., for total utilizations of 8.0, 9.0 in Figure 4.7a, and 7.0, 8.0 in Figure 4.7b. In Figures 4.7c and 4.7d, we fixed the normalized total utilization and varied the number of processors $m$ from 2 to 36. For each value of $m$, we generated task sets with total utilization $U = 0.5m$ or $U = 0.7m$ for these two experiments, respectively. Similar to the previous experiments, the schedulability ratios of the generated task sets were improved significantly using DGA-RTA compared to MBB-RTA.

To provide a trade-off between computational complexity and accuracy of schedulability test, one can employ our analysis in combination with the analysis presented in [51] by first applying their response-time analysis and then using our analysis if the task set is deemed unschedulable by [51]. In this way, one can get the best result from both analyses.

4.7 Summary

In this chapter we have considered constrained-deadline, parallel DAG tasks scheduled under a preemptive, G-FP scheduling algorithm on multiprocessor platforms. We propose a new technique for bounding carry-out workload of interfering task by converting the calculation of the bound to an optimization problem, for which efficient solvers exist. The proposed technique applies directly to general DAG tasks. The optimal solution value for the optimization problem serves as a safe and tight upper bound for carry-out workload. We present a response-time analysis for G-FP based on the proposed workload bounding technique. Experimental results affirm the dominance of the proposed approach over existing techniques. There are a couple of open questions that we would like to address in future.
They include bounding carry-in and carry-out workloads for the actual number of processors $m$ of the system and designing an efficient priority assignment scheme for parallel DAG tasks scheduled under G-FP algorithm.
Chapter 5

Analysis of Federated Scheduling for Parallel Tasks with Shared Resources

In the previous chapters, we have been assuming that tasks do not share non-processor resources, such as in-memory data structures, network buffers, or I/O devices. In this chapter, non-processor shared resources are considered in an analysis for federated scheduling of parallel tasks. We, herein, refer to those non-processor resources as \textit{shared resources}.

5.1 Introduction

As previously shown in the studies by Brandenburg et al. [26, 31], many real-world applications use locks to synchronize accesses to shared resources. Since tasks may have to wait to get these locks, this form of synchronization requires that the schedulability analyses take these waiting times into account. While researchers have developed many schedulability analyses

\footnote{Contents of this chapter are adapted from the following paper: Dinh, S., Li, J., Agrawal, K., Gill, C., and Lu, C. (2018). Blocking analysis for spin locks in real-time parallel tasks. \textit{IEEE Transactions on Parallel and Distributed Systems}, 29(4), 789-802.}
for sequential tasks with shared resources (for both uniprocessors and multiprocessors) [7, 20, 28, 29, 35, 45, 54, 84, 92], little work has been done for parallel tasks.

Parallel tasks present additional challenges with respect to synchronization. Due to the parallel structure of a task, multiple cores running different parts of the same task can access the same resource concurrently, causing one core to block the others. Thus, even with a single parallel task, adding critical sections may change its schedulability, which is not the case for sequential tasks. Therefore, for parallel tasks, we must consider both inter-task and intra-task contentions for shared resources. For similar reasons, multiple concurrent critical sections of a task can block other tasks at the same time — making the blocking time worse.

Finally, unlike sequential tasks, two critical sections $A$ and $B$ of the same task may execute in different orders: $A$ may appear before, after or concurrently with $B$, because internal scheduling of this task may vary from execution to execution (see further explanation in Section 5.3). This makes the blocking time analysis much harder for parallel tasks.

In this chapter, we provide a schedulability analysis for task sets consisting of parallel tasks that contain critical sections. Our schedulability analysis is designed for federated schedulers [8, 11, 74, 76]. We specifically consider federated schedulers that use the following parameters of each parallel task to calculate and assign the minimum number of dedicated cores to that task: work — its worst-case execution time on a single core, critical-path length — its worst-case execution time on hypothetically infinite number of cores, and relative deadline — a time interval from a job release of the task, within which it must complete.

Due to this dedicated core assignment, federated schedulers provide specific advantages when analyzing schedulability for tasks with critical sections. First, they eliminate priority inversions since tasks do not share cores and therefore do not need prioritization. Second, for
the same reason, if tasks do not access the same lock, they do not interfere with each other for either CPU cycles or the shared resource. Lastly, with global scheduling, up to $m$ (the number of cores on the machine) critical sections of the same task may be concurrent in the case of spin locks; federated scheduling limits such blocking since tasks can only execute on limited numbers of assigned cores.

To incorporate blocking times under federated schedulers, we specifically use spin locks to synchronize accesses to shared resources — that is, in case of contention, each thread will spin until its turn to access the resource. Since parallel tasks are assigned dedicated cores, the blocking time of a task on a core cannot be utilized by other tasks. Moreover, Brandenburg et al., [26, 31] have studied the distribution of critical section lengths in the Linux kernel and several real-time applications and found that most critical sections are short (e.g., 95% of them are shorter than 5µs on their specific machine). Therefore, spin locks would be a better choice for most cases since unlike suspension-based locks, spin locks do not trigger scheduling decisions, preemptions, or migrations which are high overhead OS kernel operations. Spin locks instead can be implemented simply and efficiently in user-space and generally have lower overhead than suspension-based locks.

We derive the schedulability of tasks with critical sections under federated scheduling as follows. We calculate the worst-case blocking times of each task and use them to inflate its work and critical-path length, when calculating the number of cores assigned to this task. Since the blocking times of a task actually depend on the number of cores assigned to each task, we use a fixed-point iteration algorithm to iteratively calculate the blocking times and core assignment.

Note that this schedulability test only requires basic information about the tasks including worst case execution time (work), critical-path length (span), relative deadline, the number
of critical sections for each shared resource, and their lengths; it does not require the internal graph structure of the parallel task. This property confers a few advantages. First, this simplifies the test and makes it tractable. The required information about the parallel tasks also can be collected easily. Second, a common algorithm design technique for parallel programs is divide-and-conquer in which data is divided into multiple portions and processed by multiple threads concurrently. Parallel programs written using this approach are data-dependent and the internal graph structure of such a program can only be unfolded at run time by the runtime system of the parallel language in which the program is written. Thus, the exact graph structure of a parallel task can vary from one release to the next. Finally, federated scheduling is flexible in that it can use any greedy (work conserving) scheduler; therefore, even for the same graph structure the execution can differ from one invocation to the next due to changes in the internal scheduling of the graph. If instead we wished to use information about the structure of the task, we would have to examine all of its possible internal graph structures, and for each graph structure, either we would have to fix the schedule statically, or the analysis would have to consider all possible internal schedules for that structure, which would make the analysis very expensive. Fixing the schedule also makes it difficult to guarantee greediness, which is required to ensure that parallel tasks meet their deadlines under federated scheduling. In addition, as a practical matter, it makes implementation more difficult: e.g., one would not be able to simply use OpenMP’s work conserving scheduler out of the box as our current implementation does.

In this chapter, we focus primarily on systems containing only high-utilization tasks (i.e., heavy tasks) — tasks with utilization greater than 1.0 — since these are the tasks that require parallel execution to meet their deadlines. In particular, these are tasks for which federated schedulers assign dedicated cores. Such systems are not uncommon in emerging real-time applications such as hybrid structural simulation [48] where each task is a numerical
simulation and has high computational demand relative to its deadline. Towards the end of this chapter, we will discuss the challenges involved in task sets with both light and heavy tasks as well as some potential strategies for addressing these challenges.

This chapter is organized as follows. For parallel tasks with critical sections, described in Section 5.2, we consider tasks using two types of spin locks: (i) FIFO-ordered in which requests to a resource are satisfied in first-come-first-served order; and (ii) Priority-ordered in which requests to a resource are satisfied based on their priorities — higher priority requests are guaranteed to acquire the lock before lower priority ones. Note that these priorities are only used to arbitrate the locks — the tasks never share cores and therefore no priority is needed for scheduling tasks on their cores. We describe the priority for requests in Section 5.2. We provide a high-level overview of the schedulability analysis in Section 5.3. We first analyze intra-task blocking — the blocking experienced due to critical sections within the task — in Section 5.4. Then inter-task blocking — the blocking experienced due to critical sections from other tasks — is analyzed in Section 5.5. We then summarize and discuss the results of our analysis in Section 5.6. To compare the performance between FIFO-ordered and priority-ordered locks, we first conduct numerical evaluations and observe that priority-ordered locks generally provide better schedulability (Section 5.7). In addition, we implement both FIFO-ordered and priority-ordered spin locks in a federated scheduling platform that can execute parallel programs written in OpenMP. We run empirical experiments and show results that indicate that the platform with priority-ordered locks also can schedule parallel tasks better than the one with FIFO-ordered locks in practice (Section 5.8). Finally, Section 5.9 summarizes the contributions of this chapter.
5.2 Task and Resource Model

Parallel Task Model: We consider a task model similar to the DAG task model described in Section 2.4. In particular, we consider a task set consisting of $n$ parallel tasks with implicit deadlines, i.e., $D_i = T_i$. The tasks are scheduled on $m$ identical processors. In contrast to the model described in Section 2.4, we only use the parameters of the tasks and ignore their specific DAG structures in our analysis for the reasons explained in Section 5.1.

We use $J_i$ to denote any job of the task $\tau_i$. For each task, a job must finish before the next job is released. Our model is thus different than the pipeline model, where multiple invocations of a task can run simultaneously. In the pipeline parallel model, the task graph is usually well defined, which is also different than our task model. Each job of a task $\tau_i$ can be described as a dynamically unfolding directed acyclic graph (DAG) in which each node (subjob) represents a sequence of instructions (a strand) and each edge represents a precedence constraint between nodes. A node is ready to be executed when all of its predecessors in the graph have been executed. The exact DAG of a task depends on the input data and is unfolded by the runtime system of the parallel programming language in which the task is written, such as OpenMP or Cilk Plus. Thus, the DAG of a task can be different for each release and is only known at run time. Each parallel task $\tau_i$ has two computational parameters:

- The work (or worst-case execution time) $C_i$ of task $\tau_i$ is the sum of execution times of all nodes of task $\tau_i$.

The analysis and results presented in this chapter, however, also hold for constrained-deadline tasks. The reason is that the proposed analysis and processor-allocation procedure are developed using relative deadlines, and not periods. Therefore, as long as job releases of a task satisfy a condition that a job can only be released after its previous job has completed, which is true for constrained-deadline tasks, the analysis in this chapter is not affected.
The **critical-path length** (or span) $L_i$ of task $\tau_i$ is the length of the longest weighted path in the DAG of $\tau_i$, where the weight of each node is its execution time.

These two parameters can be measured using profiling tools for the particular parallel language used to write the parallel tasks.

A job is said to be pending within an interval starting from its arrival until its completion. We use $\text{njobs}(\tau_j, t) = \left\lceil \frac{t + r_j}{T_j} \right\rceil$ [27], where $r_j$ is the response time of $\tau_j$, to denote the maximum number of jobs of $\tau_j$ that can be pending in an interval of length $t$. Table 5.1 summarizes the notation used in this chapter.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau$</td>
<td>Set of tasks in the task set</td>
</tr>
<tr>
<td>$C_i$</td>
<td>Work (i.e., worst-case execution time) of task $\tau_i$</td>
</tr>
<tr>
<td>$L_i$</td>
<td>Critical-path length of task $\tau_i$</td>
</tr>
<tr>
<td>$D_i$</td>
<td>Relative deadline of task $\tau_i$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>Minimum inter-arrival time of task $\tau_i$</td>
</tr>
<tr>
<td>$Q_i$</td>
<td>Set of shared resources accessed by task $\tau_i$</td>
</tr>
<tr>
<td>$R_{i,q}$</td>
<td>Maximum number of requests to $l_q$ of task $\tau_i$</td>
</tr>
<tr>
<td>$\Phi_{i,q}$</td>
<td>Maximum length of requests to $l_q$ by task $\tau_i$</td>
</tr>
<tr>
<td>$n_i$</td>
<td>Number of cores allocated to task $\tau_i$</td>
</tr>
<tr>
<td>$r_i$</td>
<td>Response time of task $\tau_i$</td>
</tr>
<tr>
<td>$B_i^C$</td>
<td>Blocking bound for the work of task $\tau_i$</td>
</tr>
<tr>
<td>$B_i^L$</td>
<td>Blocking bound for the critical-path of task $\tau_i$</td>
</tr>
<tr>
<td>$\tau_i^{LP}$</td>
<td>Set of tasks with lower locking-priority than $\tau_i$</td>
</tr>
<tr>
<td>$\tau_i^{HP}$</td>
<td>Set of tasks with higher locking-priority than $\tau_i$</td>
</tr>
<tr>
<td>$\text{njobs}(\tau_j, t)$</td>
<td>Maximum number of jobs of $\tau_j$ pending in an interval of length $t$</td>
</tr>
<tr>
<td>$dpr(\tau_i, l_q)$</td>
<td>Maximum time a request to $l_q$ by task $\tau_i$ can be delayed</td>
</tr>
</tbody>
</table>

Table 5.1: Notation Summary

**Resource Sharing:** We augment the parallel task model to include access to shared resources. Let $Q_i$ denote a set of resources that $\tau_i$ accesses. The maximum number of times a job of task $\tau_i$ accesses a resource $l_q \in Q_i$ is $R_{i,q}$.  

96
We consider serially reusable and non-preemptible resources that require mutual exclusion to enforce the consistency of their state, such as in-memory data structures, shared data objects, network buffers. Each shared resource is protected by a distinct spin lock. Threads of a task must acquire the lock on a shared resource before accessing the resource. If the lock is already held by another thread, it must spin non-preemptively until its turn. Threads also run non-preemptively during critical sections. Outside of critical sections and spinning durations, threads are preemptible. We overload notation and use $l_q$ to denote both the resource and the spin lock that protects it. We also use critical section and request interchangeably – they both denote a code segment executed on a core that is required to be serialized with other code segments accessing the same resources executed on other cores. The maximum length of a critical section (the amount of time it holds a lock) of a job of $\tau_i$ for resource $l_q$ is denoted by $\Phi_{i,q}$. We assume there are no nested critical sections; that is, a task can only hold one lock at a time.

We consider two orders for spin locks: FIFO-ordered and priority-ordered. In priority-ordered spin locks, requests from a task to a resource have a priority, which we call the locking-priority to distinguish it from conventional task priorities in the real-time systems literature. Requests to a shared resource from different tasks are satisfied in order of their locking-priorities. We assume that requests from a task have the same locking-priority; in other words, the locking-priorities are task-level fixed priorities. Requests for a shared resource issued by the same task are satisfied in FIFO order. Later, in our evaluation (Section 5.7), we investigate different methods to assign locking-priorities for resource requests.
5.3 Schedulability Analysis

In this section, we present a schedulability analysis for federated scheduling with parallel real-time tasks sharing resources that are protected by FIFO-ordered and priority-ordered locks. We first calculate the number of dedicated cores $n_i$ for each task with no blocking delays. We then repeatedly use these assignments to calculate an upper bound on blocking delays experienced by each task and recalculate the new assignments (new sets of $n_i$ for all tasks) based on these blocking delays, until a fixed point is found.

**Work Blocking and Critical-Path Blocking:** The blocking experienced by a parallel task has two components: work blocking and critical-path blocking, defined as follows.

**Definition 3.** Work blocking $B_C^i$ is an upper bound on the total amount of time that all the cores assigned to $\tau_i$ spend spinning (collectively), waiting for lock requests to be granted.

**Definition 4.** Critical-path blocking $B_L^i$ is an upper bound on the amount of spinning time accumulated along any single path of the DAGs of task $\tau_i$, waiting for lock requests on that path to be granted.

Intuitively, each execution of a job $J_i$ of task $\tau_i$ can be mapped to a hypothetical DAG $G'$, which is constructed by augmenting $J_i$’s DAG $G$ with “spinning nodes” — nodes that represent spinning intervals on that specific execution. For all possible executions of jobs of task $\tau_i$, we have a set of such hypothetical DAGs. Then the accumulated spinning time along any single path of any hypothetical DAG in that set is bounded by the critical-path blocking $B_L^i$.

**Calculating the Core Assignments:** Given $B_C^i$ and $B_L^i$ for task $\tau_i$, we calculate the “inflated work” as $C_i + B_C^i$ and the “inflated critical-path length” as $L_i + B_L^i$. 

98
Lemma 13. Given a parallel task $\tau_i$ with $u_i = \frac{C_i}{D_i} > 1$, work blocking $B_i^C$, and critical-path blocking $B_i^L$ such that $D_i > L_i + B_i^L$, $n_i$ dedicated cores are sufficient to guarantee that all jobs of this task will meet their deadlines when scheduled with a greedy scheduler, where

$$n_i = \left\lceil \frac{C_i + B_i^C - L_i - B_i^L}{D_i - L_i - B_i^L} \right\rceil$$  \hspace{1cm} (5.1)

Proof. Consider an arbitrary execution $E$ (i.e., schedule) of $\tau_i$ on its current number of dedicated cores. We construct a hypothetical DAG $G'$ from $E$ by inflating the nodes of the DAG $G$ of $\tau_i$ with the corresponding spinning times. By definition of $B_i^C$ and $B_i^L$, the work $C_i'$ of $G'$ is bounded by $C_i + B_i^C$, and the critical-path length $L_i'$ of $G'$ is bounded by $L_i + B_i^L$. Therefore, applying Lemma 1 gives us the number of cores required for $G'$:

$$\left\lceil \frac{C_i' - L_i'}{D_i - L_i - B_i^L} \right\rceil \leq \left\lceil \frac{C_i + B_i^C - L_i - B_i^L}{D_i - L_i - B_i^L} \right\rceil$$

Thus, the number of cores assigned to $\tau_i$ by Equation 5.1 is sufficient to schedule the DAG $G'$. In other words, it is sufficient to guarantee that all jobs of task $\tau_i$ with work blocking $B_i^C$, and critical-path blocking $B_i^L$ will meet their deadlines. \quad \square

**Schedulability Test:** Algorithm 4 shows the pseudocode for the schedulability test. We start by setting $B_i^C$ and $B_i^L$ to 0 for all tasks and calculate the set of $n_i$. We then repeatedly calculate new values of $B_i^C$ and $B_i^L$ and use Equation 5.1 to calculate the new $n_i$ for each task in alternating fashion. Note that both $B_i^L$ and $B_i^C$ depend on (i) the numbers of requests by all tasks that access the same resources that task $\tau_i$ accesses, and (ii) the numbers of cores allocated to all tasks. Meanwhile, the number of cores $n_i$ allocated to each task $\tau_i$ depends on both $B_i^L$ and $B_i^C$. Therefore, the algorithm runs iteratively until a fixed point is reached.

The algorithm declares a task set unschedulable if either the inflated critical-path length $L_i + B_i^L \geq D_i$ or the total number of cores allocated across all tasks ($\sum_i n_i$) exceeds $m$, the total number of available cores. Otherwise, the task set is declared schedulable once no task’s core allocation changes. If at any step, the new calculated $n_i$ is ever smaller, we keep the
Algorithm 4 Schedulability Test

1: procedure CalculateBlock(\( \tau_i, n_1, ..., n_n \) )
2: Calculate and return \( B^C_i \) and \( B^L_i \)
3: end procedure

4: procedure IsSchedulable(\( \tau \))
5: for each \( \tau_i \in \tau \) do \( n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \)
6: end for
7: while (true) do
8: for each \( \tau_i \in \tau \) do
9: \( B^C_i, B^L_i = \text{CalculateBlock}(\tau_i, n_1, ..., n_n) \)
10: if \( L_i + B^L_i \geq D_i \) then
11: Return Unschedulable
12: end if
13: \( n'_i = \left\lceil \frac{C_i + B^C_i - L_i - B^L_i}{D_i - L_i - B^L_i} \right\rceil \)
14: end for
15: if (\( \sum_{i=1}^{n} n'_i > m \)) then
16: Return Unschedulable
17: end if
18: if (No task’s core assignment changes) then
19: Return Schedulable
20: end if
21: for each \( \tau_i \in \tau \) do
22: if \( n'_i > n_i \) then
23: \( n_i = n'_i \)
24: end if
25: end for
26: end while
27: end procedure
previous value. Because the dependency between the blocking delays \((B^L_i, B^C_i)\) and the core allocation \((n_i)\) is complex, it is hard to assure that the core allocation to each task increases monotonically after each iteration. However, this is safe because the task is assigned at least the number of cores required to satisfy its deadline, given the calculated blocking \(B^C_i\) and \(B^L_i\) at this step. When a fixed point is reached, i.e., the task set is deemed schedulable, the blocking delays and the number of cores allocated to each task do not change anymore. This means that with this core allocation, every task in the task set meets its deadlines, with its blocking delays accounted.

In this algorithm, the crucial step is the calculation of \(B^L_i\) and \(B^C_i\) – the blocking times for each task \(\tau_i\) on the critical-path and work, respectively. We divide the blocking into two portions: **intra-task blocking** is caused by concurrent critical sections in the same task and **inter-task blocking** is caused by critical sections in other tasks. We describe this calculation in the next two sections (Sections 5.4 and 5.5), for both FIFO-ordered and priority-ordered locks. The overall work blocking and critical-path blocking for the two types of spin locks are presented in Section 5.6.

### 5.4 Calculating Intra-task Blocking

In this section, we present upper-bounds on the intra-task blocking components for the work-blocking \(B^C_i\) and the critical-path blocking \(B^L_i\) of task \(\tau_i\). Since all requests from the same task have the same priority, they are satisfied in FIFO order for both FIFO-ordered and priority-ordered locks. Therefore, the bounds for intra-task blocking for both lock types are identical.

**Intra-Task Work Blocking:** We consider the intra-task blocking component for the work blocking \(B^C_i\) of task \(\tau_i\).
Lemma 14. For a shared resource $l_q$, the intra-task blocking of task $\tau_i$ is upper bounded by:

$$Intra\_Wb = \left( \frac{\min(R_{i,q},n_i)-\min(R_{i,q},n_i)-1}{2} \right) + (n_i - 1) \cdot \max(R_{i,q} - n_i, 0) \right) \cdot \Phi_{i,q}.$$

Proof. Consider the case when the number of requests to $l_q$ from $\tau_i$ is larger than its number of cores, i.e., $R_{i,q} \geq n_i$. In this case, the above expression becomes $\left( \frac{n_i(n_i-1)}{2} + (n_i - 1) \cdot (R_{i,q} - n_i) \right) \cdot \Phi_{i,q}$. First, we observe that a particular request $r$ of a job of $\tau_i$ can wait for at most $n_i - 1$ other requests: the job has $n_i$ total cores and since a core cannot issue another request until its previous requests are satisfied, there can be at most $n_i - 1$ unsatisfied requests when $r$ arrives. Since the requests are satisfied in FIFO order, $r$ can only wait for requests that were already in the queue when it arrived. Second, not all requests can block for this long. In particular, the first request from the job does not block on any other request from this job. Similarly, the second request waits (blocks) only for the first request; the third waits for at most two, and so on. Thus, the first $n_i$ requests block for at most $1 + 2 + 3 + ... + (n_i - 1) = \frac{n_i(n_i-1)}{2}$ requests. The remaining $R_{i,q} - n_i$ requests each may have to wait for $n_i - 1$ preceding requests, giving the total bound.

For the case when $R_{i,q} < n_i$, the expression becomes $\left( \frac{R_{i,q}(R_{i,q}-1)}{2} \right) \cdot \Phi_{i,q}$. Similar to the above observation, the maximum blocking happens when all $R_{i,q}$ requests arrive at once, each from a core of $\tau_i$. The total blocking is then $\left( 1 + 2 + 3 + ... + (R_{i,q} - 1) \right) \cdot \Phi_{i,q} = \frac{R_{i,q}(R_{i,q}-1)}{2} \cdot \Phi_{i,q}$. \qed

**Intra-Task Critical-Path Blocking:** We now present the intra-task blocking component for the critical-path blocking $B^L_i$ of $\tau_i$. Consider a path $P$ of the DAG of $\tau_i$ which contains $Y$ critical sections of lock $l_q$. The blocking collected along $P$ due to contention for $l_q$ within the task $\tau_i$ is bounded as follows.

Lemma 15. The intra-task blocking collected along a path $P$ of task $\tau_i$ which contains $Y$ critical sections of lock $l_q$ is bounded by: $Intra\_Cpb(Y) = \min\{(n_i - 1) \cdot Y, R_{i,q} - Y\} \cdot \Phi_{i,q}$. 

102
Proof. Since at any time, only one processor executes nodes on the path \( P \), each request from \( P \) can be blocked by at most \( n_i - 1 \) requests from the other processors of \( \tau_i \). Summing over all \( Y \) requests of \( P \) gives us the first term. Since the total number of requests of the other processors of \( \tau_i \) to resource \( l_q \) is \( R_{i,q} - Y \), at most that many requests can block requests in \( P \). Because both bounds must hold, the minimum of the two quantities is an upper bound on the intra-task blocking.

This allows us to find the maximum intra-task blocking that can accumulate on path \( P \) for lock \( l_q \) by considering all possible values for \( Y \in [1, R_{i,q}] \). Repeating the calculation for all locks that \( \tau_i \) accesses and summing them up gives the upper-bound on the blocking that any single path of \( \tau_i \) may incur due to contention within the same task.

5.5 Calculating Inter-task Blocking

In this section, we present upper-bounds for the inter-task blocking component of the work-blocking \( B^W_i \) and critical-path blocking \( B^L_i \) of task \( \tau_i \). Here, the bounds for inter-task blocking are different for FIFO-ordered and priority-ordered locks. We first show the bounds for FIFO-ordered locks.

5.5.1 Inter-Task Blocking for FIFO-Ordered Locks

\textbf{Inter-Task Work Blocking:} We first bound the inter-task component of the work blocking for FIFO-ordered locks.

\textbf{Lemma 16.} For FIFO-ordered spin locks, the inter-task blocking of task \( \tau_i \) with respect to resource \( l_q \) caused by a task \( \tau_j \) is upper bounded by: \( \text{Inter\_Wb\_Fifo}(\tau_j) = \min\{R_{i,q} \cdot n_j, n\text{jobs}(\tau_j, D_i) \cdot R_{j,q} \cdot n_i\} \cdot \Phi_{j,q} \).
Proof. The first term is derived by calculating the maximum number of requests from \( \tau_j \) that can block a particular request from \( \tau_i \). Since requests are satisfied in FIFO order and \( \tau_j \) has \( n_j \) cores, each request from \( \tau_i \) can wait for at most \( n_j \) requests from \( \tau_j \). Summing over \( \tau_i \)’s \( R_{i,q} \) requests, at most \( R_{i,q} \cdot n_j \) requests from \( \tau_j \) can block \( \tau_i \) overall. The second term is derived by calculating the maximum number of requests from \( \tau_i \) that can be blocked by a particular request \( r \) of \( \tau_j \). Note that \( r \) can block at most \( n_i \) requests of \( \tau_i \), i.e., one request per core. While a job of \( \tau_i \) is pending, at most \( \text{njobs}(\tau_j, D_i) \) jobs of \( \tau_j \) can execute; therefore, at most \( \text{njobs}(\tau_j, D_i) \cdot R_{j,q} \) requests of \( \tau_j \) can arrive while \( \tau_i \) is pending. Since both quantities are independently upper bounds, the blocking cannot exceed either of them. Therefore, the minimum of the two quantities is also an upper bound. 

**Inter-Task Critical-Path Blocking:** Again, we consider a path \( P \) of \( \tau_i \) that contains \( Y \) critical sections of lock \( l_q \). The blocking of \( P \) with respect to resource \( l_q \) caused by contention with other tasks is bounded as follows.

**Lemma 17.** For FIFO-ordered spin locks, the inter-task blocking accumulated along a path \( P \) of \( \tau_i \) which contains \( Y \) requests to \( l_q \) is bounded by:

\[
\text{Inter}_Cpb\_Fifo(Y) = \sum_{\tau_j \in \mathcal{R}\{\tau_i\}} \min\{n_j \cdot Y, \text{njobs}(\tau_j, D_i) \cdot R_{j,q}\} \cdot \Phi_{j,q}.
\]

Proof. If task \( \tau_j \) contends with \( \tau_i \) for lock \( l_q \), for each request \( r \) to \( l_q \) from \( P \), at most \( n_j \) requests of \( \tau_j \) can block \( r \). Since \( P \) has \( Y \) requests to \( l_q \), it can be blocked by at most \( n_j \cdot Y \) requests of \( \tau_j \). The inter-task blocking on \( P \) is also bounded by the maximum number of requests of \( \tau_j \) that can interfere with requests from \( P \). Because at most \( \text{njobs}(\tau_j, D_i) \) jobs of \( \tau_j \) can interfere with a job of \( \tau_i \) and each job of \( \tau_j \) has total \( R_{j,q} \) requests, the inter-task blocking is bounded by \( \text{njobs}(\tau_j, D_i) \cdot R_{j,q} \). Taking the minimum of the two quantities and summing over all tasks \( \tau_j \) give us the bound on inter-task blocking.

\( \square \)
5.5.2 Inter-Task Blocking for Priority-Ordered Locks

For each shared resource, each task has a *distinct* locking-priority that is used to choose the next request to obtain the lock — a request with a higher locking-priority should acquire the lock before a request with a lower one. Note that locking-priorities are only used to schedule requests to shared resources, and not to schedule the tasks themselves since tasks do not share cores. In this section, we assume the locking-priorities have already been assigned for the tasks. Our evaluation in Section 5.7 explores three possible strategies for assigning locking-priorities. Also, as was mentioned in Section 5.4, all requests from the same task have the same priority.

In order to bound the inter-task blocking for priority-ordered locks, we must calculate a quantity called “delay-per-request”, which bounds the maximum amount of time a request can be delayed due to contention with other requests for the same resource. This allows us to compute the maximum number of interfering jobs that can be pending while the request is delayed, which can then be used to bound the maximum blocking experienced by the task. For simplicity of exposition, in this section we assume that all requests have identical length $\Phi$. We generalize the blocking bounds for different request lengths in Section 5.6.

**Delay Per Request Calculation:** Let $dpr(\tau_i, l_q)$ denote the maximum time a single request of task $\tau_i$ may have to wait until it acquires lock $l_q$. Specifically, a request from $\tau_i$ may have to wait for requests with higher, lower, and equal locking-priorities before it can get the lock and enter its critical section. Let $\tau_i^{LP}$ and $\tau_i^{HP}$ denote a set of tasks with lower and higher locking-priorities than $\tau_i$’s, respectively. The “delay-per-request” $dpr(\tau_i, l_q)$ is thus computed by:

$$dpr(\tau_i, l_q) = higher(\tau_i, l_q) + lower(\tau_i, l_q) + equal(\tau_i, l_q);$$

where $higher(\tau_i, l_q)$, $lower(\tau_i, l_q)$,
and \( equal(\tau_i, l_q) \) are the delays caused by higher, lower, and equal locking-priority requests, respectively.

Since each request of \( \tau_i \) can only be preceded by at most one lower locking-priority request, \( lower(\tau_i, l_q) = \Phi \), which is the length of a single request. The third term, \( equal(\tau_i, l_q) \) accounts for the delay caused by equal locking-priority requests that can only originate from the same task \( \tau_i \), since each task has a distinct priority. Hence, \( equal(\tau_i, l_q) = \min(n_i - 1, R_{i,q} - 1)\Phi \) due to the FIFO ordering between requests from the same task. Note that the calculation of \( equal(\tau_i, l_q) \) is similar to that of intra-task critical-path blocking (Section 5.4), except that it only accounts for the delay incurred by a single request of \( \tau_i \) to \( l_q \).

The calculation of \( higher(\tau_i, l_q) \) is more involved. Observe that a request of \( \tau_i \) potentially can be delayed by requests from multiple jobs of a task with higher locking-priority since jobs may complete and new jobs arrive while this request is still waiting. Therefore,

\[
higher(\tau_i, l_q) = \sum_{\tau_j \in \tau_{i, HP}} njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot \Phi.
\]

This calculation is derived as follows. Since the number of requests of \( \tau_j \) in the interval equal to \( dpr(\tau_i, l_q) \) is bounded by \( njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \). Summing over all higher locking-priority tasks gives us the equation. Since \( higher(\tau_i, l_q) \) and \( dpr(\tau_i, l_q) \) depend on each other, we use a fixed-point calculation to determine \( dpr(\tau_i, l_q) \). We first set \( dpr(\tau_i, l_q) \) to 0 and progressively recalculate it until it converges. However, if \( dpr(\tau_i, l_q) \) is larger than \( D_i \), the task is unschedulable.

**Inter-Task Work Blocking:** We first state a straightforward lemma for the blocking caused by lower locking-priority tasks.

**Lemma 18.** The inter-task blocking of task \( \tau_i \) caused by lower locking-priority tasks with respect to \( l_q \) is at most \( Inter\_Wb\_Lower = R_{i,q} \cdot \Phi \).
Proof. Each request \( r \) of \( \tau_i \) can only be blocked by a single lower locking-priority request that is executing when \( r \) arrives. Summing over all requests gives us the bound.

Next, we bound the blocking caused by higher priority tasks.

**Lemma 19.** The inter-task blocking of task \( \tau_i \) caused by higher locking-priority tasks with respect to resource \( l_q \) is bounded by:

\[
\text{Inter}_Wb_{\text{Higher}} = \sum_{\tau_j \in \tau_i^{HP}} \min\{njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot n_i, njobs(\tau_j, D_i) \cdot R_{j,q} \cdot n_i\} \cdot \Phi.
\]

Proof. \( dpr(\tau_i, l_q) \) is the maximum delay a particular request of job \( \tau_i \) can incur. During this time, there can be at most \( njobs(\tau_j, dpr(\tau_i, l_q)) \) jobs of \( \tau_j \). Hence, the total number of requests from \( \tau_j \) that can delay a particular request of \( \tau_i \) is at most \( njobs(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \). Multiplying by the total number of requests of \( \tau_i \) gives us the maximum inter-task blocking caused by higher locking-priority requests of \( \tau_j \).

Recall that \( njobs(\tau_j, D_i) \cdot R_{j,q} \) is an upper bound on the number of requests to \( l_q \) sent by jobs of task \( \tau_j \) when \( J_i \) is pending. Each of these requests can block at most \( n_i \) requests of \( \tau_i \). Thus, the inter-task blocking of \( \tau_i \) caused by \( \tau_j \) is also bounded by \( njobs(\tau_j, D_i) \cdot R_{j,q} \cdot n_i \). Hence, the inter-task blocking is bounded by the minimum of the two. Summing over all higher locking-priority tasks in \( \tau_i^{HP} \) yields the inter-task blocking caused by higher locking-priority tasks.

**Inter-Task Critical-Path Blocking:** Again, we consider a path \( P \) of task \( \tau_i \) which contains \( Y \) critical sections of lock \( l_q \).

**Lemma 20.** For priority-ordered spin locks, the inter-task blocking accumulated on path \( P \) of task \( \tau_i \) that has \( Y \) requests to \( l_q \) due to contention for \( l_q \) with lower locking-priority tasks is at most \( \text{Inter}_Cpb_{\text{Lower}}(Y) = Y \cdot \Phi \).
Proof. The proof is similar to Lemma 18, except that path $P$ only has $Y$ requests in total.

**Lemma 21.** For priority-ordered spin locks, the inter-task blocking accumulated on path $P$ of task $\tau_i$ that has $Y$ requests to $l_q$ due to contention for $l_q$ with higher locking-priority tasks is bounded by:

$$\text{Inter}_Cpb_{\text{Higher}}(Y) = \sum_{\tau_j \in \tau_{HP}^i} \min\{\text{njobs}(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot Y, \text{njobs}(\tau_j, D_i) \cdot R_{j,q}\} \cdot \Phi.$$ 

Proof. Similar to Lemma 19, the number of requests of $\tau_j$ that can delay a particular request of $\tau_i$ is bounded by $\text{njobs}(\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q}$. Since path $P$ has $Y$ requests to $l_q$, the first term is derived directly.

The second term is derived by calculating the number of requests of $\tau_j$ that can interfere with a job of $\tau_i$. There are $\text{njobs}(\tau_j, D_i) \cdot R_{j,q}$ such requests. It is also an upper bound on the number of requests of $\tau_j$ that can contribute to the blocking accumulated on path $P$. Since both bounds must hold, we take the minimum of these two quantities. Summing over all higher locking-priority tasks in $\tau_{HP}^i$ gives us the bound.

### 5.6 Overall Blocking Bounds

We now present the overall bounds for both FIFO-ordered and priority-ordered locks, by combining the results from Sections 5.4 and 5.5.

#### 5.6.1 Bounds for FIFO-Ordered Locks

To calculate the total work blocking for FIFO-ordered locks, we simply combine the results from Lemmas 14 and 16 and take the sum for all resources that the task accesses.
Theorem 22. The work blocking $B^C_i$ of task $\tau_i$ with FIFO-ordered spin locks is upper bounded by

$$B^C_i \leq \sum_{l_q \in Q_i} \left( \text{Intra}_Wb + \sum_{\tau_j \in \tau \setminus \{\tau_i\}} \text{Inter}_Wb_{Fifo}(\tau_j) \right).$$

Similarly, the critical-path blocking is calculated by combining the results of Lemmas 15, and 17.

Theorem 23. The critical-path blocking $B^L_i$ of task $\tau_i$ with FIFO-ordered spin locks is upper bounded by

$$B^L_i \leq \sum_{l_q \in Q_i} \max_{Y \in [1, R_{i,q}]} \left( \text{Intra}_{Cpb}(Y) + \text{Inter}_{Cpb}_{Fifo}(Y) \right)$$

where, for each resource $l_q$, $Y$ is the number of requests to $l_q$ in a single path of task $\tau_i$.

Proof. By definition, the critical-path blocking of task $\tau_i$ is an upper bound on the blocking that can be collected along a single path $P$ of $\tau_i$. Note that in general, as $Y$ increases the intra-task blocking on $P$ reduces, while the inter-task blocking increases. Thus, for each resource $l_q$, we can maximize over $Y$ ranging from 1 to $R_{i,q}$, and apply Lemmas 15 and 17 to get the maximum blocking due to $l_q$. In the worst case, the same path experiences maximum blocking due to all resources, so summing over all shared resources that $\tau_i$ accesses gives an upper bound for critical-path blocking.

Example for the Analysis of FIFO-Ordered Spin Locks: As is usual for most blocking analyses, these bounds are not tight for all tasks. Here, we present an example task set where the bounds are tight. Consider a task set with two high-utilization tasks with implicit deadlines: $\tau_1(C_1 = 14, L_1 = 4, D_1 = 10)$ and $\tau_2(C_2 = 6, L_2 = 4, D_2 = 5)$. The
DAGs of the tasks are shown in Figure 5.1. Both tasks have two requests to a single resource protected by a FIFO-ordered lock; each has length of one time unit. Without blocking, the number of cores allocated to each task is computed by Lemma 1: \( n_1 = 2 \) and \( n_2 = 2 \).

We focus on the blocking in task \( \tau_1 \). For \( n_1 = 2 \), we can apply Theorem 22 and get \( B_{C1}^C \leq 5 \). Figure 5.2a shows an example schedule where \( \tau_1 \) does experience the total blocking of 5. Similarly, we can apply Theorem 23 to calculate \( B_{L1}^L \leq 4 \). Figure 5.2b shows an example schedule for the critical-path blocking. Therefore, both bounds are tight for this task set, albeit for different schedules. In both schedules, \( \tau_1 \) misses deadlines with \( n_1 = 2 \). The updated number of cores allocated to \( \tau_1 \), \( n'_1 = \lceil \frac{14+5-4-4}{10-4-4} \rceil = 6 \) (Equation 5.1), is sufficient to guarantee that \( \tau_1 \) meets its deadlines as shown in Figure 5.3.
5.6.2 Bounds for Priority-Ordered Locks

We now state the bounds for work and critical-path blocking for priority-ordered locks, assuming that all requests have identical length $\Phi$. The bounds for when request lengths may differ are presented at the end of this section. The proofs are similar to the ones for FIFO locks — for work blocking, we simply sum the bounds from Lemmas 14, 18, and 19 over all shared resources. For critical-path blocking, we combine Lemmas 15, 20, and 21 and
maximize over $Y$, the number of critical sections on a single path, for each resource and then sum over all resources.

**Theorem 24.** The work blocking $B_i^C$ of task $τ_i$ with priority-ordered spin locks is upper bounded by:

$$B_i^C \leq \sum_{l_q \in Q_i} \left( \text{Intra}_Wb + \text{Inter}_Wb_{-Lower} + \text{Inter}_Wb_{-Higher} \right).$$

**Theorem 25.** The critical-path blocking of task $τ_i$ with priority-ordered spin locks is upper bounded by:

$$B_i^L \leq \sum_{l_q \in Q_i} \max_{Y \in [1,R_{i,q}]} \left( \text{Intra}_Cpb(Y) + \text{Inter}_Cpb_{-Lower}(Y) + \text{Inter}_Cpb_{-Higher}(Y) \right),$$

where, for each resource $l_q$, $Y$ is the number of critical sections of $l_q$ on a single path of task $τ_i$.

The following example shows that the bounds for priority-ordered spin locks are tight.

**Example for the Analysis of Priority-Ordered Spin Locks:** Consider a task set with four implicit deadline tasks: $τ_1(C_1 = 14, L_1 = 4, D_1 = 12)$, $τ_2(C_2 = 12, L_2 = 6, D_2 = 12)$, $τ_3(C_3 = 10, L_3 = 4, D_3 = 8)$, and $τ_4(C_4 = 10, L_4 = 4, D_4 = 8)$. Task $τ_1$ has the same structure as task $τ_1$ in Figure 5.1. The DAGs for the other tasks are shown in Figure 5.4, where tasks $τ_3$ and $τ_4$ have the same DAG. All tasks access a single shared resource protected by a priority-ordered lock. Each task has one request to the shared resource; all requests have lengths of 1 time unit. Without blocking, the numbers of processors allocated to these tasks are $n_1 = 2, n_2 = 1, n_3 = 2, n_4 = 2$ respectively (using Lemma 1). Assume the tasks have the following order of locking-priorities, $τ_2 < τ_1 < τ_3 < τ_4$, with $τ_4$ having the highest locking-priority.
Figure 5.4: Example task set with priority-ordered spin locks. Each node has work of 1 time unit.

We now calculate the blocking bounds for \( \tau_1 \) caused by contention with the other tasks. First the delay-per-request is: 
\[
dpr(\tau_1, l) = \text{lower}(\tau_1, l) + \text{equal}(\tau_1, l) + \text{higher}(\tau_1, l);
\]
where 
\[
\text{lower}(\tau_1, l) = 1, \quad \text{equal}(\tau_1, l) = 0, \quad \text{higher}(\tau_1, l) = n\text{jobs}(\tau_3, dpr(\tau_1, l)) + n\text{jobs}(\tau_4, dpr(\tau_1, l)).
\]
Thus, 
\[
dpr(\tau_1, l) = 1 + 2 \cdot \left\lceil \frac{dpr(\tau_1, l) + 8}{8} \right\rceil.
\]
The value of \( dpr(\tau_1, l) \) converges at 5 time units.

Applying Theorems 24 and 25, we can calculate the bounds for the work blocking and critical-path blocking for \( \tau_1 \): 
\[
B_C^W \leq 5 \quad \text{and} \quad B_L^W \leq 5.
\]
Note that since \( \tau_1 \) has only one request to the shared resource, the work blocking bound is equal to the critical-path blocking bound. Also, there is no intra-task blocking within task \( \tau_1 \). Figure 5.5a shows a schedule that causes the worst case blocking for task \( \tau_1 \). It happens when the request from a job \( J_1 \) of \( \tau_1 \) is delayed by one from a job of \( \tau_2 \) (lower locking-priority) and requests from two consecutive jobs of task \( \tau_3 \) and two consecutive jobs of task \( \tau_4 \) (higher locking-priority) in a back-to-back manner.

The schedule shows that the blocking bounds for priority-ordered locks are tight. Note that \( \tau_1 \) misses deadlines with the current number of processors allocated to it. The new number of processors of \( \tau_1 \) is computed using Equation 5.1: 
\[
n_1' = \left\lceil \frac{14+5-4-5}{12-4-5} \right\rceil = 4.
\]
This guarantees that \( \tau_1 \) will meet all deadlines for the current blocking bounds, as depicted in Figure 5.5b.
(a) $\tau_1$ misses deadline with the worst case blocking on two cores

(b) $\tau_1$ meets deadline with the worst case blocking on four cores

Figure 5.5: Example schedules with the current and updated number of processors for $\tau_1$ with priority-ordered spin locks.

**Generalized Bounds for Priority-Ordered Spin Locks:** The work blocking $B^C_i$ and critical-path blocking $B^L_i$ can be generalized easily for critical sections of different lengths. Note that task $\tau_i$ can be blocked by at most $R_{i,q}$ lower locking-priority requests (Lemma 18). Therefore, the blocking caused by lower locking-priority tasks to $\tau_i$ with respect to resource $l_q$ is bounded by summing the lengths of $R_{i,q}$ longest lower locking-priority requests. We use $\text{lcs}(R_{i,q}, l_q)$ to denote this quantity. Thus, $B^C_i$ and $B^L_i$ are now bounded as follows.

\[
B^C_i \leq \sum_{l_q \in Q_i} \left( \left( \min(R_{i,q}, n_i) \cdot \left( \min(R_{i,q}, n_i) - 1 \right) \right) / 2 + (n_i - 1) \cdot \max(R_{i,q} - n_i, 0) \right) \cdot \Phi_{i,q} + lcs(R_{i,q}, l_q) + \sum_{\tau_j \in \tau_{1,HP}^i} \min\{\text{njobs}((\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot R_{i,q}, \text{njobs}(\tau_j, D_i) \cdot R_{j,q} \cdot n_i) \cdot \Phi_{j,q} \}
\]

\[
B^L_i \leq \sum_{l_q \in Q_i} \max_{Y \in [1, R_{i,q}]} \left( \min\{(n_i - 1) \cdot Y, R_{i,q} - Y \} \cdot \Phi_{i,q} + lcs(Y, l_q) + \sum_{\tau_j \in \tau_{1,HP}^i} \min\{\text{njobs}((\tau_j, dpr(\tau_i, l_q)) \cdot R_{j,q} \cdot Y, \text{njobs}(\tau_j, D_i) \cdot R_{j,q}) \cdot \Phi_{j,q} \}
\]
5.6.3 Discussion

We now discuss some qualitative properties of our bounds and some possible avenues for improvement. Given our constraints (we don’t know the graph structures of DAG tasks and don’t control the internal schedules of their job releases), our individual bounds for work blocking and critical-path blocking are as tight as they can be, in the sense that there exist task sets and respective schedules in which the actual worst case work blocking (critical-path blocking, respectively) is exactly equal to the work blocking bound (critical-path blocking bound, respectively) calculated using our method. However, the worst cases for work blocking and critical-path blocking may not both happen in one schedule. For instance, Figure 5.2a shows a schedule in which $\tau_1$ incurs the worst case work blocking of 5 (and critical-path blocking of 3), whereas Figure 5.2b shows a different schedule in which $\tau_1$ incurs the worst case critical-path blocking of 4 (and work blocking of 4). The reason is that different arrangements of resource requests among processors of a task may cause different values for work blocking and critical-path blocking for each task, and the one that causes worst case work blocking may not cause worst case critical-path blocking, and vice versa. However, in general, in a schedule where a task incurs its worst case work blocking, its critical-path blocking in that schedule is also close to its worst case critical-path blocking, and vice versa. Because of this, even though the numbers of processors allocated to tasks using Lemma 13 may be more than the actual numbers of processors required by the tasks to be schedulable, the differences are small.

Secondly, we focus on analyzing tasks with high-utilizations, but our analyses can also be extended to work for task sets with both low and high-utilization tasks. For systems that have low-utilization tasks, if the low-utilization tasks do not share any resources with high-utilization tasks, then we can directly apply an existing synchronization protocol for
sequential tasks (e.g., [35, 54]) and apply the respective analysis to low-utilization tasks, since federated scheduling executes them sequentially using existing multiprocessor schedulers (e.g., P-EDF) on the remaining set of cores. However, if the low-utilization tasks share resources with the high-utilization tasks, one possible method is to extend the Multiprocessor Stack Resource Policy (MSRP) [54] to resources shared by low- and high-utilization tasks. In particular, shared resources are categorized as global resources — resources accessed by tasks from different processors, and local resources — resources accessed only by tasks from the same processor. The low-utilization tasks are scheduled sequentially on the remaining set of processors using schedulers such as P-EDF. Local resources can only be shared between low-utilization tasks allocated to the same processor. Therefore, the Stack Resource Policy (SRP) [7] can be used to coordinate access to local resources on each processor running low-utilization tasks. For global resources, spin locks can be used to coordinate accesses from both high- and low-utilization tasks. The analysis for high-utilization tasks then also would need to account for blocking caused by low-utilization tasks, e.g., for FIFO-ordered spin locks, a request from a high-utilization task can be blocked by at most one request from each processor allocated to low-utilization tasks. Similarly, low-utilization tasks must account for blocking caused by contention for global resources.

5.7 Numerical Evaluation

In this section, we evaluate our schedulability analysis on randomly generated tasks to see which lock ordering offers better schedulability. We also evaluate the effect of the number of cores, number of tasks in the task set, total utilization, number of shared resources, and number of critical sections, on schedulability.
**Task Set Generation:** We randomly generated task sets for systems with varying numbers of cores $m \in \{12, 24, 36\}$. Periods of the parallel tasks were generated to be $2^\lambda \mu s$, where integer $\lambda$ was uniformly chosen from $[13, 20]$. The periods hence were in the range from approximately 8 milliseconds to 1 second, which covers a wide range of real-time applications. Critical-path lengths of the tasks were generated proportionally to the periods: the ratio between critical-path length and period for each task was chosen uniformly from $\{0.125, 0.125, 0.125, 0.125, 0.1625, 0.1625, 0.1625, 0.1875, 0.1875, 0.25\}$. For each value of $m$, tasks were generated with utilizations in the range $[1.25, \sqrt{m}]$, so that tasks can have higher utilizations on systems with larger numbers of cores, which reveals the scalability of the system. The total utilization of task sets $U$ was varied over $\{0.5m, 0.625m, 0.75m\}$. We used Stafford’s RandFixedSum algorithm [47, 93] to generate tasks’ utilizations. This algorithm guarantees that the generated utilizations are sampled uniformly in the range $[1.25, \sqrt{m}]$, and that they sum to the chosen total utilization $U$. The algorithm also allows us to control the number of tasks $n$ in the task sets.

The number of shared resources was varied among $\{1, 2, ..., n\}$. For each shared resource, we varied the total number of critical sections across all tasks of the task set and randomly assigned them to the tasks. We consider two types of critical section length: short and moderate. The lengths were picked uniformly between $[1\mu s, 15\mu s]$ for short critical sections, and $[1\mu s, 100\mu s]$ for moderate ones. In our numerical evaluation, for each parameter configuration we generated 1000 task sets and measured the percentage of schedulable task sets. Schedulability of each task set was also gauged using Algorithm 4.

**Locking-priority Assignment:** While finding the best method to assign locking-priorities is beyond the scope of this work, we are interested in the performance of priority-ordered spin locks with a reasonable locking-priority assignment. Hence, we consider three strategies to assign locking-priorities. In the first strategy, DM, we simply assign the locking-priorities
based on the tasks’ relative deadlines (i.e., in a deadline-monotonic manner). Therefore, tasks with tighter deadlines get higher locking-priority, which is reasonable since it reflects the urgency of each task. In the second strategy, OPT, we search for an optimal assignment by trying all permutations of priorities. For each permutation, we apply Algorithm 4 to test the schedulability with that assignment. The procedure stops when we find an assignment that renders the task set schedulable, or after all permutations have been checked. This gives us the best results for priority-ordered locks at the cost of increasing the running time of the test. The last strategy, SIM, implements simulated annealing to find an approximately optimal locking-priority assignment. We define the cost function for an assignment to be the total number of cores required for the task set to be schedulable. Starting from an original solution, which is the same locking-priority as in DM, the algorithm picks a random neighbor solution and goes to the neighbor with a probability returned from an acceptance probability function. It stops when it finds a solution that requires less than $m$ cores or after it finishes all iterations. This method improves the running time for OPT without sacrificing too much schedulability for priority-ordered locks.

**FIFO-ordered vs. Priority-ordered:** Figures 5.6, 5.7, 5.8, 5.9 and 5.10 show the schedulability results for representative settings (other settings have similar trends). Note that in figure 5.9 we truncate the curve for optimal locking-priority assignment since for 11 tasks, the exhaustive search is impractical in the worst case. With FIFO-ordered locks, each task experiences interference from requests by all the other tasks, while for priority-ordered locks most of the interference comes from tasks with higher locking-priorities. On the other hand, priority-ordered locks can cause a single request to be blocked by multiple jobs of the same task, which cannot happen for FIFO-ordered locks. Results indicate that the first effect exceeded the other in our experiments, and thus priority-ordered locks yielded higher schedulability. We also observe that while DM gave reasonable schedulability for
Figure 5.6: Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections.

Figure 5.7: Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, 4 shared resources, varying number of critical sections per resource, and short critical sections.

Figure 5.8: Schedulability for $m = 36$, $n = 7$, $U = 0.75m$, varying number of resources, 128 requests per shared resource, and short critical sections.

priority-ordered locks (compared to FIFO-ordered locks), it was much worse than OPT (Figures 5.6, 5.7, 5.8, and 5.10). SIM offers a trade-off between these two: the schedulability was closer to that of OPT while the running time improved significantly. For instance, on our machine (see Section 5.8 for description), the schedulability test for 9 tasks sharing a single resource, where the total number of requests by all tasks to the resource is 896, takes
Figure 5.9: Schedulability for $m = 36$, $n = 11$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections.

Figure 5.10: Schedulability for $m = 12$, $n = 5$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and short critical sections.

Figure 5.11: Schedulability for $m = 36$, $n = 9$, 1 shared resource with a priority-ordered lock, short requests, varying total utilization and number of requests.

an hour for OPT, but only takes a couple of minutes for SIM, and a tenth of a second for DM to finish.

**Effect of the Number of Shared Resources:** Figures 5.7 and 5.8 show the effect of the number of shared resources. In Figure 5.7, the task sets shared 4 resources, but the total number of critical sections over all resources was kept the same as in Figure 5.6. Similarly,
in Figure 5.8 we varied the number of shared resources and kept the number of critical sections per resource at 128. Compared to Figure 5.6 we can see that having critical sections spread across multiple shared resources vs. having them all on one resource doesn’t affect the schedulability much. (Note that the corresponding points on these figures represent the same number of critical sections overall). This seems counter-intuitive since individual requests should experience smaller contention if the requests are spread out across resources. However, the sum of the worst case blocking times over multiple resources turns out to be similar to that of a single resource with the same total number of requests.

Effect of $m$: Comparing Figures 5.6 and 5.10 reveals that the schedulability for 36 cores is higher when the number of requests is small but decreases more quickly when the number of requests increases, compared to 12 cores. In our analysis and experiment design, there are competing factors, some of which favor fewer cores and some more. For the former, $B_i^L$ and $B_i^C$ increase as $m$ increases (since they depend on $n_i$’s and the sum of $n_i$’s is bounded by $m$). For the latter, since the total utilization ($0.75m$) is 9 on 12 cores and is 27 on 36 cores, there is more “absolute slack” (9 vs. 3) for the 36 core experiments. It appears that the latter factor may exceed the former when the number of critical sections is small; however, as the number of critical sections increases, the increase in contention decreases schedulability despite the extra slack.

Effect of Utilization and the Number of Critical Sections: Figure 5.11 shows the schedulability results for $m = 36$ with $n = 9$ tasks and a single shared resource protected by a priority-ordered lock (the results for FIFO-ordered locks are similar). We varied the normalized total utilization in range $[0.1, 1.0]$ with the gap of 0.05 and the number of critical sections in range $[16, 1024]$ with the gap of 16. For each pair of the values for the two parameters, we recorded the percentage of the task sets that are schedulable. The lighter the data point, the greater the percentage of the task sets that are schedulable. Both parameters
have a noticeable impact on schedulability. Also, for very high utilization (above 0.8\textit{m}), the task sets are unschedulable even for a small number of critical sections.

5.8 Empirical Evaluation

We incorporated an implementation of both FIFO-ordered and priority-ordered spin locks into a federated scheduling system for OpenMP programs [76]. The locks are implemented in user-space and provide lock and unlock interfaces to real-time tasks. Typically, spin locks require support for atomic operations; we specifically utilized atomic built-in extensions of GCC [56] to implement our spin locks.

\textbf{FIFO-Ordered Spin Locks:} For FIFO-ordered locks, we implemented an MCS lock, a scalable list-based FIFO spin lock [82]. In the MCS lock approach, each processor maintains a data structure called an MCS node: a structure containing a pointer next to the next MCS node in the waiting list, and a flag spin indicating whether the owner of the MCS node gets the lock. In this approach, requests to a lock form a linked list of MCS nodes, each from a processor. An MCS lock is a pointer to the MCS node at the tail of the list, or null if the list is empty. When a processor sends a request, it atomically obtains the current value of the lock, and sets the lock to point to the processor’s MCS node. If the lock is null, the requesting processor is the first in the list and has successfully acquired the lock. Otherwise, it will append itself to the list by setting the next pointer of the preceding MCS node (i.e., the previous tail of the list) to point to its MCS node, and spin on its MCS node’s spin flag. When a processor finishes its critical section, it passes ownership of the lock to the next processor in the list by flipping the succeeding MCS node’s spin flag to allow it to break out of spinning. If there is no succeeding processor, it simply resets the lock to null and returns.
Algorithm 5 Priority-Ordered Spin Locks

```plaintext
1: procedure Lock(PriorityLock: lock, MCSNode: mynode)
2:    Get the pointer tail to my task’s MCS lock
3:    lock_mcs(tail, mynode)
4:    while (1) do
5:       Check if any higher locking-priority requests exist
6:       if (No higher locking-priority requests exist) then
7:          cmpxchg(lock→owner, null, mynode)
8:             if (cmpxchg was successful) then
9:                Return
10:           end if
11:       end if
12:    end while
13: end procedure

14: procedure UNLOCK(PriorityLock: lock, MCSNode: mynode)
15:    Get the pointer tail to my task’s MCS lock
16:    unlock_mcs(tail, mynode)
17:    lock→owner = null
18: end procedure
```

**Priority-Ordered Spin Locks:** Pseudocode for priority-ordered locks is shown in Algorithm 5. We extend the MCS lock approach for priority-ordered locks where each processor also maintains an MCS node and uses the node whenever it sends a request to a resource. A priority-ordered lock is a structure consisting of `owner`, a pointer pointing to the MCS node of the processor that currently owns the lock, and `task_heads`, an array of MCS locks, each for a task. Note that here as well, an MCS lock is a pointer to the tail MCS node in the list of requests for that lock. Each task is associated with an MCS lock in the array at an index calculated from the locking-priority of requests from the task. For instance, a task with the highest locking-priority uses the MCS lock at index 0, a task with the second highest locking-priority uses the MCS lock at index 1, and so on. Elements in the array are also aligned to the cache line size to avoid false sharing. Each MCS lock in the array is used to accommodate requests issued by processors of the corresponding task. When a processor of a task issues a request, it first calls the MCS’s `lock` method to acquire the task’s MCS lock.
After the processor successfully acquires the task’s MCS lock, it constantly checks for the existence of any higher locking-priority requests (line 5). It does this by checking the values of the MCS locks of the tasks with higher locking-priorities (recall that each MCS lock is a pointer to the tail of a list of MCS nodes from processors of the same task). If all MCS locks for higher locking-priority tasks are null, which means there are no requests from those tasks, the processor will try to acquire the priority-ordered lock by atomically comparing the lock’s owner with null and if so setting owner to point to its MCS node (line 7). It returns if the compare-and-exchange instruction succeeded (line 9). Otherwise, the lock must have been acquired by some higher locking-priority processor. In this case, the processor spins again until there is no higher locking-priority request (line 4). When the processor finishes its critical section, it calls the MCS lock’s unlock method to release its task’s MCS lock and resets the priority-ordered lock’s owner to null (lines 16, 17).

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>FIFO-ordered (ns)</th>
<th>Priority-ordered (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>58</td>
</tr>
<tr>
<td>2</td>
<td>461</td>
<td>143</td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>287</td>
</tr>
<tr>
<td>8</td>
<td>550</td>
<td>526</td>
</tr>
<tr>
<td>12</td>
<td>540</td>
<td>765</td>
</tr>
<tr>
<td>16</td>
<td>609</td>
<td>891</td>
</tr>
<tr>
<td>24</td>
<td>625</td>
<td>1575</td>
</tr>
<tr>
<td>32</td>
<td>676</td>
<td>2185</td>
</tr>
</tbody>
</table>

Table 5.2: Overhead of spin locks

**Spin Locks Overhead:** Table 5.2 summarizes the average overhead per critical section of FIFO-ordered and priority-ordered locks, recorded on the same machine that ran our empirical experiments (see below for description). We created \( m \) threads (\( m \) varies in the first column) and pinned one thread on each core. The threads were synchronized using a pthread barrier. Each thread repeatedly acquires and releases the lock and performs work inside each
critical section. We measured the time between the earliest lock acquisition and the latest lock release among all threads. We also measured the time to perform the same total amount of work (by all threads) on a single core. The difference between these two measurements was divided by the total number of acquire-release pairs to get the average overhead per critical section. We repeated the measurement 100 times and recorded the largest average overhead in Table 5.2. We notice that FIFO-ordered locks implemented by the MCS algorithm show good scalability when the number of cores increases, as expected, whereas priority-ordered locks do not scale as well. This is due to the way each processor, after acquiring its task’s MCS lock, checks for the existence of any higher locking-priority requests (Algorithm 5, line 5). Recall that it does this by constantly reading the MCS locks of all tasks with higher locking-priorities, and only executes the compare-and-exchange instruction when all these MCS locks are null. Thus, when any MCS lock of a higher locking-priority task changes, either by an arrival of a new request or by a departure of the last request, processors (who hold their tasks’ MCS locks) of the lower locking-priority tasks will have to read the new value into their caches. This creates high contention for the shared bus, and thus priority-ordered locks do not scale as well as MCS locks.

**Empirical Results:** We conducted experiments on our federated scheduling platform to observe the performance of the two types of spin locks in practice. Using the same task sets generated in Section 5.7, we constructed synthetic parallel task sets written in OpenMP. Critical sections were distributed randomly inside the structure of the tasks. We ran the experiments on a 48-core machine composed of four AMD Opteron 6168 processors, each with 12 cores. Linux 3.4.4 with RT-PREEMPT patch version 3.4.4-rt14 was used as the underlying RTOS. Since each task was allocated a set of dedicated cores, we used Linux’s CPU mask and `sched_setaffinity` system call to bind each task to its cores. For each parameter configuration, we tested 100 task sets; each task set was run for 100 hyper-periods.
Figure 5.12: Ratio of successful task sets for $m = 36$, $U = 0.75m$, 1 shared resource, varying number of critical sections per resource, and moderate critical sections.

After each experiment finished, we recorded the ratio of task sets in which no job of any task missed its deadline. For all of the settings, we observed a similar trend as in the schedulability analysis (Section 5.7) — priority-ordered locks have better results than FIFO-ordered locks. Figure 5.12 shows a representative result for $m = 36$ cores, total utilization $U = 0.75m$, a single shared resource, and all requests have moderate lengths.

5.9 Summary

In this chapter, we have presented the first blocking analysis and schedulability test for parallel real-time tasks that use spin locks to arbitrate access to shared resources. We analyze blocking times under federated scheduling, which assigns dedicated cores to each task and thus avoids CPU priority inversion. We also incorporated spin lock implementations on a
federated scheduling platform, which allows us to run parallel real-time programs written in OpenMP with spin locks. Our numerical and empirical experiments using randomly generated task sets both indicate that priority-ordered locks outperform FIFO-ordered locks in terms of schedulability. This work opens up new research problems for schedulability tests for parallel task sets which access shared resources: for instance, how to construct a schedulability test when using schedulers, such as G-EDF, where tasks can share cores, and how to design good protocols for systems that have both high and low-utilization tasks.
Chapter 6

Scalable Scheduling Platform for Soft Real-Time Parallel Tasks

In this chapter, we look at a practical aspect of scheduling real-time parallel tasks. In particular, we present a case study that compares two prominent scheduling strategies for parallel tasks, namely centralized scheduling and randomized work stealing, in terms of suitability for soft real-time systems. We show that randomized work stealing combined with federated scheduling can provide a scalable platform for soft real-time applications.

6.1 Introduction

Despite recent results in parallel real-time scheduling, we still face significant challenges in deploying large-scale real-time applications on microprocessors with increasing numbers of cores. In order to guarantee desired parallel execution of a task to meet its deadlines,
theoretic analysis often assumes that it is executed by a greedy (work conserving) scheduler, which requires a centralized data structure for scheduling. On the other hand, for general-purpose parallel job scheduling it has been known that centralized scheduling approaches suffer considerable scheduling overhead and performance bottleneck as the number of cores increases. In contrast, a randomized work stealing approach is widely used in many parallel runtime systems, such as Cilk [21], Cilk Plus [40], TBB [41], X10 [94], and TPL [73]. In work stealing, each core steals work from a randomly chosen core in a decentralized manner, thereby avoiding the overhead and bottleneck of centralized scheduling. However, unlike a centralized scheduler, due to the randomized and distributed scheduling decision making strategy, work stealing may not be suitable for hard real-time tasks.

In this chapter, we explore using randomized work stealing to support large-scale soft real-time applications that have timing constraints but do not require hard guarantees. Despite the unpredictable nature of work stealing, our experiments with benchmark programs found that work stealing (in Cilk Plus) delivers smaller maximum response times than a centralized greedy scheduler (in GNU OpenMP) while exhibiting small variance. To leverage randomized work stealing for scalable real-time computing, we present Real-Time Work Stealing (RTWS), a real-time extension to the widely used Cilk Plus concurrency platform. RTWS employs federated scheduling to decide static core assignment of parallel real-time tasks offline, while using the work stealing scheduler to execute each task on its dedicated cores online. RTWS supports parallel programs written in Cilk Plus with only minimal modifications, namely a single level of indirection of the program’s entry point. Furthermore, RTWS requires only task parameters that can be readily measured using existing Cilk Plus tools.

The rest of this chapter presents the following contributions.
1. An empirical study of the performance and variability of parallel tasks under randomized work stealing vs. centralized greedy scheduler.

2. The design and implementation of RTWS, which schedules multiple parallel real-time tasks through the integration of federating scheduling and work stealing.

3. An evaluation of RTWS with benchmark applications on a 32-core testbed that demonstrates the significant advantages of RTWS in terms of deadline miss ratio, relative response time and required resource capacity, compared to the integration of federated scheduling and a centralized scheduler.

6.2 Parallel Tasks

We are interested in parallel programs that can be generated using parallel languages and libraries, such as Cilk [21], Intel Cilk Plus [40], OpenMP [23], Microsoft’s Task Parallel Library [73], IBM X10 [94]. In these languages, the programmer expresses algorithmic parallelism, through linguistic constructs such as “spawn” and “sync,” “fork” and “join,” or parallel-for loops. These programs can be modeled using the DAG task model (Section 2.4).

In general, parallel programs can have arbitrary DAG structures. In real-time systems, researchers have given special consideration to a subset of DAG tasks, where the programs only use the parallel-for construct and do not nest these parallel-for loops. This restriction generates synchronous tasks, which can be modeled using the synchronous DAG task model discussed in Section 2.3. In that model, each parallel for-loop is represented by a segment, with each of its iterations is represented by a node in the segment. A sequential region of code is simply a segment with 1 iteration. Each synchronous task is a sequence of such segments. Synchronous tasks are also called as Fork/Join tasks in some related works. Figure 6.1 shows
main()
{
    // Do some sequential work
    foo();
    // Do the first parallel segment
    parallel_for (i = 1; i <= 20; i++) {
        first_func();
    }
    // Other sequential work
    bar();
    // Do the second parallel segment
    parallel_for (i = 1; i <= 3; i++) {
        second_func();
    }
    // The last sequential work
    baz();
}

Figure 6.1: Example of a synchronous program.

an example of a parallel program that generates a synchronous structure. In this example, parallel_for constructs can be Cilk Plus’ cilk_for constructs or OpenMP’s omp for directives.

In this chapter, we consider a task set \( \tau \) consists of \( n \) parallel tasks \( \tau = \{\tau_1, \tau_2, \ldots, \tau_n\} \), where each job of a task \( \tau_i \) is either a general DAG program or a synchronous program. In principle, each job may have a different internal structure — the internal structure of a job is data-dependent. We consider sporadic task sets with implicit deadlines, i.e. \( T_i = D_i \). We want to schedule the task set \( \tau \) on an \( m \)-core homogeneous multiprocessor machine.

We are specifically interested in soft real-time tasks where a task is allowed to miss a few deadlines occasionally. Using the same resource capacity, a scheduling algorithm \( S \) has better performance if it schedules the same task set with a smaller **deadline miss ratio**, which is defined as the number of missed deadlines over the number of released jobs of the task set during a time interval.
6.3 Scheduling Parallel Tasks

Most parallel languages and libraries, including those mentioned above, provide a runtime system that is responsible for scheduling the DAG on the available cores, i.e., dynamically dispatching the nodes of the DAG to these cores as the nodes become ready to execute. At a high-level, two types of scheduling strategies are often used: centralized scheduling and randomized work-stealing.

6.3.1 Centralized Schedulers

The system maintains a centralized data structure (such as a queue) of ready nodes that is shared by all the cores in a work sharing manner. There are a couple of possible instantiations of this strategy. In push schedulers, there is a master thread that dispatches work to other threads as they need this work. In pull schedulers, worker threads access this data structure themselves to grab work (ready nodes) as they need them. For example, the scheduler in the runtime system of GNU OpenMP is a pull scheduler, as shown in Figure 6.2a.

Work-sharing schedulers have the nice property that they are greedy or work-conserving — as long as there are available ready nodes, no worker idles. However, these schedulers often have high overheads due to frequent synchronization. In particular, in a push scheduler, the master thread can only send work to cores one at a time. In a pull scheduler, the centralized queue must be protected by a lock and may incur high overheads due to this.

6.3.2 Randomized Work-Stealing Schedulers

In a randomized work-stealing scheduler, there is no centralized queue and the work dispatching is done in a distributed manner [21]. If a job is assigned \( n_i \) cores, the runtime system
creates \( n \) worker threads for it. Each worker thread maintains a local double-ended queue (a *deque*), as shown in Figure 6.2b. When a worker generates new work (enables a ready node from the job’s DAG), it pushes the node onto the bottom of its deque. When a worker finishes its current node, it pops a ready node from the bottom of its deque. If the local deque is empty, the worker thread becomes a *thief* and randomly picks a *victim* thread among the other workers working on the same task and tries to steal work from the top of the victim’s deque. For example, the third worker thread’s deque is empty in Figure 6.2b, so it randomly picks the second worker thread and steals work.

Randomized work-stealing is very efficient in practice and the amount of scheduling and synchronization overhead is usually small. In contrast to centralized schedulers where the threads synchronize frequently, very little synchronization is needed in work-stealing schedulers since (1) workers work off their own deques most of the time and don’t need to communicate with each other at all and (2) even when a worker runs out of work and steals occur, the thief and the victim generally look at the opposite ends of the deque and don’t conflict unless the deque has only 1 node on it.
However, because of this randomized and distributed characteristic, work-stealing is not strictly greedy (work conserving). In principle, workers can spend a large amount of time stealing, even if some other worker has a lot of ready nodes available on its deque. On the other hand, work-stealing provides a strong probabilistic guarantee of linear speedup (“near-greediness”) [95]. Moreover, it is much more efficient than centralized schedulers in practice. Therefore, variants of work stealing are the default strategies in many parallel runtime systems such as Cilk [21], Cilk Plus [40], TBB [41], X10 [94], and TPL [73]. Thus, for soft real-time systems where occasional deadline misses are allowed, work stealing can be more resource efficient than a strictly greedy scheduler.

6.3.3 Specific Implementations of Centralized and Work-Stealing Schedulers

In this chapter, we compare specific implementations of centralized and work-stealing schedulers: GNU OpenMP’s centralized scheduler and GNU Cilk Plus’s work-stealing scheduler. We choose these two implementations because OpenMP and CilkPlus are two of the most widely used parallel languages (and runtime systems) that have been developed by industry and the open source community over more than a decade; they are the only two parallel languages that are supported by both GCC and ICC.

OpenMP is a programming interface standard [23] for C, C++, and FORTRAN that allows programmers to specify where parallelism can be exploited, and the GNU OpenMP runtime library in GCC is one of the implementations of the OpenMP standard. OpenMP allows programmers to express parallelism using compiler directives. In particular, parallel for loops are expressed by \texttt{#pragma omp parallel for}, a parallel node in a DAG is expressed by \texttt{#pragma omp task}, and synchronization between omp tasks is expressed by \texttt{#pragma omp taskwait}. 134
While the details of scheduling are somewhat complex, and vary between omp parallel for loops and omp tasks, at a high level, GNU OpenMP provides an instantiation of a centralized pull scheduler. Available parallel work of a program is kept in a centralized queue protected by a global lock. Whenever a worker thread generates nodes of omp tasks or iterations of a parallel for loop, it has to get the global lock and places nodes with that work in the queue. When it finishes its current work, it again has to grab the lock to get more work from the queue.

Cilk Plus is a language extension to C++ for parallel programs and its runtime system schedules parallel programs using randomized work stealing. All Cilk Plus features are supported by GCC. Potential parallelism can be expressed using three keywords in the Cilk Plus language: a parallel node in a DAG is generated by `cilk_spawn` and the synchronization point is realized by `cilk_sync`; additionally, parallel for-loops are supported using a `cilk_for` programming construct. Note that in the underlying Cilk Plus runtime system, `cilk_for` is expanded into `cilk_spawn` and `cilk_sync` in a divide and conquer manner. Therefore, there is no fundamental difference between executing parallel DAGs or synchronous tasks in Cilk Plus. The Cilk Plus runtime system implements a version of randomized work stealing. When a function spawns another function, the child function is executed and the parent is placed on the bottom of the worker’s deque. A worker always works off the bottom of its own deque. When its deque becomes empty, it picks a random victim and steals from the top of that victim’s deque.
6.4 The Case for Randomized Work Stealing for Soft Real-Time Tasks

In this section, we compare the performance of a work stealing scheduler in GNU Cilk Plus with a centralized scheduler in GNU OpenMP for highly scalable parallel programs. Our goal is to answer two questions: (1) Is it indeed the case that work stealing provides substantially better performance than a centralized scheduler for parallel programs? Our experiments indicate that for many programs, including both synthetic tasks and real benchmark programs, work stealing provides much higher scalability. (2) Can work stealing be used for real-time systems? In particular, one might suspect that even if work stealing performs better than centralized scheduler on average, the randomization used in work stealing would make its performance too unpredictable to use even in soft real-time systems. Our experiments indicate that this is not the case — in fact, the variation in execution time using Cilk Plus’ work-stealing scheduler is small and is comparable to or better than the variation seen in the (supposedly more deterministic) centralized scheduler.

6.4.1 Scalability Comparison

We first compare the scalability of the OpenMP centralized scheduler with the Cilk Plus work-stealing scheduler. To do so, we use two types of programs: (1) three synthetic programs that are synchronous tasks; and (2) three real benchmark programs, namely Cholesky factorization, LU decomposition and Heat diffusion — none is synchronous and all have complex DAG dependences (of different types).

We implemented these programs in both Cilk Plus and OpenMP. It is important to note that in each implementation the entire source code of each program is the same, except that the
parallel directives are in either Cilk Plus or OpenMP. Both implementations are compiled by GCC, and linked to either Cilk Plus or OpenMP runtime libraries. Hence, the same program written in Cilk Plus and OpenMP has the same structure and therefore the same theoretical work and span.

**Synthetic Synchronous Tasks:** The synthetic synchronous tasks have different characteristics to compare the schedulers under different circumstances:

1. **Type 1** tasks have a large number of nodes per segment, but nodes have small execution times.

2. **Type 2** tasks have a moderate number of nodes per segment and moderate work per node.

3. **Type 3** tasks have a small number of nodes per segment, but nodes have large execution times.

The number of segments for all three types of synchronous tasks is generated from 10 to 20. For synchronous task type 1, we generate the number of nodes for each segment from 100,000 to 200,000 and the execution time per node from 5 to 10 nanoseconds; for task type 2, the number of nodes per segment varies from 10,000 to 20,000 and the execution time of each node from 2,000 to 4,500 nanoseconds; for task type 3, the number of nodes for each segment is from 1,000 to 2,000 and each node runs for between 20,000 and 50,000 nanoseconds. The total work for synchronous tasks of different types was therefore similar. For each synchronous task generated, we ran it on varying numbers of cores with both Cilk Plus and OpenMP and we ran it 1000 times for each setting.

Table 6.1 shows the median, maximum, and 99th percentile execution times of OpenMP and Cilk Plus tasks as well as the ratios of the maximum execution time of Cilk Plus over
<table>
<thead>
<tr>
<th>No. cores</th>
<th>Synchronous Task - Type 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>OpenMP (med., max, 99(^{th}) per.)</td>
<td>Cilk Plus (med., max, 99(^{th}) per.)</td>
<td>Ratio</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>955.13, 958.10, 956.98</td>
<td>948.52, 953.41, 950.94</td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>173.68, 174.31, 174.18</td>
<td>160.77, 161.46, 161.26</td>
<td>0.93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>256.63, 259.19, 258.89</td>
<td>81.68, 82.56, 81.93</td>
<td>0.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>342.20, 365.99, 362.99</td>
<td>55.42, 59.22, 58.96</td>
<td>0.16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>328.52, 331.11, 329.75</td>
<td>41.23, 45.22, 44.78</td>
<td>0.14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>311.92, 330.00, 329.00</td>
<td>33.66, 35.02, 34.64</td>
<td>0.11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No. cores</th>
<th>Synchronous Task - Type 2</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>OpenMP (med., max, 99(^{th}) per.)</td>
<td>Cilk Plus (med., max, 99(^{th}) per.)</td>
<td>Ratio</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1243.7, 1247.2, 1246.6</td>
<td>1237.2, 1239.9, 1239.2</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>210.22, 210.84, 210.74</td>
<td>213.77, 214.30, 214.19</td>
<td>1.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>111.58, 111.94, 111.87</td>
<td>107.90, 108.69, 108.11</td>
<td>0.97</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>95.55, 95.96, 95.92</td>
<td>73.45, 73.82, 73.62</td>
<td>0.77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>85.97, 126.00, 123.01</td>
<td>58.95, 74.80, 69.18</td>
<td>0.59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>86.74, 119.01, 86.96</td>
<td>45.07, 48.27, 47.33</td>
<td>0.41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No. cores</th>
<th>Synchronous Task - Type 3</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>OpenMP (med., max, 99(^{th}) per.)</td>
<td>Cilk Plus (med., max, 99(^{th}) per.)</td>
<td>Ratio</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>948.42, 950.39, 949.97</td>
<td>902.38, 903.29, 903.18</td>
<td>0.95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>156.47, 156.94, 156.77</td>
<td>155.77, 156.06, 156.00</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>79.03, 79.34, 79.27</td>
<td>78.80, 79.46, 78.97</td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>53.07, 53.49, 53.28</td>
<td>54.05, 54.41, 54.29</td>
<td>1.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>39.99, 69.95, 40.18</td>
<td>40.68, 44.35, 43.62</td>
<td>0.63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>32.20, 33.18, 32.39</td>
<td>33.40, 37.12, 34.48</td>
<td>1.12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Median, maximum, and 99\(^{th}\) percentile execution times of synchronous tasks for OpenMP and Cilk Plus implementations (in milliseconds) and the ratios of the maximum execution times of Cilk Plus over OpenMP implementations.
OpenMP implementations for the three types of synchronous tasks on varying numbers of cores. For most settings, Cilk Plus tasks obtain smaller maximum execution times than OpenMP tasks, as shown in the ratios. We also notice that for type 1 tasks the execution times of the OpenMP tasks may even increase when the number of cores is high (e.g., for 18, 24, 30 cores) whereas Cilk Plus tasks keep a steady speedup.

Figure 6.3 shows the speedup of these synchronous tasks. For all three types of tasks, Cilk Plus provides steady and almost linear speedup as we scale up the number of cores. In contrast, for synchronous task type 1 in Figure 6.3a where the segment lengths are short
and there are many nodes in each segment, OpenMP inevitably suffers high synchronization overhead due to the contention among threads that frequently access the global work queue. This overhead is mitigated when the number of nodes in each segment is smaller and the segment lengths are longer, as in Fig. 6.3c. In this setting, OpenMP slightly outperforms Cilk Plus, though Cilk Plus still has comparable speedup to OpenMP. Figure 6.3b demonstrates the scalability of OpenMP and Cilk Plus with parameters generated in between those more extreme cases, with correspondingly intermediate results: performance is equivalent up to 12 cores, after which Cilk Plus dominates.

**Real DAG Benchmark Programs:** To compare the performance differences between work stealing and a centralized scheduler for programs with more complex DAG structures, we use three benchmark programs as described below.

(a) **Cholesky factorization (Cholesky):** Using divide and conquer, the Cholesky program performs Cholesky factorization of a sparse symmetric positive definite matrix into the product of a lower triangular matrix and its transpose. The work and parallelism of the Cholesky program both increase when the matrix size increases. Note that because Cholesky is parallelized using divide and conquer method, it has lots of spawn and sync operations, forming a complex DAG structure.

(b) **LU decomposition (LU):** Similar to Cholesky, the LU program also performs matrix factorization, but the input matrix does not need to be positive definite and the output upper triangular matrix is not necessarily the transpose of the lower triangular matrix. LU also decomposes the matrix using divide and conquer and provides abundant parallelism.

(c) **Heat diffusion (Heat):** This program uses the Jacobi iterative method to solve an approximation of a partial differential equation that models the heat diffusion problem. The input includes a 2-dimensional grid with the numbers of rows and columns, and the number
### Cholesky Factorization

<table>
<thead>
<tr>
<th>No. cores</th>
<th>OpenMP (med., max, 99&lt;sup&gt;th&lt;/sup&gt; per.)</th>
<th>Cilk Plus (med., max, 99&lt;sup&gt;th&lt;/sup&gt; per.)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32.12, 32.18, 32.17</td>
<td>32.31, 32.36, 32.35</td>
<td>1.01</td>
</tr>
<tr>
<td>6</td>
<td>7.39, 7.62, 7.61</td>
<td>5.44, 5.47, 5.47</td>
<td>0.72</td>
</tr>
<tr>
<td>12</td>
<td>3.58, 3.72, 3.71</td>
<td>2.79, 2.89, 2.87</td>
<td>0.78</td>
</tr>
<tr>
<td>18</td>
<td>2.36, 2.43, 2.43</td>
<td>1.91, 1.96, 1.95</td>
<td>0.81</td>
</tr>
<tr>
<td>24</td>
<td>1.85, 1.92, 1.92</td>
<td>1.48, 1.52, 1.51</td>
<td>0.79</td>
</tr>
<tr>
<td>30</td>
<td>1.56, 1.62, 1.61</td>
<td>1.23, 1.28, 1.28</td>
<td>0.79</td>
</tr>
</tbody>
</table>

### LU Decomposition

<table>
<thead>
<tr>
<th>No. cores</th>
<th>OpenMP (med., max, 99&lt;sup&gt;th&lt;/sup&gt; per.)</th>
<th>Cilk Plus (med., max, 99&lt;sup&gt;th&lt;/sup&gt; per.)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16.98, 17.09, 17.07</td>
<td>16.76, 16.82, 16.82</td>
<td>0.98</td>
</tr>
<tr>
<td>6</td>
<td>3.53, 3.79, 3.79</td>
<td>2.82, 2.84, 2.83</td>
<td>0.75</td>
</tr>
<tr>
<td>12</td>
<td>1.89, 1.97, 1.97</td>
<td>1.44, 1.87, 1.78</td>
<td>0.95</td>
</tr>
<tr>
<td>18</td>
<td>1.27, 1.37, 1.35</td>
<td>0.99, 1.07, 1.06</td>
<td>0.78</td>
</tr>
<tr>
<td>24</td>
<td>0.99, 1.06, 1.05</td>
<td>0.76, 0.84, 0.83</td>
<td>0.79</td>
</tr>
<tr>
<td>30</td>
<td>0.82, 0.86, 0.86</td>
<td>0.64, 0.71, 0.69</td>
<td>0.82</td>
</tr>
</tbody>
</table>

### Heat Diffusion

<table>
<thead>
<tr>
<th>No. cores</th>
<th>OpenMP (med., max, 99&lt;sup&gt;th&lt;/sup&gt; per.)</th>
<th>Cilk Plus (med., max, 99&lt;sup&gt;th&lt;/sup&gt; per.)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51.57, 52.04, 52.04</td>
<td>51.70, 52.11, 52.11</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>13.50, 13.83, 13.81</td>
<td>8.80, 9.28, 9.26</td>
<td>0.67</td>
</tr>
<tr>
<td>12</td>
<td>7.93, 8.41, 8.31</td>
<td>5.06, 5.82, 5.70</td>
<td>0.69</td>
</tr>
<tr>
<td>18</td>
<td>6.40, 6.73, 6.69</td>
<td>3.73, 3.96, 3.95</td>
<td>0.59</td>
</tr>
<tr>
<td>24</td>
<td>5.94, 6.10, 6.10</td>
<td>3.06, 4.06, 3.67</td>
<td>0.67</td>
</tr>
<tr>
<td>30</td>
<td>6.87, 7.20, 7.17</td>
<td>2.62, 2.73, 2.73</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Table 6.2: Median, maximum, and 99<sup>th</sup> percentile execution times of Cholesky, LU, and Heat for OpenMP and Cilk Plus implementations (in seconds) and the ratio of the maximum execution times of Cilk Plus over OpenMP implementations.
of time steps (or iterations) over which the computation is performed on that 2D grid. Within each time step, the computation is carried out in a divide and conquer manner.

Figure 6.4: Speedup of benchmark programs in OpenMP and Cilk Plus implementations

The Cholesky program was run for a matrix of size $3000 \times 3000$. The LU program was run for a matrix of size $2048 \times 2048$. For both of them, the base case matrix had size $32 \times 32$. The Heat program was run with a 2-dimensional input of size $4096 \times 1024$ and 800 time steps. For each setting, we ran the program 100 times.
For each program, we first compare its execution times under work stealing and the centralized scheduler on varying numbers of cores, as shown in Table 6.2. For all three benchmarks, we notice that the execution times are tight which means both scheduling strategies have reasonable predictability. However, Cilk Plus implementations have smaller maximum execution times which means that Cilk Plus tasks have higher chance of finishing by their deadlines.

Figure 6.4 shows the speedups of these programs in the same experiments. For matrix computation programs like Cholesky and LU, where there is abundant parallelism, OpenMP obtains good speedups but Cilk Plus obtains even better speedups. The difference is more notable in the Heat diffusion program, where there is less parallelism to exploit. For this program, Cilk Plus still has reasonable speedup, while the speedup of OpenMP starts to degrade when the number of cores is more than 21.

Figure 6.5: Cholesky with input size 1000 × 1000 base case 4 × 4
We also notice that for Cholesky and LU programs, the performances of OpenMP are quite sensitive to the base case sizes whereas Cilk Plus performed equally well regardless of the base case sizes. For example, Figure 6.5 shows the experiment results of Cholesky with a base case matrix of size $4 \times 4$. Notably, no speedup was observed for OpenMP when the number of cores increases. Thus, one has to tune the base case sizes for OpenMP in order to get comparable performance with their Cilk Plus counterparts. This is again caused by the fact that the overhead of centralized scheduler adds up and outweighs the performance gain by running a program in parallel, when the base case is small.

### 6.4.2 Tightness of Randomized Work Stealing in Practice

One might expect that even though a work-stealing scheduler may perform well on average due to low overheads, it would not be suitable for real-time platforms due to high variability in its execution times due to randomness. However, this intuition turns out to be inaccurate. Theoretically, strong high probability bounds have been proven for the execution times for work stealing [22, 95]. Our experiments also suggest that the variation in execution time is small in practice. In our experiments, the difference between the mean execution time and the $99^{th}$ percentile execution time is less than 5% most of the time and the variation between the mean and the maximum execution time is also small.

More importantly, the variation shown by work stealing is never worse than (and is generally better than) that shown by the deterministic scheduler used by OpenMP. This indicates that work-stealing schedulers show promise for use in real-time systems, especially soft real-time systems which can tolerate some deadline misses, since they can potentially provide much better resource utilization than centralized schedulers for parallel tasks.
**Discussion:** One might wonder whether a different centralized scheduler that builds on better synchronization primitives can outperform Cilk Plus’s work-stealing scheduler. Our experiments shown in Figures 6.3a, 6.3b and 6.5 indicate that the higher overhead of the centralized scheduler mostly comes from the larger number of synchronization operations on the centralized global queue compared to lower contention on the distributed local queues. Therefore, even if synchronization primitives of the centralized scheduler were further optimized to reduce overheads, it is still unlikely to negate the inherent scalability advantages of randomized work-stealing, especially with increasing number of cores and workload complexity.

### 6.5 RTWS Platform

In this section, we describe the design of the *RTWS* platform, which provides a work-stealing federated scheduling service for parallel real-time tasks. RTWS has several benefits: (1) It separates the goals of efficient parallel performance and rigorous real-time execution. This separation of concerns allows programmers to re-purpose existing parallel applications to be run with real-time semantics with minimal modifications. (2) It allows the use of existing parallel languages and runtime systems (not designed for real-time programs) to explore the degree of real-time performance one can achieve without implementing an entirely new parallel runtime system. Therefore, we were able to evaluate the performance of the centralized scheduler from OpenMP and the work stealing scheduler from Cilk Plus for real-time task sets. (3) While RTWS does not explicitly consider cache overheads, the scheduling policy has an inherent advantage with respect to cache locality, since parallel tasks are allocated dedicated cores and never migrate.
**Application Programming Interface (API):** The RTWS API makes it easy to convert existing parallel programs into real-time programs. Tasks are C or C++ programs that include a header file (`task.h`) and conform to a simple structure: instead of a `main` function, a `run` function is specified, which is periodically executed when a job of the task is invoked. In addition, a configuration file must be provided for the task set, specifying runtime parameters (including program name and arguments) and real-time parameters (including period, work and burdened critical-path length) for each task.

**Platform Structure and Operation:** RTWS separates the functionalities of parallel scheduling and real-time scheduling. We use two components to enforce these two functionalities, a *real-time scheduler (RT-scheduler)* and a *parallel dispatcher (PL-dispatcher)*.

Specifically, the RT-scheduler configures the real-time performance of a task. Prior to execution, it reads tasks’ real-time parameters from the configuration file and calculates a core assignment, which has incorporated work stealing overheads into federated scheduling using a method developed by Li et al. [77]. The main function (provided by RTWS) binds each task to its assigned cores (by changing the CPU affinity mask). This core assignment ensures that each task has sufficient number of dedicated cores to meet most of its deadline during execution. Moreover, because each parallel task is executed on dedicated cores and no other tasks can introduce CPU interference with it, the PL-dispatcher does not need to be deadline- or priority-aware.

During execution, the PL-dispatcher enforces the periodic invocation of each task and calls an individual GNU Cilk Plus (or OpenMP) runtime system to provide parallel execution of each task. Since there are multiple concurrent parallel runtime systems that are unaware of each other, we need to entirely isolate them from each other to minimize scheduling overheads and CPU interference. Therefore, for Cilk Plus we modified its runtime system, so that each
Cilk Plus runtime only creates \( n_i \) workers, each of which is pinned to one of the \( n_i \) assigned cores. Similarly, for OpenMP we use static thread management and create exactly \( n_i \) threads to each task. In other words, there is only one worker thread per core and hence the worker assignment by PL-dispatcher is consistent with the core assignment of the RT-scheduler.

**Profiling Tool:** Since the work and critical-path length of each task must be specified to the platform (in the configuration file), we also provide a simple profiling utility to automatically measure these quantities for each task. The work of a task can be measured by running the profiling program on a single core. Measuring the critical-path length is more difficult. We adopt a profiling tool Cilkprof [90], which can automatically measure the work and the burdened critical-path length of a single job. In particular, Cilkprof uses compiler instrumentation to gather the execution time of every call site (i.e., a node in the DAG) and calculate the critical-path length in nanoseconds. To be consistent with GNU Cilk Plus (and GNU OpenMP), we used a version of Cilkprof that instrumented the GCC compiler and incorporated the burdened DAG into the measurement. 7

**Discussion:** In addition to using the work-stealing scheduler of Cilk Plus in RTWS, the design of RTWS allows us to instantiate another federated scheduling service that uses the centralized scheduler of GNU OpenMP, which we call the Real-Time Centralized Greedy (RTCG) platform. As shown in Section 6.4, work stealing has better parallel performance than the centralized scheduler. Thus, RTWS using work stealing is a better candidate for parallel tasks with soft real-time constraints, as confirmed via the empirical comparison in Section 6.6. However, it may not be the best approach for other scenarios. First and foremost, the execution time of a parallel task using work stealing can be as slow as its sequential execution time in the worst case, even though the probability of the worst case happening

---

7Intel provides another tool named Cilkview [61] that can measure work and burdened critical-path length, though in the number of instructions instead of amount of time, using dynamic binary instrumentation.
can be extremely low in practice. Therefore, it cannot be applied to hard real-time systems without modifying the work stealing protocol to provide some form of progress guarantee. In addition, for special purpose systems where the structure of parallel task is static and well measured, a static scheduler that decides how to execute the parallel task prior to execution can effectively reduce scheduling overheads and may perform better than work stealing.

6.6 Platform Evaluation

In this section, we evaluate the soft real-time performance provided by RTWS using a randomized work-stealing scheduler (RTWS) compared to the alternative implementation of federated scheduling using a centralized greedy scheduler (RTC). We use three DAG applications written in both Cilk Plus and OpenMP (discussed in Section 6.4) to randomly generate task sets for empirical experiments. To the best of our knowledge, the overarching RTWS framework is the first real-time platform that supports general DAG tasks, such as these benchmark programs, atop multiple widely used runtimes such as OpenMP and Cilk Plus. Since other existing real-time systems do not support parallel DAG tasks, we do not compare against them.

Experiments were conducted on a 32-core machine composed of four Intel Xeon processors (each with 8 cores). When running experiments, we reserved two cores for operating system services, leaving 30 experimental cores. Linux with CONFIG_PREEMPT_RT patch version r14 applied was used as the underlying RTOS.

6.6.1 Benchmark Task Sets Generation

We now describe how we generate task sets composed of the three benchmark programs (Cholesky, Heat and LU) with general DAG structures. We generate 4 sets of task sets and
evaluate their performances. The first 3 sets are composed with tasks running the same application, denoted as *Cholesky, Head* and *LU task sets*. The last set comprises a mix of all benchmarks, denoted as *Mixed task sets*.

We profile Cholesky, Heat and LU programs using 14, 6 and 3 different input sizes, respectively. For each program with each input size, we measure its work and burdened critical-path length using Cilkprof. Then we generate different tasks (from one benchmark with one input size) and assign it a randomly generated utilization. To see the effect of scalability of large parallel tasks (i.e., spanning many cores), we intentionally create 5 types of tasks: tasks with mean utilization selected from \{1, 3, 6, 12, 15\}. When assigning utilization to a task, we always try to pick the largest mean utilization that does not make the task set utilization exceed the total utilization that we desire. After deciding a mean utilization, we then randomly generate the utilization of the task using the mean value. A task’s period is calculated using its work over its utilization. We keep adding tasks into the task set, until it reaches the desired total utilization. For each setting, we randomly generate 10 task sets.

6.6.2 Evaluation Results

For each DAG task set, we record the *deadline miss ratio*, which is calculated using the total number of deadline misses divided by the total number of jobs in the task set. We also record the response time of each individual job during its execution to calculate a *relative response time*, which is the job’s response time over its deadline. We then calculate the average relative response time for each task set.

In the first two comparisons between RTWS and RTCG, we’d like to see how the integration of federated scheduling and randomized work stealing performs compared with federated scheduling using a centralized greedy scheduler *given the same resource capacity* for soft
real-time task sets. Therefore, for these experiments we use the *same core assignment* as described by Li et al. [77], which incorporates work stealing overheads into federated scheduling.

Since the centralized scheduler generally has larger overheads and takes longer to execute as shown in Section 6.4, it is not surprising to see that RTCG performs worse than RTWS given the same resource capacity. To further analyze the performance difference between the two approaches, in the last experiment we increase the resource capacity for RTCG. We would like to see how much more resource capacity RTCG requires in order to *schedule the same task sets* compared with RTWS.

**Deadline miss ratio comparison:** We first compare the deadline miss ratio in Figure 6.6a,6.6b,6.6c and 6.6d for Cholesky, Heat, LU and Mixed task sets, respectively. Notably, most of the task sets under RTWS has no deadline misses and all of the task sets have a deadline miss ratio no more than 10%. In fact, from all the experiments we run, there are only 2.25% tasks (28 out of 1243 tasks) having deadline misses. In contrast, given the same core assignment RTCG misses substantially more deadlines, especially for Heat task sets where many tasks miss all of their deadlines.

**Relative response time comparison:** In Figure 6.7a,6.7b,6.7c and 6.7d, we observe that RTCG has much higher average relative response time than RTWS, given the same resource capacity. For all task sets, the average relative response time of RTWS is less than 1, while some tasks under RTCG even have relative response times larger than a hundred. In order to clearly see the relative response times smaller than 1, when plotting the figures we mark all the relative response times that are larger than 3 as 3.

**Required resource capacity:** From the first two comparisons, we can clearly see that RTCG requires more cores (i.e., resource capacity) in order to provide the same real-time
performance as RTWS. Thus, in Figure 6.8a,6.8b,6.8c and 6.8d we keep increasing the number of cores assigned to tasks under RTCG that have more than 25% of deadline misses. Note that all tasks under RTWS meet at least 80% of deadlines. We compare the required number of cores of RTCG and RTWS for the same task sets to meet most of their deadlines. If a task set misses most deadlines when allocated all 30 available cores, then we mark the number of required cores as 34. For Cholesky and LU task sets, RTCG requires about 1 to 3 additional cores. For some Heat task sets, even doubling the number of cores for RTCG is still not sufficient.
Figure 6.6: Deadline miss ratio of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS (providing federated scheduling service integrated with a randomized work-stealing scheduler in GNU Cilk Plus) and RTCG (providing federated scheduling service integrated with a centralized greedy scheduler in GNU OpenMP). RTWS and RTCG use the same core assignment.
Figure 6.7: Average relative response time of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS (providing federated scheduling service integrated with a randomized work-stealing scheduler in GNU Cilk Plus) and RTCG (providing federated scheduling service integrated with a centralized greedy scheduler in GNU OpenMP). RTWS and RTCG use the same core assignment.
Figure 6.8: Required number of cores of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS (providing federated scheduling service integrated with a randomized work-stealing scheduler in GNU Cilk Plus) and RTCG (providing federated scheduling service integrated with a centralized greedy scheduler in GNU OpenMP). We increase the number of cores for each task under RTCG until it misses no more than 60% of deadlines.
6.7 Summary

In this chapter, we have presented RTWS, the first parallel scheduling system for soft real-time tasks with general DAG structures that can target multiple widely used runtimes such as OpenMP and Cilk Plus. RTWS adapts and integrates federated scheduling with work stealing. Integrated with the widely used Cilk Plus concurrency platform, RTWS can schedule standard Cilk Plus programs. Furthermore, RTWS does not require detailed knowledge of task structure. Instead, it only use coarse-grained task parameters that can be easily measured using existing Cilk Plus tools. Experimental results demonstrated that RTWS can considerably improve the response time of tasks on a given number of cores. Especially when running real benchmark programs, it significantly reduces the required resources for task set schedulability. Therefore it represents a promising step towards practical development and deployment of real-time applications based on existing programming languages, platform and tools.
Chapter 7

Conclusion

As the density of electronic components in a single processing unit reaches saturation, multiprocessors are increasingly becoming the norm. Multiprocessors are being deployed in many computing systems nowadays, including real-time computing systems. Examples of such systems range from standalone systems like real-time robotic controls and automotive applications to connected systems like real-time clouds. At the same time, real-time applications are becoming more computationally demanding: for instance computer vision and artificial intelligence applications in autonomous vehicles can be large and complex, while their temporal requirements often remain unchanged since those requirements are still dictated by the same physical world. Consequently, these two factors require real-time applications to be parallelized to exploit the parallelism provided by multiprocessors, in order to satisfy their temporal requirements.

Scheduling real-time parallel applications on multiprocessors, however, is more challenging compared to scheduling sequential applications. This is because parallel tasks introduce another dimension of complexity in analyzing schedulability for real-time systems — intra-task
parallelism is allowed in addition to inter-task parallelism. In this dissertation, we have addressed the problem of scheduling real-time parallel tasks that can be independent or can access shared resources. In particular, for independent parallel tasks, we presented a novel deterministic algorithm for scheduling parallel tasks represented by directed acyclic graphs (Chapter 3). The proposed algorithm schedules each parallel task efficiently on its exclusive set of processors, thus reducing the number of processors required by the task. We show that this algorithm when combined with federated scheduling approach outperforms other state-of-the-art federated-based scheduling algorithms in term of task sets’ schedulability.

Global scheduling is a widely known approach for scheduling parallel tasks in which threads of each task are allowed to execute simultaneously and migrate between processors during execution. Analyzing schedulability for global scheduling is, however, more complex than that of sequential tasks due to intra-task parallelism. This dissertation presents an analysis of global fixed-priority scheduling which assigns each parallel task a fixed priority — all jobs of each task have the same priority (Chapter 4). The analysis is shown to have better performance than previously proposed analyses for the same scheduling assumption.

For parallel tasks that access shared non-processor resources such as in-memory buffers or data structures, we presented a new analysis based on federated scheduling (Chapter 5). In this task model, each shared resource is protected by a mutual exclusive spin lock. We consider two commonly used orderings for the spin locks: FIFO-ordered and priority-ordered, and analyze worst-case blocking terms for both types. Our experiments show that priority-ordered spin locks provide better performance than FIFO-ordered spin locks in terms of schedulability of task sets.

Finally, this dissertation discusses two different strategies for implementing parallel schedulers: centralized scheduling and randomized work stealing (Chapter 6). These two strategies are
compared using synthetic and real-world benchmarks to evaluate their suitability for soft real-time systems. Experimental results suggest that randomized work stealing combined with federated scheduling provides a scalable platform that can support large scale soft real-time applications.

This dissertation gives rise to several open problems that we would like to address in the future.

**Open Problems:** Firstly, partitioned scheduling, even though it may not be applied to parallel tasks directly, is worth investigating. Typically, to apply partitioned scheduling for parallel tasks, we must first assign nodes of each task to processors. Then an existing scheduling algorithm and analysis for sequential tasks can be applied to the nodes on individual processors with respect to the precedence constraint of the original parallel task. These two smaller problems may affect each other, and thus it may be beneficial to study them holistically.

Secondly, though all contributions in this dissertation assume preemptive tasks, non-preemptive scheduling is a reasonable option for which not much work has been invested for parallel tasks. Non-preemptive scheduling has some advantages over preemptive scheduling: (i) non-preemptive scheduling tends to be more deterministic, thus may be easier to analyze and less pessimistic analytically than preemptive scheduling, and (ii) since preemptions are abandoned, non-preemptive scheduling likely has lower overheads than preemptive scheduling in practice, thus can be a practical solution.

Thirdly, for parallel tasks that share non-processor resources, we currently assume that there is no nested request and there are only heavy tasks in the task sets. For future work, we would like to lift these limitations — heavy tasks and light tasks can access the same set of non-processor resources, and nested requests are allowed. Furthermore, different types of
locks, including semaphores, reader-writer locks and k-exclusion locks, are worth studying. Lastly, the resource sharing problem for parallel tasks should also be extended to other scheduling approaches such as global scheduling.
References


161


[100] Shuai Yuan, Jun Wang, and Xiaoxue Zhao. “Real-time bidding for online advertising: measurement and analysis”. In: *Proceedings of the Seventh International Workshop on Data Mining for Online Advertising*. ACM. 2013, p. 3.