SCHEDULING REAL-TIME GRAPH-BASED WORKLOADS

Sergey Voronov

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

Chapel Hill
2023

Approved by:

James H. Anderson

Nan Guan

Parasara Sridhar Duggirala

Samarjit Chakraborty

Shahriar Nirjon

ABSTRACT

Sergey Voronov: Scheduling Real-Time Graph-Based Workloads
(Under the direction of James H. Anderson)

Developments in the semiconductor industry in the previous decades have made possible computing platforms with very large computing capacities that, in turn, have stimulated the rapid progress of computationally intensive computer vision (CV) algorithms with highly parallelizable structure (often represented as graphs). Applications using such algorithms are the foundation for the transformation of semi-autonomous systems (*e.g.,* advanced driver-assist systems) to future fully-autonomous systems (*e.g.,* self-driving cars). Enabling mass-produced safety-critical systems with full autonomy requires real-time execution guarantees as a part of system certification.

Since multiple CV applications may need to share the same hardware platform due to size, weight, power, and cost constraints, system component isolation is necessary to avoid explosive interference growth that breaks all execution guarantees. Existing software certification processes achieve component isolation through time partitioning, which can be broken by accelerator usage, which is essential for high-efficacy CV algorithms.

The goal of this dissertation is to make a first step towards providing real-time guarantees for safety-critical systems by analyzing the scheduling of highly parallel accelerator-using workloads isolated in system components. The specific contributions are threefold.

First, a general method for graph-based workloads' response-time-bound reduction through graph structure modifications is introduced, leading to significant response-time-bound reductions. Second, a generalized real-time task model is introduced that enables real-time response-time bounds for a wider range of graph-based workloads. A proposed response-time analysis for the introduced model accounts for potential accelerator usage within tasks. Third, a scheduling approach for graph-based workloads in a single system component is proposed that ensures the temporal isolation of system components. A response-time analysis for workloads with accelerator usage is presented alongside a non-mandatory schedulability-improvement step. This approach can help to enable component-wise certification in the considered systems.

ACKNOWLEDGEMENTS

My dissertation would not have been possible without the help of many people. First and foremost, I want to thank my advisor, Jim Anderson. His great passion for research, intuition for identifying interesting problems, and problem-solving skills have always inspired me. His invaluable support and guidance in my own research has helped me to solve many problems. His endless support and advice has fundamentally altered my approach towards technical writing. I cannot imagine a better advisor. I would also like to thank my dissertation committee: Nan Guan, Samarjit Chakraborty, Parasara Sridhar Duggirala, and Shahriar Nirjon, for all of their feedback.

I am extremely thankful for my co-authors and collaborators: Tanya Amert, Stephen Tang, Peter Tong, Kecheng Yang, Joshua Bakita, F. Donelson Smith, Ming Yang, Saujas Nandi, and Thanh Vu. Your efforts and advice were extremely helpful; I learned so much from you all. In particular, I'd like to highlight Stephen and Tanya. Their impact stands out even among the extraordinary people mentioned above.

I have enjoyed meeting the current and former members of the Real-Time Systems Group, including Nathan Otterness, Shareef Ahmed, Sims Osborne, Catherine Nemitz, Namhoon Kim, Sanjoy Baruah, Tyler Yandrofski, Clara Hobbs, Hennady Leontiev, Sarah Rust, and Abhishek Singh. I am very grateful to the members of the Computer Vision and Cyber Physical Systems Groups, including but not limited to Angelos Angelopoulos, Ron Alterovitz, Parasara Sridhar Duggirala, Shahriar Nirjon, and Don Porter.

I express my gratitude to the current and former Computer Science Department staff, who keep everything running as smoothly as possible, especially: Jodie Gregoritsch, Denise Kenney, Missy Wood, the late Bil Hays, Alicia Holtz, Jim Mahaney, and Tatyana Devis. I would also like to thank all of the faculty who taught me, especially Montek Singh and Jan Prins.

I would like to thank the people that participated in my internship experience at Advanced Micro Devices and General Motors, especially Daniel Lowell and Shige Wang.

I am endlessly grateful for the support of my friends Stephen, Mikhail, Vsevolod, Konstantin, and Andrew. Also I want to thank the board games group (Tanya, Peter, Nic, Auston, and Qiuyu). However, most

of all I am thankful to my family, who patiently supported me and helped with any problems that appeared on the way.

The funding for this research was provided by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960; ARO grant W911NF-17-1-0294; ONR grant N00014-20-1-2698; and a grant from General Motors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
CV	Computer Vision
DAG	Directed Acyclic Graph
EDF	Earliest-Deadline-First
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FP	Fixed-Priority
GPU	Graphics Processing Unit
G-EDF	Global Earliest-Deadline-First
HAC	Hardware Accelerator
HRT	Hard Real-Time
ML	Machine Learning
NP-EDF	Non-Preemptive Earliest-Deadline-First
NP	Nondeterministic Polynomial-Time
OMLP	$O(m)$ Locking Protocol
PCR	Periodic Component Reservation
pi-blocking	Priority-inversion Blocking
SAT	Boolean Satisfiability Problem
SRT	Soft Real-Time
WCET	Worst-Case Execution Time

CHAPTER 1: INTRODUCTION

Continuous progress in the semiconductor industry over the last 20 years has significantly increased modern chips' computing capacity. For example, a typical consumer central processing unit (CPU) from 20 years ago could complete around 10^9 floating point operations per second (FLOPS). In contrast, the more modern AMD 5900X CPU achieves $9 \cdot 10^{11}$ FLOPS, and the NVIDIA RTX 3090 GPU provides $3.6 \cdot 10^{13}$ FLOPS. Such progress enabled extensive research of compute-heavy algorithms and enabled their mass usage in the real world. One of the most promising areas of such research is machine learning (ML), including computer vision (CV) algorithms. Advanced computing capacities allowed researchers to significantly improve the quality of solutions to many notable CV problems (*e.g.,* image classification or video object detection). Examples of new algorithms include AlexNet (Krizhevsky et al., 2012), ResNet (He et al., 2016), and YOLO (Bochkovskiy et al., 2020). However, the improved efficacy came at the cost of significant resource consumption (*e.g.,* CoCa (Yu et al., 2022) has 2 billion parameters). Fortunately, new computing platforms were created with acceptable size, weight, and power (SWaP) due to the aforementioned semiconductor progress; these platforms can be acceptable for embedded usage (*e.g.,* the NVIDIA Xavier (Ditty et al., 2018)).

These algorithms and computing platforms have provided a foundation for the rapid development of semi-autonomous driver assist systems (*e.g.,* Tesla Autopilot, Mercedes Intelligent Drive, BMW Driving Assistant, *etc.*). Currently, these systems partially rely on the driver in the case of an emergency and cannot achieve full autonomy (level 4 or 5 by the classification (SAE, 2018)). Two significant challenges should need to be addressed to enable full autonomy. The first challenge is to improve the algorithmic efficacy (*e.g.,* Tesla Autopilot sometimes classifies objects incorrectly). The second challenge is to provide system execution guarantees: all system parts should finish within time intervals sufficient to ensure safe operation. In this dissertation, we consider the second problem and provide timing guarantees for such systems.

Many CV (and ML) algorithms are designed to maximize system performance (using average execution time) and/or result quality (via classification quality). Unfortunately, both of these characteristics do not guarantee the completion of such algorithms within the prescribed time intervals in the worst case. One way

to provide these guarantees is to perform excessive testing. Unfortunately, such testing has to be repeated for each software update even if unrelated software is updated (in addition to being very time and money consuming). At the same time, the mass adoption of fully autonomous cars requires their systems to be certified so that real-time execution guarantees can be provided.

The timing analysis of the aforementioned systems should consider three issues: the parallel nature of the CV/ML workloads; the heavy consumption of computing resources, which motivates using hardware accelerators; and the isolation of various components from each other. We explain these issues and their necessity in detail later in Section 1.2.

This dissertation aims to provide a response-time analysis for such systems. Such analysis provides an upper bound on the time it take to finish each workload execution. The remainder of this chapter is organized as follows. First, we discuss such systems in general. Next, we explore three essential aspects of the considered real-time systems. Then, we present the dissertation thesis. Finally, we outline the remaining chapters in this dissertation.

1.1 Real-Time Systems

The example of a self-driving car discussed above shows that some systems have a dual notion of correctness, specifically *logical* and *temporal* correctness. Broadly speaking, a logically correct system produces correct results at some point, and a temporally correct system produces results within a required time interval. While logical correctness is required for any working system (including non-real-time ones), we consider the second aspect. Systems that require temporal correctness are called *real-time systems*.

Temporal correctness is typically verified via a two-step approach. The first step involves modeling the system formally by specifying a task and hardware model. The second step involves analyzing the obtained model to ensure all timing constraints are met. Note that both steps are performed during the system design phase. We follow the same pattern.

There are multiple real-time system models widely accepted in the research community. In order to find/develop a correct model, we consider several important aspects of the considered real-time systems that are our focus. Later we analyze the chosen model by abstracting from characteristics of real systems.

1.2 Considered Aspects of Real-Time Systems

As mentioned before, we seek to provide real-time analysis for safety-critical systems with compute-heavy CV applications on board. Notable examples of such systems include self-driving cars and unmanned aerial vehicles (*e.g.,* drones). These examples are definitely safety-critical, as a temporal correctness violation in these systems may cause a very serious accident (*e.g.,* injury or death). Progress in these areas is impacted by certification, as widely accepted safety-critical systems must be certified to be safe. At the same time, no such certification exists at the moment. Onboard usage of compute-heavy CV applications and necessary certification requirements motivate three non-trivial system aspects we need to consider to build a correct system model. We now describe them in detail.

Isolation of system components. We start with a discussion about the initial system design. We motivate it to be component-based for three major reasons.

Firstly, a safety-critical real-time system certification requires worst-case response times to be bounded and certified. These worst-case response times typically grow with the size of the competing workload (by the definition of "worst-case"). As mentioned before, the systems considered here became an object of interest primarily due to the improved algorithmic efficacy. Unfortunately, most novel algorithms require large computations and solve multiple relatively independent problems (*e.g.,* vehicle control and object detection); their execution on the same computing platform significantly increases the potential for interference (the execution time of a task of interest may increase significantly due to the competitive usage of shared system components—like a system bus). Thus, a good system design aims to reduce the number of interference channels. Certification standards for uniprocessor systems reduce interference by splitting a system into components and isolating them from each other (*e.g.,* ARINC 653 (Prisaznuk, 2008) in the safety-critical avionics domain).

Secondly, motivation for enabling component-based design comes from the software development process. The complexity of a software system can grow over time, and the size of modern systems requires a large developer team, where different software parts are developed by different groups. The division into groups means developers will need to consider inter-component interference. Thus, the certification process must consider all possible interference scenarios, the scale of which grows exponentially with the number of components. Unfortunately, static analysis tools cannot consider all interference scenarios, and testing of all such scenarios may require an unacceptable amount of time (*e.g.,* years).

Thirdly, CV/ML algorithms tend to change rapidly over time (*e.g.,* as evidenced by the last few years' progress in deep learning research). This rapid progress in ML research requires any manufacturer to use newer algorithms (to get better results). Since even a small algorithm change requires retesting the whole system, the only scalable way to certify such a system is to isolate system components from each other (to reduce interference sources among various components significantly).

Hardware accelerator usage. The second aspect of the considered systems is the necessity of hardware accelerators. An autonomous driver assist system should be able to detect objects (*e.g.,* pedestrians), recognize objects (*e.g.,* traffic signs), and perform trajectory detection (*e.g.,* other cars) as a part of a normal working mode. As mentioned before, the intensive research and development (*e.g.,* Waymo, Tesla, GM, Mercedes, Cruise, Volkswagen, *etc.*) of self-driving cars was motivated by the discovery of new CV/ML algorithms with good efficacy. Unfortunately, these algorithms require heavy computing capacities, and requirements grow over time. For example, ResNet-152 (the best ImageNet classifier as of 2016 with 79% accuracy) requires $1.1 \cdot 10^{10}$ FLOPS to process one frame per second (He et al., 2016), and CoAtNet-7 (the best ImageNet classifier of 2021 with 91% accuracy) requires $2.5 \cdot 10^{12}$ FLOPS (Dai et al., 2021). Modern CPUs cannot provide the required capacity for such algorithms to be responsive (e.g., to process 20 or 30 frames per second alongside other workloads). A typical modern desktop CPU (AMD 7950X) provides around 10^{12} FLOPS, which is not enough. Thus, hardware accelerators have to be used (*e.g.,* graphical processing units, GPUs). The internal structure and scheduling properties of these accelerators differ from CPUs (*e.g.,* accelerators are typically non-preemptive), so it should be reflected in the system model.

Task self-parallelization. While some workloads of the considered systems may be offloaded and accelerated with hardware accelerators, some tasks are not suited for it (*e.g.,* tasks with long sequential segments or heavy data usage) and need to be scheduled on the CPU. Unfortunately, the speed of a single CPU execution thread is limited by physical constraints, so these applications need to use several CPU cores simultaneously. The natural model of a parallel application with multiple internal data dependencies is a graph whose nodes represent sequentially executed code sequences, and edges represent data dependencies between various pieces in the form of various precedence constraints.

An example of a graph-based model in a safety-critical application can be found in Autoware (Kato et al., 2018) (see Figure 1.1). Another example of the model can be found in OpenVX (The Khronos Group, 2023), an open royalty-free standard for cross-platform acceleration of CV applications.

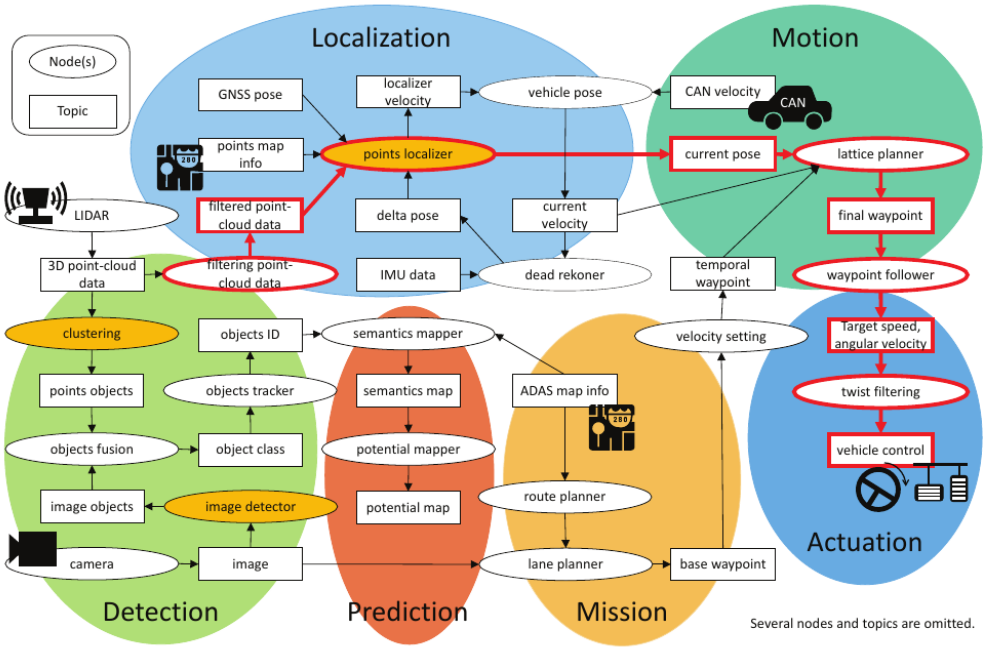


Figure 1.1: A package configuration diagram of Autoware ((Kato et al., 2018, p. 7, Figure 5)).

1.3 Thesis Statement

The three aspects discussed above (component isolation, accelerator usage, and task self-parallelism) should be considered in modeling a real-time system with graph-based applications and analyzing its properties as part of its certification. In this dissertation, we assume a multicore+accelerator platform and propose the relevant analysis, as summarized in the following thesis statement:

> Time-isolation guarantees can be provided for multi-component systems with graph-based applications in the presence of non-preemptive accelerator accesses; various scheduling choices can significantly improve the schedulability of such systems.

1.4 Contributions

The above thesis is supported by the following contributions.

1.4.1 Generalized Parallel Task Model

In Chapter 2, we introduce the rp-sporadic task model (stands for restricted parallelism). While the standard real-time model assumes each task is sequential (discussed in Chapter 2), our model considers

an additional per-task parameter, specifying the number of invocations of the same task that may execute concurrently. As we show later, such parallelism improves system schedulability, preserving its data dependencies. In Chapter 3, we provide a response-time analysis of this task model under standard conditions assuming partial non-preemtivity of tasks.

1.4.2 Graph Response-Time Bound Reduction Technique

As we show in Chapter 4, the existing real-time analysis of graph-based tasks may produce unacceptable bounds in some cases. We propose a novel method of node merging to reduce these bounds and evaluate the proposed approach.

1.4.3 Component Isolation

Component-based design requires some notion of component isolation. In Chapter 5, we discuss an isolation-preserving approach assuming non-preemptive hardware accelerator requests. In addition, we provide real-time analysis of a single isolated component under the partial availability of the computing platform.

1.5 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we provide a general background and review related prior work. Focusing on the rp-sporadic task model alone, we provide a response-time analysis of the model in Chapter 3. In Chapter 4, we consider graph-based response-time analysis and bound reduction techniques. In Chapter 5, we consider a single isolated component with hardware accelerator(s) usage. Finally, we conclude in Chapter 6.

CHAPTER 2: BACKGROUND

As discussed in Section 1.1, the first step in verifying temporal correctness is system modeling. In this chapter, we describe several existing workload models, examine hardware models, and survey prior real-time research. We first discuss the basic task model in Section 2.1, the system model in Section 2.2, and various sequential task model extensions in Section 2.3. We then discuss parallel task models in Section 2.4 and graph scheduling approaches in Section 2.5. Finally, we extend the basic system model in Section 2.6, review relevant prior work, and discuss our contributions in Section 2.7.

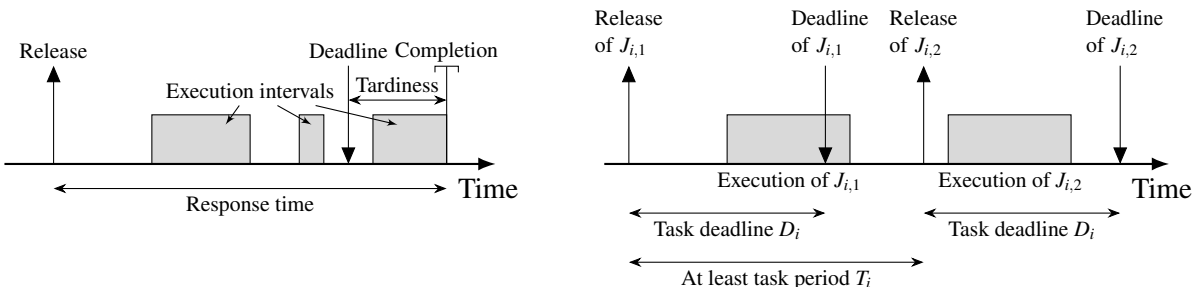
2.1 Task Model

In many systems, the overall workload can be split into the repeated execution of sequential pieces of code (different invocations may use different inputs). This behavior was captured in the seminal work of Liu and Layland (Liu and Layland, 1973): repeating computations are grouped into *tasks*, and tasks have timing constraints in the form of *deadlines*.

More formally, a sporadic task τ_i releases a (potentially infinite) sequence of jobs $J_{i,0}, J_{i,1}, \ldots$. An example of a single job is shown in Figure 2.1a. Each task τ_i is characterized by three parameters (C_i, T_i, D_i): its *worst-case execution time* (WCET) C_i is an upper bound on the execution time of any job of τ_i; its *period* T_i is the minimal inter-arrival time of jobs of τ_i; and its *relative deadline* D_i is the amount of time for each job of τ_i to complete execution. The *utilization* of τ_i is defined as $u_i = \frac{C_i}{T_i}$. An example of a sporadic task is shown in Figure 2.1b.

For a job $J_{i,j}$ of τ_i with *release* time $r_{i,j}$, its *absolute deadline* is defined as $r_{i,j} + D_i$, and its *completion* (or *finish*) time is denoted $f_{i,j}$. The *response time* of $J_{i,j}$ is defined as $f_{i,j} - r_{i,j}$, *i.e.,* the time required to complete it. If R_i upper bounds the response time of any job of τ_i, then it is called called a *response-time bound* of τ_i. The *tardiness* of $J_{i,j}$ is defined $\max(f_{i,j} - d_{i,j}, 0)$. If a job has positive tardiness, then its deadline was missed. Similarly, a task's tardiness is an upper bound on the tardiness of any of its jobs.

A job J is *pending* if it is released but has not completed yet, and is *ready*, if it is pending and all jobs it depends on are completed. Under the sporadic task model, J depends on all previous jobs of the same task.



(a) A single job of a task τ_i.

(b) Several jobs of τ_i.

Figure 2.1: A sporadic task example.

2.2 System Model

In the previous section, we defined the basic task model. Now we are able to describe the remaining aspects of the system model: the considered computing platform in Section 2.2.1 (where to schedule), the considered system correctness criteria in Section 2.2.2 (how to check a schedule's correctness), and the considered scheduling algorithm in Section 2.2.3 (how to schedule).

2.2.1 Platform Model

We schedule a job on a CPU platform π with m identical cores $\pi_1, \pi_2, ..., \pi_m$ (*identical multiprocessor* platform). For simplicity, we call each core a CPU throughout the rest of the dissertation. Each of m CPUs can execute at most one job at a time and has a speed of one (a job completes one unit of execution in one unit of time). If a CPU is executing any job at time t, then the CPU is called *busy* at t; otherwise the CPU is called *free* at t.

A job is called *preemptive* if a scheduling algorithm (defined in Section 2.2.3) is able to stop the execution of the job on the CPU at any time and resume it later from the same position (possibly on a different CPU); such an action is called a *job preemption*. We assume that the preemption cost of a preemptive job is negligible compared to its execution time; equivalently, a job's execution cost can be assumed to be inflated to account for the costs of any preemptions of it, so the latter costs can be safely ignored. A job is called *non-preemptive* if its execution cannot be preempted at all.

The definitions of preemptive and non-preemptive jobs may need to be relaxed for use with real applications. We consider later jobs that may contain preemptive (*e.g.,* normal CPU execution) and non-

preemptive (*e.g.,* using a non-preemptive accelerator) execution intervals. We also consider hardware accelerators (described in detail in Section 2.6).

2.2.2 Temporal Correctness Models

Real-world real-time systems may have different deadline violation costs. For example, the violation cost of a deadline in an autonomous aircraft is exceptionally high, while video conferencing can tolerate several missed frames (however, deadline violations are not desirable). Thus, temporal correctness can be defined in two ways: *hard* and *soft* real-time (HRT and SRT). Thus, if the violation cost is extremely high, then all jobs must meet their deadlines; the system is called an HRT system. If, in contrast, the system may sustain some deadline violations (task tardiness is upper-bounded by a constant value), the system is called an SRT system. We describe these concepts formally in the following definition.

Definition 2.1. A real-time system is called HRT-schedulable (SRT-schedulable, respectively) under a scheduling algorithm if and only if the tardiness of every job in that system is guaranteed to be zero (upper-bounded by a constant, respectively) under that scheduling algorithm.

A task's tardiness is bounded if and only if its response time is bounded, so Definition 2.1 can be rephrased using response-time bounds.

In a system with properly isolated components, each component may have its own temporal correctness definition (*e.g.,* some components may be HRT, while others are SRT). A survey of industry practices (Akesson et al., 2020, Q. 13) showed that SRT deadlines are often present in existing systems (67% of responders faced it). Typically, an SRT analysis provides weaker guarantees for a larger set of considered workloads than an HRT analysis of the same system. For example, the same set of tasks may be non-HRT-schedulable but SRT-schedulable on the same computing platform under the same scheduling algorithm.

In this dissertation, we focus on SRT systems. Additional temporal-correctness definitions exist (*e.g., firm* real-time (Hamdaoui and Ramanathan, 1995)) that are omitted here. Moreover, there are other ways to define SRT schedulability (*e.g.,* based on deadline-miss probabilities (Fontanelli et al., 2013)). These approaches are out of the focus for this dissertation; we focus on proving response-time / tardiness bounds for SRT task sets (as defined in Definition 2.1).

2.2.3 System Scheduler

In Sections 2.1 and 2.2.1, we described the platform and the basic task model. In this section, we describe the remaining part: the *scheduler*. A scheduler is an algorithm that defines the actual job schedule on the considered platform during run time. In this section, we describe general scheduling approaches and specify our choice of scheduler. Historically, the earliest real-time research considered *table-driven* (*static cyclic*) schedulers (according to (Audsley et al., 1995)). Such schedulers construct a fixed-length schedule cycle before the actual system runs; the final schedule is obtained by cyclicly repeating these. Because the schedule is deterministic, it provides ease of analysis, but somewhat inflexible behavior. Cyclic schedulers are still used in real systems, often as part of more complicated scheduling algorithms (Akesson et al., 2022, Q.20). Several schedulers have been proposed to improve scheduling flexibility.

In this section, we consider only "greedy" schedulers: if a ready job is not scheduled, then all CPUs are busy. Such schedulers are called *work-conserving*. In fact, every work-conserving scheduler A is defined by addressing two issues: job prioritization and job scheduling. At any given moment, A first assigns priorities to all ready jobs in the system. Second, A schedules m jobs with the highest priority (all ready jobs if less than m jobs are ready). Thus, the schedule is constructed by prioritizing higher-priority jobs over lower-priority jobs. We group all work-conserving schedulers into three broad scheduling classes based on the job prioritization algorithm.

The first class assumes a constant task-level priority assignment; such schedulers are called *fixed-priority* (FP) schedulers. Schedulers of this class define a constant priority for each *task* independently; jobs share the priorities of their tasks. The priorities of tasks stay the same over time. Notable examples include the Rate-Monotonic (Liu and Layland, 1973) and the Deadline-Monotonic (Leung and Whitehead, 1982) schedulers.

The second class assumes a constant job-level priority assignment. Schedulers of this class define a constant priority for each *job* independently. The priorities of jobs do not change over time. Notable examples include the Earliest-Deadline-First (Serlin, 1972; Labetoulle, 1974) scheduler (EDF).

The final scheduling class allows job priorities to change over time. Schedulers of this class may change job priorities on the fly. Notable examples include P-Fair (Baruah et al., 1993; Holman and Anderson, 2005; Srinivasan and Anderson, 2006).

Two attributes can be used to compare these classes: how many task sets they can schedule (schedulability) and how easy they are to implement (applicability). The first class of schedulers provides the lowest schedulability and highest applicability; the second class provides intermediate schedulability and applicability; and the third class provides the highest schedulability and lowest applicability. Note that schedulers of the third class may change job priorities frequently, so overheads are expected to be significantly higher compared to the schedulers of the first and second classes.

In this dissertation, we focus on EDF, a widely studied example of a fixed job-level priority scheduler (the most balanced class among those discussed). EDF prioritizes ready jobs by their absolute deadlines. It can be formally defined as follows.

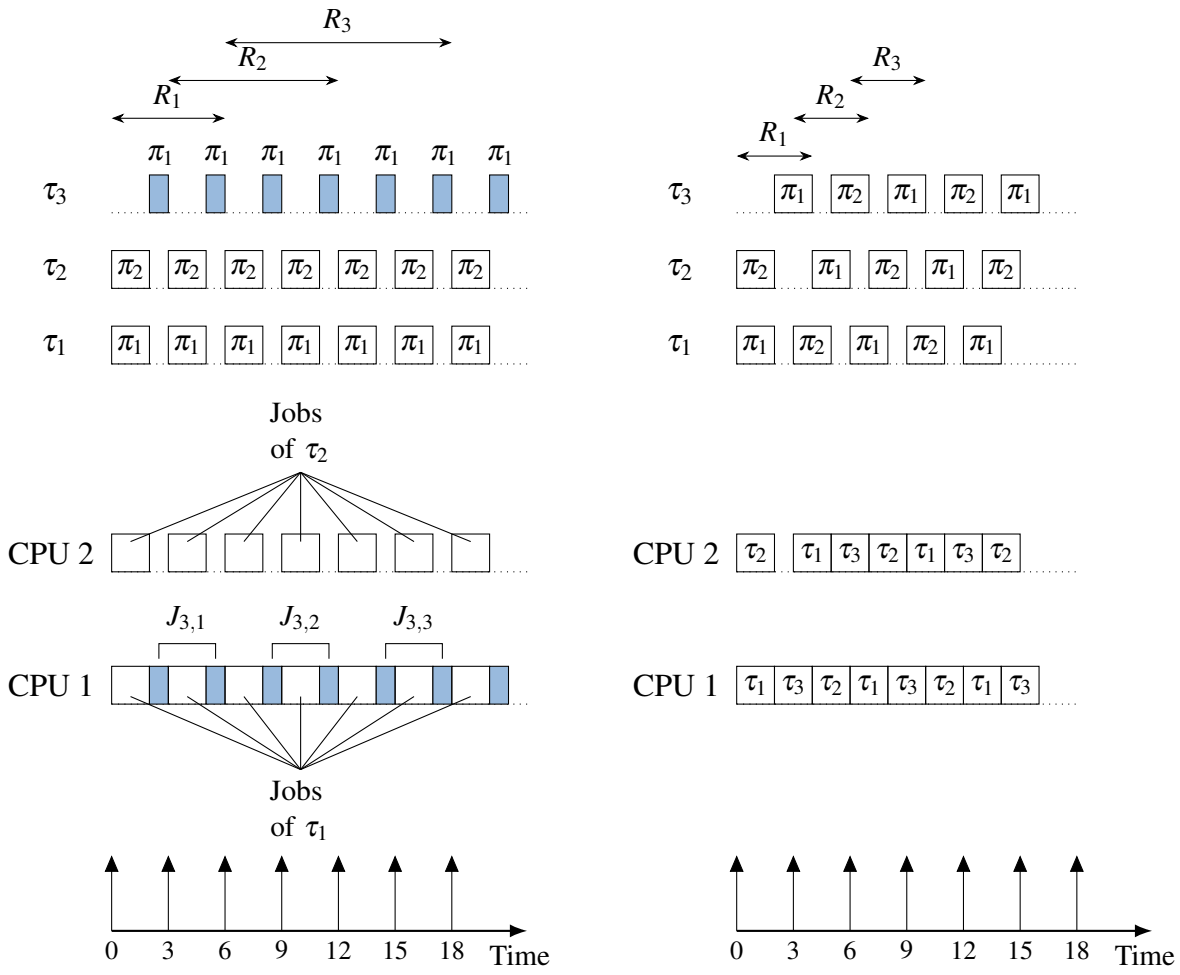
Definition 2.2. If at least m jobs are ready, m ready jobs with the earliest deadlines are scheduled; otherwise all ready jobs are scheduled (and some CPU(s) are free). Deadline ties are broken arbitrarily, but consistently with respect to task identifiers.

The EDF scheduler is known to provide the best possible schedulability with SRT task sets under various task and platform models (*i.e.,* EDF can schedule any schedulable task set). Examples include both preemptive and non-preemptive EDF on identical platforms (Devi and Anderson, 2008) (the same platform as in our dissertation), preemptive EDF on uniform computing platforms (Yang and Anderson, 2017), and preemptive EDF on identical mutiprocessor platforms with affinity masks (Tang et al., 2019). The following example illustrates the difference between FP and EDF SRT schedulers.

Example 2.1 (see Figure 2.2). Consider three periodic tasks with $C = 2$ and $T = 3$ scheduled on two CPUs (released at time instants 0, 3, 6, ...). The top and bottom parts of Figure 2.2 represent task-focused (which CPU schedules the current task) and processor-focused (which job is executed on the current CPU) views of the same schedule (as before π_i represents CPU i).

An FP schedule for the same task set is shown in Figure 2.2a assuming τ_1 has highest priority and τ_3 has lowest priority. As we can see, task τ_3 has unbounded response times (the response times of its first three jobs are illustrated with R_1, R_2, and R_3 with values of 6, 9, and 12). At the same time, CPU 2 is underutilized because no jobs of τ_3 can be scheduled in parallel.

An EDF schedule for these three tasks is shown in Figure 2.2b, assuming that jobs of τ_1 have higher priority in case of identical deadlines. Both CPUs are (almost) fully utilized, and the response time of each job of τ_3 is 4. $\Diamond$



(a) A FP schedule.

(b) An EDF schedule.

Figure 2.2: Schedules of three periodic tasks with $C = 2$ and $T = 3$ under FP and EDF schedulers.

This example shows the better SRT schedulability of the EDF scheduler compared to FP.

2.3 Sequential Task Model Extensions

The sporadic task model described in Section 2.1 is a simple model proposed 50 year ago by Liu and Layland (1973). Real-time researchers have since considered several extended models to support a broader range of real-time systems. We now describe them in detail.

2.3.1 Task Deadline Models

Many early real-time researchers considered the *implicit*-deadline model, which requires $D_i = T_i$ for each task τ_i. However, actual workloads may have deadlines different from periods. The implicit-deadline model

was generalized to support $D_i \leq T_i$ in (Mok, 1983). The resulting model is called the *constrained*-deadline model ($D_i \leq T_i$ must hold for each task τ_i).

Another deadline model of interest is the *arbitrary*-deadline model (Lehoczky, 1990) (any deadline assignment is allowed). The constrained-deadline model simplifies HRT systems' analysis by reducing the amount of workload competing with a task of interest. If a task set is HRT-schedulable under the constrained deadline model, then a job of a task is completed before the job's deadline, which is no later than the release of the next job. Thus, at most one job of any other task completes with the task of interest at any fixed time instant. In contrast, the arbitrary-deadline model supports a larger set of workloads.

2.3.2 Task Dependency Models

Under the sporadic task model, job $J_{i,j+1}$ cannot be executed before the completion of job $J_{i,j}$. This assumption defines a one-way dependency between these jobs. We call these dependencies *sporadic*. The sporadic model is natural for tasks whose jobs use computed results of previous jobs. Unfortunately, a sporadic dependency may postpone the execution of a job if the previous job of the same task still needs to be completed.

At the same, some tasks do not have explicit dependencies between consecutive jobs (*e.g.,* a task whose jobs independently process data from a periodic sensor). The *npc-sporadic* (no precedence constraints) task model allows successive jobs of the same task to be scheduled in parallel (each job is still a sequential code). It was first studied in (Baker and Baruah, 2009) (for HRT systems) and (Erickson and Anderson, 2011) (for SRT systems), and formalized in (Yang and Anderson, 2014). Other work pertaining to this model includes (Voronov et al., 2018). The main benefit of this model is that response-time analysis under the npc-sporadic task model typically yields lower response-time bounds than under the sporadic model. We explain the analytical reasons behind this observations after Example 2.2.

Unfortunately, the npc-sporadic task model may be too extreme for some workloads, as it prohibits any dependencies. At the same time, some tasks may not require the most recent data, but can process older data instead (Amert et al., 2021b). To address these issues, the *rp-sporadic* task model was proposed to generalize the sporadic and npc-sporadic task models (Amert et al., 2019). Each rp-sporadic task τ_i has a parallelization level P_i, meaning up to P_i jobs of τ_i can be scheduled in parallel. The following example illustrates the difference between these task models.

Example 2.2 (see Figure 2.3). Consider a task τ_1 with $C_1 = 4$ and $T_1 = 5$ scheduled by some algorithm on a processor with four CPUs under three different task models (sporadic, rp-sporadic with $P = 2$, and npc-sporadic). Assume that τ_1 periodically releases its first four jobs J_0, J_1, J_2, J_3 (we omit the task index for simplicity) at time instants 0, 5, 10, 15, and each job has an execution requirement of 4 time units.

Consider first the case where there is no other workload in the system (the contention-absent case). The schedule of the first four jobs is identical under each of these task models (shown in Figure 2.3a).

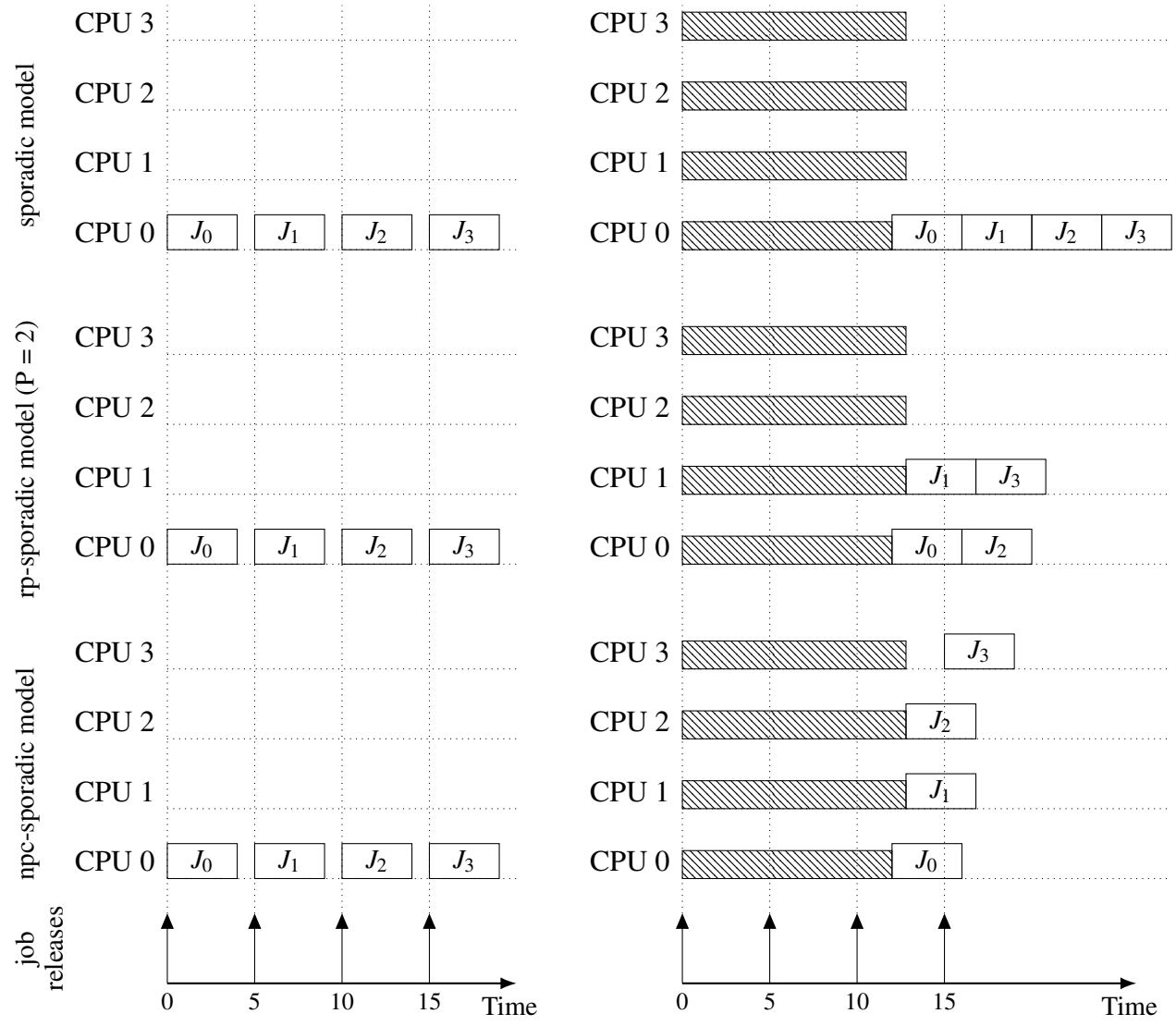
Consider now the case with an additional workload that has a higher priority (according to the scheduling algorithm) than J_0, J_1, J_2, J_3 (the contention-present case). The corresponding schedules are shown in Figure 2.3b. Assume that CPU 0 is busy with an additional workload during $[0, 12]$, and CPUs 1-3 are busy during time interval $[0, 13]$. Under the sporadic task model, the four jobs of interest are scheduled sequentially, and J_3 is completed at time 28. Under the rp-sporadic task model, two jobs are scheduled in parallel during the time interval $[13, 20]$, and J_3 is completed at time 21. Under the npc-sporadic task model, four jobs are scheduled in parallel during the time interval $[13, 16]$, and J_3 is completed at time 17. $\Diamond$

Example 2.2 illustrates an important observation. If a task has multiple uncompleted jobs at any given time, the remaining workload will execute faster under more relaxed models (assuming more than one free CPU exists). Thus, the same task sets tend to have lower analytical response-time bounds (as well as actual response times) under the rp- and npc-sporadic task models. An informal generalization of this observation can be formulated as follows.

> **Observation 1.** Under a relaxed task model, the analysis tends to be more "optimistic" than under a constrained task model. This means a lower response-time bound or "higher schedulability" (more task sets are schedulable under a given scheduler) can be achieved under the relaxed model. The analytical reason for this is that in the worst case (large response time of a specific job of interest), the task of interest can significantly increase its resource consumption (*e.g.*, $\times m$ under the npc-sporadic task model compared to the sporadic models) to "catch up."

In Observation 1, we mentioned higher schedulability. We illustrate this with the following example.

Example 2.3 (see Figure 2.4). Firstly, consider a task τ_1 with $C_1 = 3$ and $T_1 = 2$, scheduled alone under some algorithm on a processor with four CPUs under the three different task models (sporadic, rp-sporadic with $P = 2$, and npc-sporadic). The schedule of the first six jobs is stable (the response time R of each job is



(a) No higher-priority workload (the contention-absent case).

(b) With a higher-priority workload (the contention-present case).

Figure 2.3: Comparison of sporadic, rp-sporadic, and npc-sporadic task models.

the same) and identical under the rp- and npc-sporadic task models, see Figure 2.4a. However, under the sporadic task model, the response time of jobs increases indefinitely (3, 4, 5, 6, 7, ...).

Secondly, consider a task τ_2 with $C_1 = 5$ and $T_1 = 2$, scheduled alone under some algorithm on a processor with four CPUs under the three different task models (sporadic, rp-sporadic with $P = 2$, and npc-sporadic); see Figure 2.4b. Under both the sporadic and rp-sporadic task models, the response time grows indefinitely, while under the npc-sporadic task model, the schedule is stable. $\Diamond$

Example 2.3 illustrates how the npc-sporadic model has the highest schedulability among the models considered, and the sporadic model has the lowest schedulability. A sporadic task with a utilization of more than one cannot be scheduled by any algorithm, while an npc-sporadic (resp., rp-sporadic) task τ_i can tolerate a utilization of m (resp., P_i) when scheduled on a system with m (resp., at least P_i) CPUs.

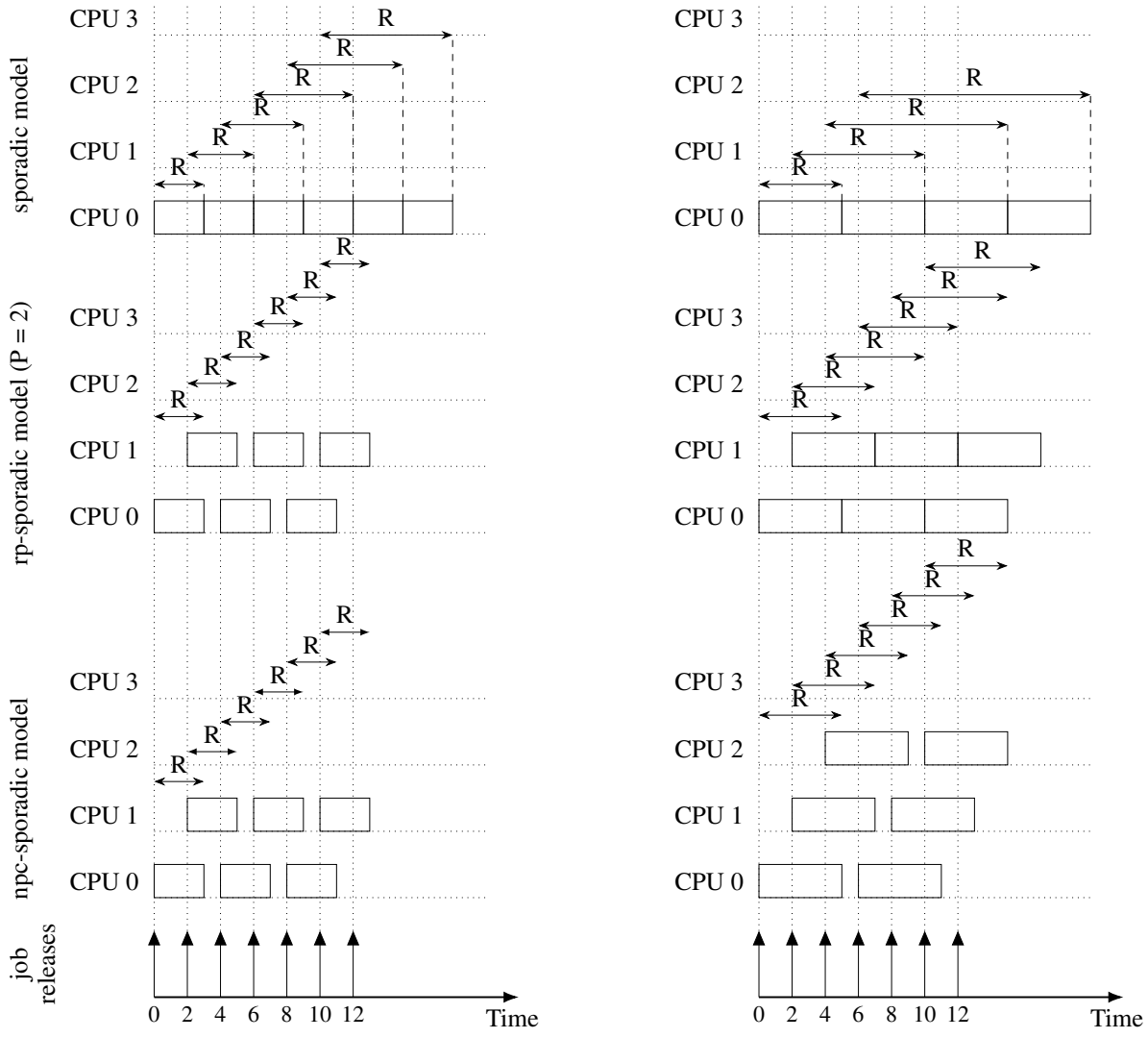
2.4 Parallel Task Models

As explained in Section 2.1, the notion of a sporadic task was defined to model a sequential piece of code that is invoked sporadically. Different tasks were expected to be completely independent. Unfortunately, the single-threaded performance of a CPU is limited by the laws of physics.

The first limiting factor is the information propagation speed. According to the special theory of relativity, the speed of any signal carrying information is limited by the speed of light (about $3 \cdot 10^8 m/s$ in a vacuum). Assuming a CPU frequency of 4 GHz, a signal can travel about 7.5 cm (3 inches) per clock cycle in a vacuum (and even less in an actual CPU). The second limiting factor is the heat dissipation of the chip. Because of the first factor, chips tend to be relatively small. Small chip area affects the efficiency of the cooling system (limits the power dissipated). High chip temperatures reduce the speed of transistors and may damage the chip itself. These two factors limit the significant increase in CPU frequency.

One solution industry has found is to build multiple independent CPUs into a single processor. Unfortunately, with the single-thread frequency limit, sequential task models may not be useful for modern applications. To overcome this limitation, several task models have been proposed to allow a single job to use more than one CPU at a time.

Several parallel task models have been proposed; in this dissertation, we mention two: *Gang* and *DAG* scheduling. The Gang task model was defined in (Ousterhout et al., 1982). This model assumes a task to occupy ν CPUs simultaneously; ν may be fixed or chosen offline/online. A job of a task cannot be scheduled



(a) τ_1 with $C_1 = 3$ and $T_1 = 2$.

(b) τ_2 with $C_1 = 5$ and $T_1 = 2$.

Figure 2.4: Comparison of sporadic, rp-sporadic, and npc-sporadic task models (overutilized task).

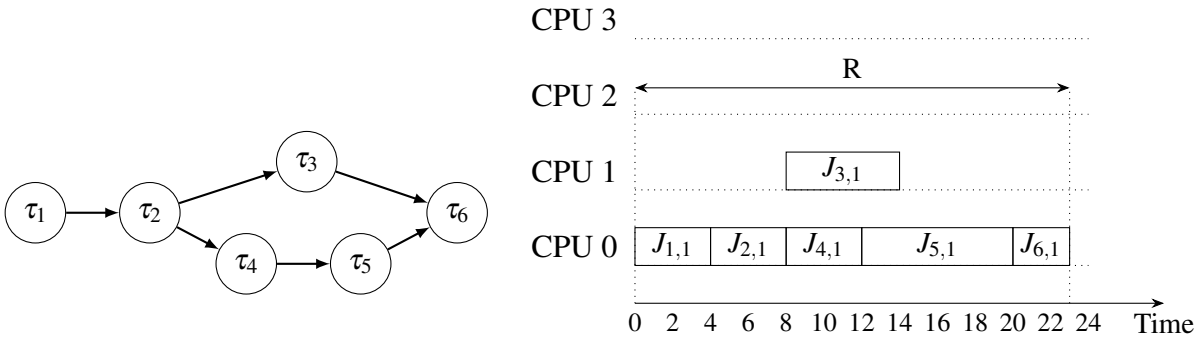
on fewer than ν CPUs. Gang scheduling research includes (Blazewicz et al., 1986; Feitelson and Rudolph, 1992; Kato and Ishikawa, 2009; Ahmed Bhuiyan et al., 2019; Dong et al., 2021; Lee et al., 2022; Nelissen et al., 2022).

The Gang model was generalized in (Collette et al., 2008) to support a variable number of threads with variable execution rates. Another generalization was made in (Lakshmanan et al., 2010) (*fork-join model*). Under this model, each job is a sequence of parallel (produced by the *fork* function in OpenMP (Chandra et al., 2001)) and sequential (produced by the *join* function) segments. A similar model was used in (Saifullah et al., 2013). All of the proposed models require parallel regions of a job to be scheduled simultaneously. Such a requirement was lifted in (Baruah et al., 2012) with the *DAG* (*direct acyclic graph*) task model. We extend this model in this dissertation and briefly mention other DAG models in Section 2.4.2.

Under the DAG task model, a task is modeled as a DAG $G = (V, E)$, where V is a set of nodes and E is a set of edges among them. To simplify the notation, we omit the graph index when discussing a single graph. Each node $v_i \in V$ represents an rp-sporadic task τ_i with a WCET of C_i. All graph nodes share the same *graph period* T_G, *relative deadline* D_G, and *parallelization level*, and release their i^{th} jobs at the same time. We call the set of i^{th} jobs a *graph instance*. The edges of the graph represent dependencies between jobs of the same instance: a job must wait to begin execution until all jobs it depends on have completed. The response time of an instance of G is the maximum response time among all instance jobs; R_G is called a *graph response-time bound* if R_G is an upper bound on any graph instance. Throughout the dissertation, we use graphs to represent the respective graph tasks.

Example 2.4 (see Figure 2.5). Consider a single graph task G with six nodes depicted in Figure 2.5a. Consider its schedule under some scheduling algorithm in Figure 2.5b. Tasks τ_2 and τ_3 depend on task τ_1, so $J_{3,1}$ and $J_{4,1}$ cannot start before the completion of $J_{2,1}$ (even though three CPUs are free); the same happens with $J_{5,1}$ and $J_{4,1}$. However, $J_{3,1}$ and $J_{4,1}$ are scheduled in parallel between time instants 4 and 8. The response time of the first instance of G is 23 time units. $\Diamond$

Note that most papers on real-time DAG scheduling use the sporadic task model or the npc-sporadic task model even if they do not formally define "npc" (*e.g.*,, (Bonifaci et al., 2013), (Li et al., 2013), or (Parri et al., 2015)). We generalize both of these approaches using the rp-sporadic task model.



(a) DAG G.

(b) A schedule of the first instance of G released at 0.

Figure 2.5: An example of graph task G with six nodes $\tau_1, ..., \tau_6$.

EDF scheduling of a DAG instance. Under the DAG model, we assume all jobs of the same instance are released at the same time. Thus, all ready jobs of the same instance have the same priority for the scheduler. As mentioned in Definition 2.2, deadline ties are broken consistently using node indexes.

We now discuss the classification of job dependencies under the DAG model.

2.4.1 Classification of Job Dependencies

In essence, any real-time task model describes jobs (representing strictly sequential code execution) and placement rules (which jobs must or must not be scheduled together). Complex models typically capture more complex job dependency behaviors. For example, under the sporadic task model (described in Section 2.1), a job depends only on the previous job of the same task.

Because such a dependency is required by a sporadic pattern of job releases, we call a dependency between jobs of the same task sporadic. Under the DAG model, sporadic dependencies are established between jobs of the same node; these dependencies are defined by the sequential task model used to define nodes.

The DAG task model introduces *regular* dependencies defined with the graph itself. Formally, we call dependencies between jobs of the same graph instance regular. The dependencies of each graph instance are the same (defined with the same graph). This is necessary because the number of jobs produced by each node is infinite, and dependencies should be consistent between instances to keep the analysis of a single graph G tractable. We require graph G to be acyclic; it is necessary because otherwise even a single instance cannot be scheduled due to the circular dependencies. We illustrate this point with the following example.

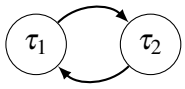


Figure 2.6: An example of cyclic graph task G.

Example 2.5 (see Figure 2.6). Consider a single sporadic DAG with two nodes depending on each other as illustrated in Figure 2.6. By definition, the cycle indicates that job $J_{1,1}$ depends on job $J_{1,2}$, and $J_{1,2}$ depends on $J_{1,1}$. Thus, none of these jobs can become ready (and scheduled) under any scheduler. $\Diamond$

Thus, the regular dependencies of the graph must be acyclic. In Chapter 4, we show how to deal with graphs with dependency cycles, but the regular dependencies must remain acyclic.

Regular dependencies cover dependencies between jobs of the same instance, and sporadic dependencies cover dependencies between jobs of different instances but the same node. We call the remaining class of dependencies (between jobs of the different instances and different nodes) as *delay* dependencies. For each sporadic or delay dependency $\tau_i \rightarrow \tau_j$, if the s^{th} job of node τ_j depends on the k^{th} job of node τ_i, we call $(s-k)$ as the dependency's *level*. Our model does not allow non-positive dependency levels because this would indicate a dependency on future jobs. We call a delay dependency *forward* if it does not form a cycle in the graph being transformed into a regular one, and *backward* otherwise.

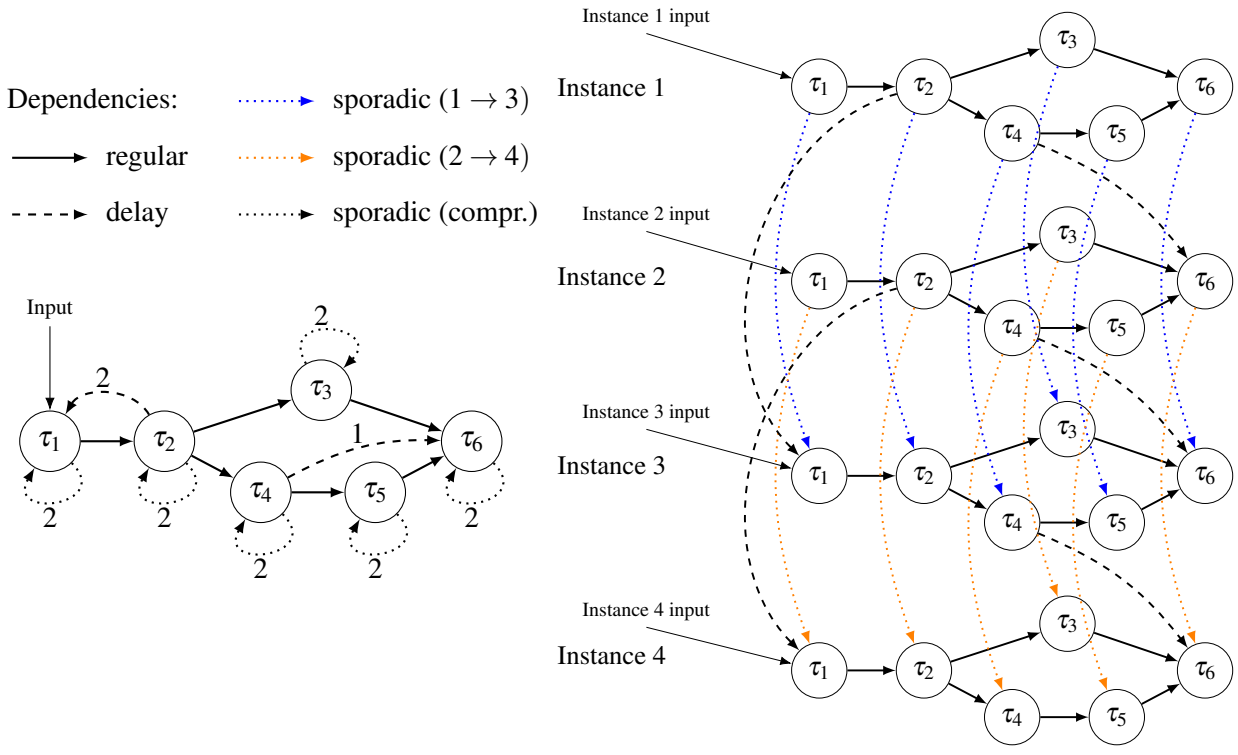
Our model only supports dependencies within the same graph task; different tasks are completely independent (as in many other models, such as the sporadic task model or the Gang model).

Example 2.6 (see Figure 2.7). Consider the same graph task with six nodes as in Example 2.4. Its regular dependencies are represented by G in Figure 2.5a (and shown in Fig 2.7a). Assume that its nodes represent rp-sporadic tasks with a parallelization level of two and there are additional delay dependencies $\tau_2 \rightarrow \tau_1$ (backward with level one) and $\tau_4 \rightarrow \tau_6$ (forward with level one).

The compressed form of the graph task is shown in Fig 2.7a; all dependencies among jobs of the first four instances are illustrated in Fig 2.7b. A number above a sporadic/delay dependency in the compressed form represents the dependency level (*e.g.,* 2 for all sporadic dependencies).

Note that a sporadic (or delay) dependency can be satisfied if the later job of the same task is completed earlier than the actual dependency. Consider a sporadic dependency $J_{1,1} \rightarrow J_{3,1}$. Under normal conditions, $J_{3,1}$ has to wait until the completion of $J_{1,1}$ (to get some data from it).

However, dependencies in the DAG model are defined to represent data movement across the jobs, so newer data of the same task (*e.g.,* the completion result of $J_{2,1}$) can be used instead of old data (the completion



(a) Compressed form of the graph task.

(b) All dependencies of the first four instances.

Figure 2.7: Graph task G of Example 2.6 and its dependencies.

result of $J_{1,1}$). Thus, if $J_{2,1}$ is completed before the completion of $J_{1,1}$ (*e.g.,* actual execution time of $J_{2,1}$ is very small), then $J_{1,3}$ can be scheduled in parallel with $J_{1,1}$. ◊

2.4.2 Other Real-Time Graph Models

In this subsection, we describe other graph models used in real-time research.

Typed graphs. This task model assumes multiple different execution engines (*e.g.,* CPU, GPU, digital signal processor, or field-programmable gate array) on the system platform and specifies an execution engine type for each node. This model was first defined in (Jaffe, 1980) (considering the scheduling of a single instance of a single graph task). Later research on real-time typed graphs includes (Baruah, 2020) (single DAG instance) and (Han et al., 2019; Chang et al., 2020) (constrained deadlines), whose authors considered the scheduling of a single DAG with fully available computational resources of several types. Chang *et al.* (2022b) considered list scheduling of a single DAG using the SAT solver (via Satisfiability Modulo Theories) to get

the exact response-time bound. Han *et al*. (2021) considered the federated scheduling of constrained-deadline multi-DAG typed tasks (the federated scheduling approach is described in Section 2.5). This federated scheduling approach (with two types of tasks—heavy and light) was extended in (Lin et al., 2022) (with four types of tasks—heavy-heavy, heavy-light, light-heavy, and light-light) on two types of execution engines. The only SRT paper on typed DAGs is (Yang et al., 2016), which considered the typed DAG task model without sporadic or delay dependencies.

Multi-rate graphs. This task model assumes possibly different periods for tasks that are represented with DAG nodes. Under this model, exact job-to-job dependencies are resolved during job release and not known a priori (*e.g.,* a regular dependency $\tau_i \rightarrow \tau_j$ means that a job of τ_j depends on the latest released job of τ_i). To our knowledge, most papers on multi-rate graphs convert them into some form of DAG tasks for analysis. Notable works include (Forget et al., 2010; Saidi et al., 2017; Saito et al., 2018; Verucchi et al., 2020).

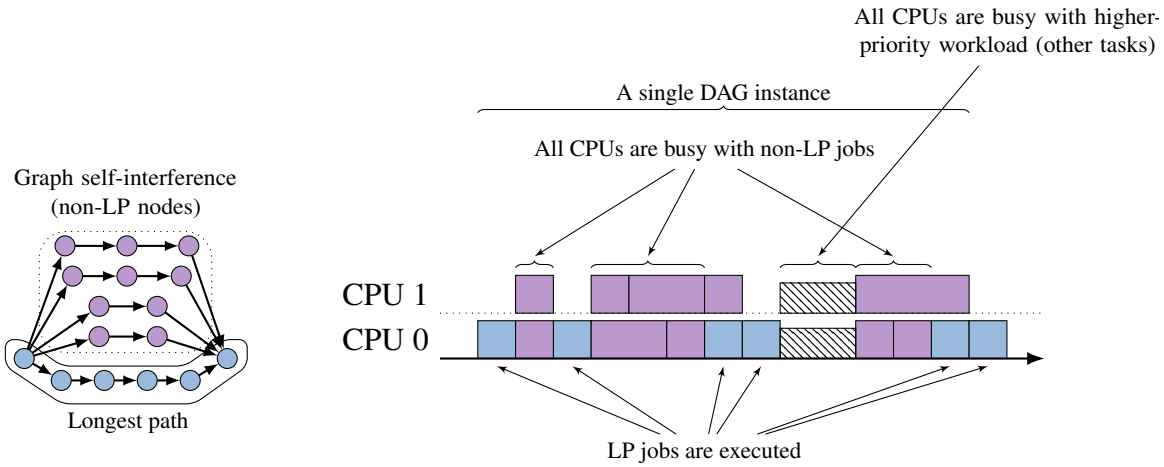
Conditional graphs. This task model introduces *conditional* nodes, which represent conditional if-then-else code constructions. These nodes act as switches that enable specific execution paths in the DAG (a node's task may be skipped within a single instance). This model was introduced in (Fonseca et al., 2015; Baruah et al., 2015). Other notable works include (Melani et al., 2015; Parri et al., 2015; Jiang et al., 2016, 2019a; Sun et al., 2019; He et al., 2023).

Other recent papers examine OpenMP graphs (Sun et al., 2021) and graphs with mutually exclusive nodes (Bi et al., 2022).

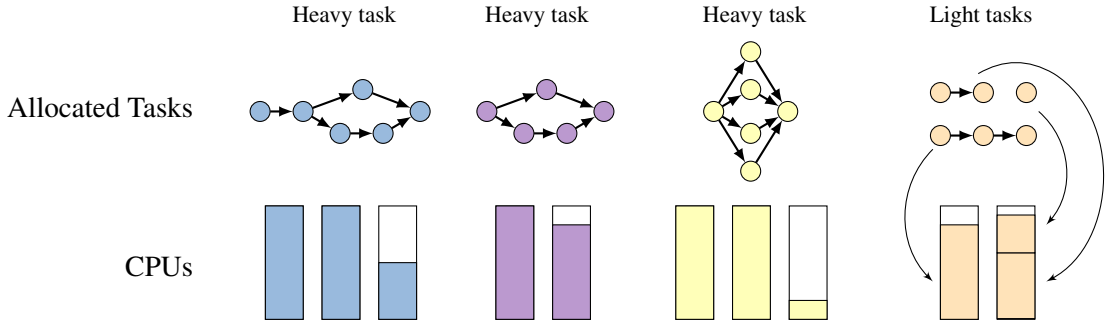
2.5 Graph Scheduling Approaches

In this section, we describe the existing approaches to scheduling a DAG task set on an identical multiprocessor platform. There are three main approaches to graph scheduling in real-time research. The standard approach is to directly analyze and bound the interfering workload to compute the response time for the graph of interest under the considered scheduler. However, there are two ways to simplify the analysis by simplifying the system: using federated scheduling or using decomposition scheduling. We briefly describe each of these approaches.

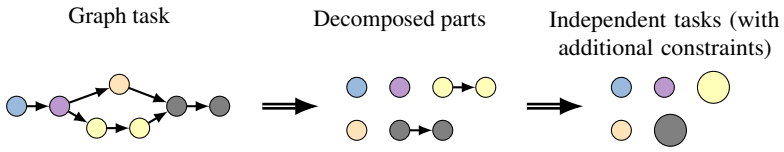
Direct interference computation. Under this approach (illustrated in Figure 2.8a), a bound on other tasks' interference over a DAG is computed. Typically, the interference is computed for one or more critical paths of the DAG (the longest path or the most recently completed path). Figure 2.8a shows an example of a DAG



(a) Direct interference computation on two CPUs (LP stands for the Longest Path).



(b) Federated scheduling of three heavy tasks and three light tasks on ten CPUs (CPU fill represents util. of allocated tasks).



(c) An example of a decomposition of a graph task.

Figure 2.8: Various scheduling approaches.

task (the left inset) and a schedule of its single instance (the right inset). The schedule shows the workload of the longest path and the self-interference of the instance. Papers that directly analyze interference include (Baruah et al., 2012; Bonifaci et al., 2013; Li et al., 2013; Baruah, 2014; Parri et al., 2015; Fonseca et al., 2019; He et al., 2019; Nasri et al., 2019; Wang et al., 2019; Sun et al., 2020; He et al., 2021, 2022; Zhao et al., 2022).

Federated scheduling. Under this approach (illustrated in Figure 2.8b), all graphs in the task set are classified into *heavy* and *light* graphs. Heavy graphs have utilization greater than 1.0, and light graphs have utilization of at most 1.0. Each heavy DAG is scheduled on a dedicated set of CPUs, while all light graphs are serialized into sporadic non-parallel tasks and scheduled on the remaining processors (via global or partitioned scheduling). This wastes some capacity of the dedicated CPUs: a heavy task with utilization of $1+\varepsilon$ ($\varepsilon \to 0$) requires 2 CPUs; a set of such tasks wastes almost half of the system capacity. Papers considering federated scheduling include (Li et al., 2014; Baruah, 2015; Li et al., 2017; Jiang et al., 2018; Wang et al., 2019; Jiang et al., 2019b; Dinh et al., 2020; Jiang et al., 2020).

Improved CPU sharing under the federated approach has been proposed in (Jiang et al., 2021, 2022b) (using virtual CPUs) and in (Osborne et al., 2022) (scheduling two jobs on the same CPU simultaneously using a hardware feature—simultaneous multithreading). Additional, semi-federated algorithms allow to reduce capacity loss (Jiang et al., 2017). Unfortunately, such algorithms often add significant overhead.

DAG decomposition. Under this approach (illustrated in Figure 2.8c), each DAG is decomposed into subgraphs. These subgraphs typically contain several nodes of the initial graph; however, some of the nodes may also be split. Each subgraph is scheduled as an independent (potentially sequential) task, preserving the dependencies of the original DAG by using well-defined releases and deadlines of the subgraph. Typically, the decomposed subtasks of all DAG tasks are scheduled via EDF. Papers considering DAG decomposition include (Liu and Anderson, 2010; Saifullah et al., 2012; Qamhieh et al., 2013, 2014; Jiang et al., 2016; Pathan et al., 2017; Guan et al., 2020; Jiang et al., 2020; Zhao et al., 2020; Ahmed and Anderson, 2022).

2.5.1 HRT Graph Scheduling

The problem of determining HRT schedulability of a single DAG instance is NP-hard in the strong sense (Ullman, 1975) (makespan minimization problem). In addition, the sporadic DAG model is itself a complex model, and complex models usually require complex solutions. Thus, many papers use some simplifications.

In this subsection, we show what simplifications are considered in HRT research (compared to our model/analysis). For example, even some recent papers consider only a single DAG task to find the most capacity-efficient analysis (Sun et al., 2020; Zhao et al., 2020; He et al., 2021). Unfortunately, modern complex applications are expected to have multiple tasks.

Another simplification used in papers is sporadic/delay dependency avoidance; HRT papers tend to ignore non-regular dependencies (directly or indirectly), which can be found in real algorithms. Two methods of avoiding non-regular dependencies are presented based on whether deadlines are constrained.

Constrained/implicit graph deadlines. Similar to the sporadic task model, a constrained-deadline (resp., implicit-deadline) task set contains only graph tasks with $D_G \leq T_G$ (resp., $D_G = T_G$). An HRT-schedulable task system requires all deadlines of all jobs to be met. Thus, an instance of a DAG task G released at t must be completed by $t + D_G$. Due to the definition of the graph period, the next instance of G cannot be released before $t + T_G$. Since $D_G \leq T_G$ holds for both models, two consecutive instances of G cannot be scheduled at the same time. Thus, sporadic or delay dependencies of any level are implicitly satisfied (at the cost of reduced schedulability compared to SRT). Informally, the standard HRT constrained-deadline approach considers only coarse-grained (graph-instance to graph-instance) sporadic dependencies, while our model allows fine-grained (node to node) sporadic and delay dependencies.

Papers that consider constrained or implicit deadlines include (Saifullah et al., 2012; Bonifaci et al., 2013; Li et al., 2013; Qamhieh et al., 2013; Baruah, 2014; Li et al., 2014; Qamhieh et al., 2014; Jiang et al., 2016; Li et al., 2017; Pathan et al., 2017; He et al., 2019; Jiang et al., 2019b; Nasri et al., 2019; Guan et al., 2020; Jiang et al., 2020; Sun et al., 2020; Zhao et al., 2020; He et al., 2021, 2022).

Arbitrary deadlines. The arbitrary deadline model does not impose any restrictions on D_G and T_G (*e.g.,* a task set may contain G such that $D_G > T_G$). Under this deadline model, the simultaneous execution of different instances is not implicitly forbidden, which requires more sophisticated analysis than under constrained or implicit deadlines. To simplify the analysis, papers that consider arbitrary deadlines do not consider sporadic or delay dependencies at all, thereby assuming the npc-sporadic task model for all graph nodes. Informally speaking, the standard HRT arbitrary-deadline approach ignores all non-regular dependencies, while our model, in contrast, allows them.

Papers that consider arbitrary deadlines include (Baruah et al., 2012; Bonifaci et al., 2013; Li et al., 2013; Baruah, 2015; Parri et al., 2015; Fonseca et al., 2019; Wang et al., 2019).

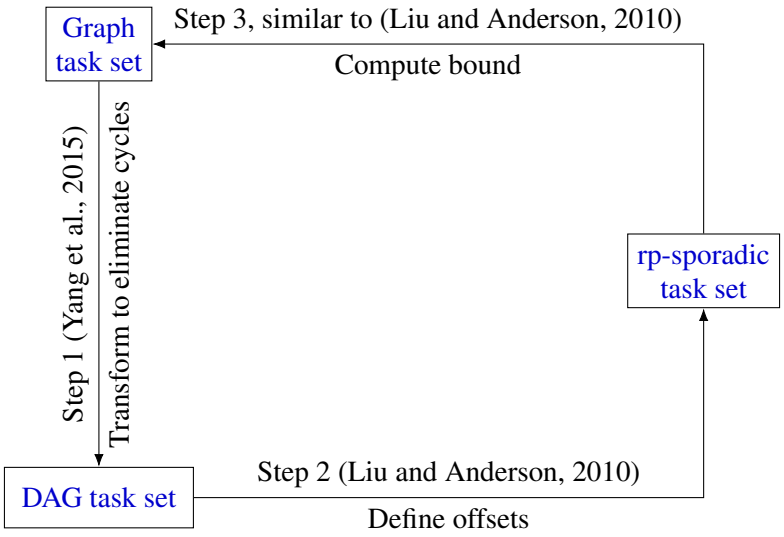
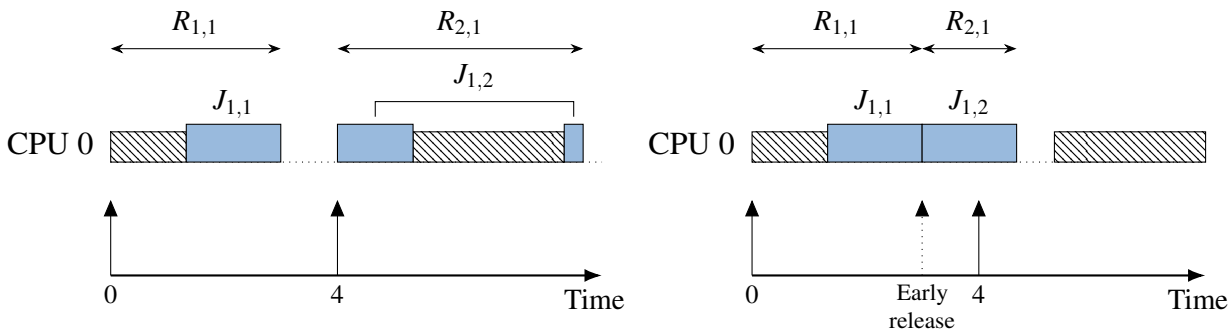


Figure 2.9: Response-time bound computation process.

2.5.2 SRT Graph Scheduling

In this subsection, we briefly describe the considered SRT approach for computing the response-time bound of a sporadic graph task on an identical multiprocessor. We describe it in detail later in Chapter 4 (including potential drawbacks and suggested improvements); here we give a brief description. The approach is somewhat similar to the decomposition approach mentioned above; it contains three steps (illustrated in Figure 2.9). The first step is to convert a sporadic graph task into a set of sporadic tasks with predefined offsets. Then, per-task response-time bounds for the obtained task set are computed. Lastly, these bounds are used to compute the resulting graph's end-to-end response-time bound. The approach was proposed by Liu and Anderson (2010) for sporadic DAG tasks, and was later extended by Yang *et al.* (2015) to support sporadic graph tasks with delay dependencies.

Note that this scheduling approach for response-time bound computation can postpone node execution due to the offsets introduction. Such postponement may be avoided with the concept of *early releasing*: a job J can be considered for scheduling as soon as all jobs it depends on have completed, even if this occurs before J's actual release time, as long as its scheduling priority remains unchanged (see Figure 2.10). Since most G-EDF analyses are not broken by early releasing, the same response-time bound applies to the original unmodified schedule.



(a) Without early resealing.

(b) With early releasing.

Figure 2.10: Schedule of two jobs $J_{1,1}$ and $J_{1,2}$ of a sporadic task τ_1 ($J_{1,2}$ depends on $J_{1,1}$) with a higher-priority workload. The response time of the second job is significantly smaller with early releasing, while its priority (defined by the absolute deadline) is unchanged.

SRT papers that consider the same or a very similar scheduling approach include (Liu and Anderson, 2010; Yang et al., 2015; Ahmed and Anderson, 2022). Note that a few SRT papers consider different approaches (*e.g.,* federated scheduling in (Liu and Anderson, 2011; Jiang et al., 2018)).

2.6 System Model Extension

Most of the research described in Section 2.5 considers a *fully available* computing platform—the scheduler can use all platform resources at any time instant. However, as described in Section 1.2, we consider systems with isolated components and accelerator usage. Such isolation implies that resources are not fully available; we provide resource and isolation models in this section.

Accelerator model. While most of the research described in Section 2.5 considers an identical multiprocessor model, we have to consider the usage of hardware accelerators (HACs) because modern processors do not provide enough computational power (as discussed in Chapter 1).

We assume that some graph nodes may have one or more accelerator accesses as a part of their workload. Unfortunately, accesses to HACs tend to be non-preemptive (*e.g.,* GPU preemptions typically introduce prohibitively high overhead). Since the total number of CPUs is typically larger than the total number of HACs (*e.g.,* 32 CPUs with 2 or 4 HACs), these accesses need to be managed. We assume that accelerator accesses are arbitrated with a locking protocol (one for each type of HACs used in the system) with bounded blocking time (*e.g.,* GPUSync (Elliott et al., 2013)). We specify the details of the used protocol in Chapter 5.

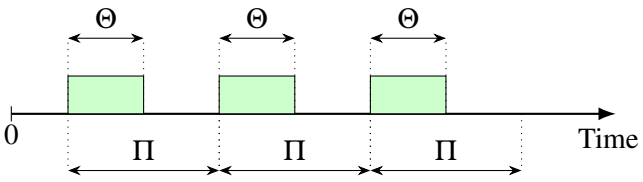


Figure 2.11: Periodic reservation Λ.

Thus, all graph nodes can be assumed to be CPU nodes whose worst-case execution times are inflated to include HAC blocking and execution times; these CPU tasks may contain non-preemptive regions.

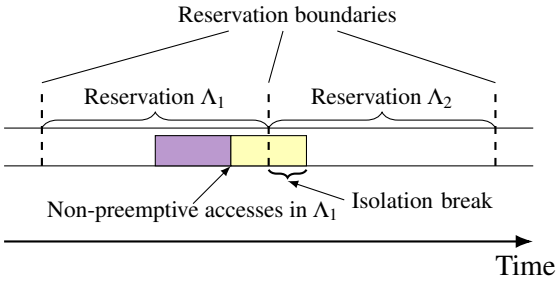
An alternative approach can be used in which the CPU and accelerator-related parts each are considered as separate scheduling entities (*e.g.,* (Yang et al., 2018)). Under this approach, the scheduling of these parts is considered independently. We also describe and apply it to reduce response-time bounds in Chapter 5.

Resource model. As discussed in Section 2.5.1, the response-time/schedulability analysis of the DAG task model on the fully available platforms is not simple, and becomes even more when complicated a partially available computing platform. To avoid overall over-complication, we consider a relatively simple periodic time slicing: each system component Ω_i has *exclusive* periodic access to a set of computing resources. This notion is called the *periodic component reservation* (PCR) model and is similar to the Single Time Slot Periodic Partition model (Mok et al., 2001). We call this exclusive access a *reservation*; each reservation Λ_i is defined by its length Θ_i, period Π_i ($\Theta_i \leq \Pi_i$), and its set of computing resources Υ_i (m_i CPUs and g_i hardware accelerators).

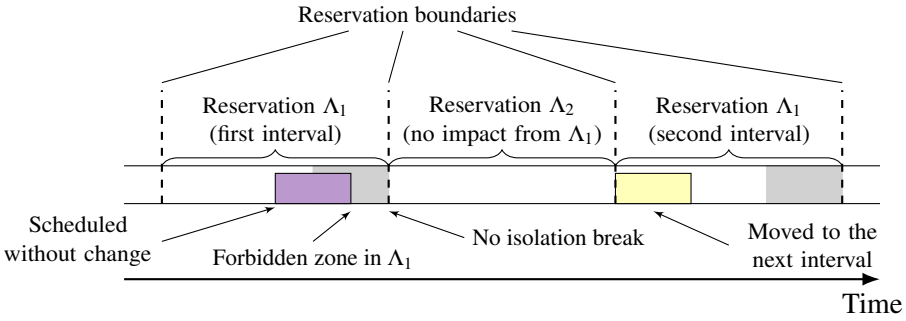
An example of a periodic reservation is shown in Figure 2.11. Note that exclusivity of the resources requires that any CPU or HAC may be in at most one reservation at any time instant.

Component isolation model. We say that system component Ω_i is *isolated* if no workload of any other component is scheduled during Λ_i. The access exclusivity of the resource model ensures the temporal isolation of different system components as long as each component uses only its reservation's resources.

While preemptive CPU workloads can be preempted at the end of a reservation with low overhead, non-preemptive accelerator accesses near the reservation boundary can break isolation (see Figure 2.12a). To avoid breaking isolation, we use *forbidden zones* (Holman and Anderson, 2006). A forbidden zone for a given HAC access in Λ_i is a region of time in which that access may not be initiated, as it may cross the end boundary of the current reservation interval (see Figure 2.12b); the zone length is thus the worst-case duration of that access. Note that accelerator usage by other components has no impact on Ω_i's forbidden-zone lengths.



(a) HAC access near boundary (no forbidden zone).



(b) HAC access near boundary (with forbidden zone).

Figure 2.12: Isolation break and forbidden zone.

2.7 Relevant Prior Work

In this section, we describe relevant real-time research and highlight its inapplicability for the considered problem (scheduling isolated components containing graph-based workloads with HAC accesses). In Section 2.7.1 we discuss existing HAC research. In Section 2.7.2 we cover the remaining graph research not mentioned in Sections 2.4 and 2.5. In Section 2.7.3 we summarize research on real-time isolation. Papers that address at least two of the three topics are discussed in Section 2.7.4.

2.7.1 Hardware Accelerators

There are several types of hardware accelerators (Peccerillo et al., 2022). In this section, we mention GPUs, field-programmable gate arrays (FPGA), and application-specific integrated circuits (ASIC) because they are commonly used to speed up AI workloads: (Nurvitadhi et al., 2017; Hesse, 2021; Wang and Luo, 2022) (FPGA) and (Seshadri et al., 2022) (Google TPU ASIC). Notable work on real-time scheduling analysis with FPGAs includes (Danne and Platzner, 2005; Biondi et al., 2016) (single CPU). Some papers propose new (hardware) schedulers for systems with FPGAs (Zhu et al., 2019; Saha et al., 2021; Kohútka, 2022). To

the best of our knowledge, there are only two papers (Ramezani, 2021; Xu et al., 2022) that consider graph scheduling in the presence of FPGAs (heuristic comparison for the makespan minimization problem). To the best of our knowledge, there is no ASIC-based real-time scheduling research except (Derafshi et al., 2019) (a hardware scheduler with ASIC implementation). Thus, as the FPGA/ASIC real-time research is limited, we mostly focus on GPUs; our approach, however, supports FPGAs and ASICs as well.

Managing GPU accesses. The standard scheduling approach in real-time GPU research is to provide a real-time analysis from the CPU point of view with an additional management of GPU requests because a modern system typically has several CPUs (*e.g.,* 8-16-32) and a few GPUs (*e.g.,* 1-4). There are three approaches to arbitrating GPU/HAC accesses.

Under the first approach, a dedicated CPU is assigned for each GPU; workloads related to this GPU are partitioned onto the CPU (Xu et al., 2016; Golyanik et al., 2017; Xiang and Kim, 2019; Kang et al., 2021). Thus, only jobs from the dedicated CPU can access the considered GPU.

Under the second approach, a real-time locking protocol with proved access-time guarantees is used (Kato et al., 2011a; Elliott et al., 2013; Yang et al., 2013; Verner et al., 2014; Elliott and Anderson, 2014), or some locking protocol (Zou et al., 2023).

Under the third approach, a GPU driver and/or a hardware scheduler (provided by the GPU manufacturer) are used to manage accesses. Typically, full scheduling policy descriptions are not officially available (*e.g.,* NVIDIA) or well-defined (*e.g.,* AMD), so some work has attempted to understand the scheduling rules employed by the GPU drivers themselves through micro-benchmarking experiments (Otterness et al., 2016; Amert et al., 2017; Yang, 2018; Olmedo et al., 2020; Otterness and Anderson, 2020, 2021; Bakita and Anderson, 2022). Alternatively, the GPU driver may be modified to support specific scheduling rules (Kato et al., 2011b).

GPU preemption policies. Typically, accelerators are considered to be non-preemptive (or have high preemption overhead). We are aware of only one paper on fully preemptable GPU scheduling (Capodieci et al., 2018), but important details are hidden due to NVIDIA non-disclosure agreement.

Two methods have been proposed for adding (limited) GPU preemption support. The first method uses *kernel abortion*: a preempted GPU job is aborted and must be restarted from scratch on the next run (Park et al., 2015; Lee and Al Faruque, 2016; Lee et al., 2018, 2020; Han et al., 2022). This method may cause high overheads in the case of long jobs. The second method uses *kernel slicing*: a GPU job is sliced into

smaller sub-jobs; preemption occurs after a sub-job has completed (Basaran and Kang, 2012; Zhou et al., 2015; Chen et al., 2017; Wu et al., 2017). Some papers consider similar techniques (Hartmann and Margull, 2019; Yao et al., 2021). This method provides only limited preemption with some slicing-related overhead and, sometimes, requires source code modification.

Other GPU frameworks consider GPU requests to be non-preemptive (Kato et al., 2011b,a; Elliott, 2015). In this dissertation, we focus primarily on non-preemptive HACs (although we support preemptive HACs as well); we explain how to manage HAC requests using the first and second approaches in Chapter 5.

2.7.2 Real-time Graph Research

Prior work on real-time graph scheduling has mostly focused on the HRT DAG model on an identical multiprocessor (described in Section 2.5). At the same time, the problem considered in this dissertation implies a partially available SRT platform with accelerator accesses. Apart from the temporal correctness criteria and the platform, existing DAG research differs from our model by avoiding dependencies (as described in Section 2.5.1) (Qamhieh et al., 2013; Bonifaci et al., 2013; Baruah, 2014; Zhao et al., 2020; Li et al., 2013; Jiang et al., 2019b; Wang et al., 2019; Fonseca et al., 2019; Baruah, 2015), by a different scheduler (*e.g.,* Fixed-Priority Scheduler) (Li et al., 2014; Serrano et al., 2016; Jiang et al., 2017; Ren et al., 2018; Nasri et al., 2019; Verucchi et al., 2020; Chang et al., 2022a), or by objectives (*e.g.,* minimizing makespan or energy consumption instead of schedulability analysis) (Xie et al., 2017; Bhuiyan et al., 2018; Wang et al., 2019; Bhuiyan et al., 2020; Hu et al., 2021).

Existing SRT DAG research considers only sporadic dependencies of level one on an identical multiprocessor (Liu and Anderson, 2010; Yang et al., 2015, 2018). Among other models described in Section 2.4.2, only the typed graph model supports graphs with HAC accesses. Unfortunately, this model has not been extensively researched to cover the other issues we consider: graph dependencies and time isolation. Therefore, we incorporate accelerator accesses into a slightly extended DAG model, resulting in a more flexible model rather than working with the typed model.

2.7.3 Isolation of System Components

The real-time isolation of system components is typically done using the notion of virtual processors (or reservations) with guaranteed capacity. We consider a simple PCR model that requires the capacity to be supplied over a continuous time interval (see Section 2.6). Similar models have been used in different

contexts in (Giannopoulou et al., 2017; Springer and Zhao, 2021) (isolation of different task classes) and (Tămaş-Selicean and Pop, 2015) (mixed-criticality systems).

More general models have been studied in (Chakraborty et al., 2003) (uses lower and upper bounds on the contending resource), (Shin et al., 2008) (does not require continuity of the supply interval), and (Leontyev and Anderson, 2009) (changes container size over time). Unfortunately, these models cannot easily be used instead due to HAC non-preemptivity issues. In addition, the prior work with these models uses a simpler (and less general) task model (*i.e.,* the standard sporadic task) without accelerator accesses (Bini et al., 2009; Burmyakov et al., 2014; Abeni et al., 2019).

Works that consider graph models and component isolation (Buttazzo et al., 2011; Wu et al., 2014; Yang et al., 2019; Casini et al., 2019) do not address the complexities associated with the use of non-preemptive accelerators, such as the potential for component isolation violations. Nemati *et al*. (2011) considered accelerator accesses in component-based systems. However, they required that a given CPU be dedicated to a single component, and thus did not consider a model in which a component has exclusive access to a HAC.

2.7.4 Most Relevant Papers

In this subsection, we consider papers that provide analysis supporting at least two issues out of graph task, accelerator accesses, and component isolation. A summary of these papers can be found in Tbl. 2.1.

The second column of the table shows how the paper covers sporadic and delay dependencies. Some papers avoid dependencies as described in Section 2.5.1. As we use the rp-sporadic model, we use a parallelization level of $P = 1$ (resp., $P = m$) to indicate that the corresponding paper supports the sporadic (resp., npc-sporadic) node model. Our model supports intermediate values.

The third column of the table represents the accelerator access policy ("Locking" stands for the locking protocol, and "Sch. Entity" stands for treating an accelerator as an independent scheduling entity). Our model supports both policies as described in Chapter 5.

Note that none of the papers consider all three issues simultaneously. The work of (Yang et al., 2015) (the closest to the proposed research) partially inspired this dissertation; other papers consider at most one issue in detail.

Contributions of this dissertation. In Chapter 3, we provide a response-time analysis of rp-sporadic task model. We provide an analysis for an isolated single component in Chapter 5, where graph tasks of that

	Graph Model		Accelerators Usage	Component Isolation	HRT / SRT
	Supported level of sporadic depend.	Supports delay depend.			
(Serrano and Quinones, 2018)	Obviated[1]		Partially[2]	No	HRT
(Buttazzo et al., 2011)	Obviated[1]		No	Yes	HRT
(Wu et al., 2014)[4]	Obviated[1]		No	Yes	HRT
(Nemati et al., 2011)	Sequential Tasks		Locking	Partially[5]	HRT
(Ueter et al., 2018)	$P = 1$ only	No	No	Partially[3]	HRT
(Yang et al., 2015)	$P = 1$ only	Yes	Locking	No	SRT
(Yang et al., 2016)	$P = m$ only	No	Sch. Entity	No	SRT
(Yang et al., 2018)	$P = m$ only	No	Sch. Entity	No	SRT
This dissertation	$P \in [1, m]$	Yes	Both	Yes	SRT

[1] Fine-grained dependencies are replaced with per-instance graph dependency because of the HRT model with implicit/constrained deadlines (see Section 2.5.1).
[2] Graph must have only one accelerated node.
[3] Different CPU resource model.
[4] Task model considers communication costs among graph nodes.
[5] CPU can be occupied by only one component, no isolation over accelerators.

Table 2.1: Comparison of papers related to this dissertation.

component can use non-preemptive HACs. Our analysis supports any level of sporadic dependencies. We propose two methods to deal with non-preemptivity-related isolation issues.

2.8 Chapter Summary

In this chapter, we formally defined our task, platform, accelerator, and isolation models, as well as our scheduling policy. We reviewed the existing real-time research on graph scheduling, accelerator usage, and component isolation. In Chapter 3, we provide response-time analysis for the sequential rp-sporadic task model introduced in Section 2.3 assuming accelerator accesses. In Section 2.5, we discussed existing approaches for the graph scheduling. However, the SRT graph decomposition scheduling approach may not work efficiently for certain types of graphs. In Chapter 4, we describe the approach in detail, propose an analysis improvement method, and apply the analysis provided in Chapter 3. In Chapter 5, we further extend this analysis by considering the case of a single isolated component and discuss isolation-related issues.

CHAPTER 3: RESPONSE-TIME ANALYSIS OF RP-SPORADIC TASKS[1]

In this chapter, we analyze the rp-sporadic task model, defined in Section 2.3.2 to find an upper bound on the task-set response time. The remainder of this chapter is organized as follows. In Section 3.1, we provide necessary definitions. Then, in Section 3.2, we prove the response-time bound. Later, in Section 3.3, we derive a closed form of the obtained bound. We describe ways to improve the bound in Section 3.4, and conclude in Section 3.5.

3.1 Necessary Definitions

In this chapter, we consider the scheduling of a task set τ of n rp-sporadic tasks $\tau_1, \ldots, \tau_n$ on an identical multiprocessor with m CPUs. Each task τ_i is specified as (T_i, C_i, P_i, Φ_i), where T_i is its period, C_i is its WCET (as defined in Section 2.2), P_i is its parallelization level (as defined in Section 2.3.2), and Φ_i is its *offset* (a constant shift of all jobs of a task τ_i). For simplicity, we assume that tasks have implicit deadlines, i.e., $D_i = T_i$.

We assume that G-EDF breaks deadline ties arbitrarily but consistently (*e.g.,* by task index). We let $J_{i,j} \prec J_{k,l}$ denote that job $J_{i,j}$ has higher priority than job $J_{k,l}$. We define the total system utilization as $U = \sum_i u_i = \sum_i \frac{C_i}{T_i}$. In proving the bound in Section 3.2, we assume time to be continuous.

We assume that CPU computations of the considered tasks are fully preemptive. However, real workloads may need to use HACs (*e.g.,* GPUs). We assume that these HAC accesses are arbitrated via a locking protocol, as explained in Section 2.6. This arbitration adds non-preemptive sections to jobs; we denote the maximal length of a single non-preemptive section as B_{max}.

Feasibility conditions. An SRT task set τ is called *feasible* under scheduler A if its response-time can be bounded. Devi and Anderson (Devi and Anderson, 2008) showed that the sporadic task model has the

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

Amert, T., Voronov, S., and Anderson, J. H. (2019). OpenVX and real-time certification: The troublesome history. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 312–325.

following feasibility condition:

$$\forall \tau_i \in \tau, u_i \leq 1 \text{ and } U \leq m.$$

The rp-sporadic task model relaxes the sporadic task model, so the feasibility condition is also relaxed:

$$\forall \tau_i \in \tau, u_i \leq P_i \text{ and } U \leq m. \tag{3.1}$$

The necessity of these conditions can be simply proved by contradiction. In the case of strictly periodic releases, the amount of incoming work from a task τ_i is C_i per T_i time units. However, at most $P_i \cdot T_i$ work can be completed within the same period (at most P_i jobs are scheduled at the same time). Thus, the total incomplete workload generated by τ_i is at least $C_i - P_i \cdot T_i = T_i(u_i - P_i)$ per T_i time units. If $u_i - P_i > 0$ then the total incomplete workload grows without bound and task τ_i cannot have a bounded response time. The same contradiction can be achieved with the whole system if $U > m$ (the entire system becomes overutilized).

As the bound to be obtained in Section 3.2 uses only (3.1), condition (3.1) is, in fact, sufficient for an rp-sporadic task set to be feasible. In the rest of the chapter we assume that the task set satisfies (3.1).

3.2 The Basic Bound

Throughout this section, we consider a job of interest $J_{k,l}$; as the proven response-time bound holds for any job of interest, it inductively applies to all jobs of all tasks in the task system (over $\prec$). We consider an analysis window, and bound the amount of work that conflicts with the job of interest within this window.

Initially, we consider a simpler edge case (Lemma 3.2). For the more complex case, we first show that non-preemptive sections of lower-priority jobs can affect the execution of higher-priority jobs only if such sections are scheduled at the start of the analysis window (Lemma 3.3). To bound the response time for the job of interest, we first bound the total workload of high-priority jobs given their maximal response times (Lemma 3.4). Then, we show that the inductively assumed response-time bounds of high-priority jobs ensure the same bound for the job of interest if it is big enough (Lemma 3.5). Finally, we present our full response-time theorem (Theorem 3.1).

Definition 3.1. At a time instant t, job $J_{i,j}$ is *unreleased* if $t < r_{i,j}$ and *released* otherwise; $J_{i,j}$ is *complete* if it finishes execution by t; $J_{i,j}$ is *pending* if it is released but not completed; and $J_{i,j}$ is *ready* if it is pending and job $J_{i,j-P_i}$ is complete (*i.e.*,, $J_{i,j}$ can be scheduled at t).

Job of interest. We consider an arbitrary job $J_{k,l}$ of a task $\tau_k \in \tau$. Let t_d be the absolute deadline of $J_{k,l}$, *i.e.,* $t_d = r_{k,l} + T_k$. Let t_f be the completion time of $J_{k,l}$. We will show inductively with respect to $\prec$ that the response time of τ_k is bounded by $x + T_k + C_k$ for any positive x that is large enough (as formalized later in (3.9)). We assume $t_d \leq t_f$, for otherwise the response time of $J_{k,l}$ is less than T_k.

Definition 3.2. We let Ψ (resp., $\overline{\Psi}$) denote the job set consisting of all jobs that have higher (resp., lower) priority than $J_{k,l}$.

Definition 3.3. We say that a time instant t is *busy* if m jobs in $\Psi \cup \{J_{k,l}\}$ are scheduled, or there is a ready job in $\Psi \cup \{J_{k,l}\}$ that is not scheduled at t, and *non-busy* otherwise. Both busy conditions imply that every CPU executes a job. We say that a time interval $[t, t')$ is *busy* if all instants in it are busy.

Lemma 3.1. *For any task τ_i, the number of its ready jobs in $\Psi \cup \{J_{k,l}\}$ does not increase after t_d.*

Proof. All jobs in $\Psi \cup \{J_{k,l}\}$ are released within $[0, t_d]$. A pending job $J_{i,j}$ in this set can become ready after t_d only at the time instant when $J_{i,j-P_i}$ completes (and is no longer ready). Thus, the total number of ready jobs of τ_i in $\Psi \cup \{J_{k,l}\}$ does not increase after t_d. □

There are two cases for t_d: it is either a busy or a non-busy time instant. We will consider the non-busy case in Lemma 3.2 first and then the busy case in Lemmas 3.3–3.5.

Lemma 3.2. *If t_d is a non-busy time instant, and the response time of each job of τ_k released before $J_{k,l}$ is at most $x + T_k + C_k$, then the response time of $J_{k,l}$ is bounded by $x + T_k + C_k$. (No conditions on x except $x \geq 0$ are implied in this lemma.)*

Proof. By Lemma 3.1, the number of ready jobs in $\Psi \cup \{J_{k,l}\}$ does not increase after t_d. Any job in $\Psi \cup \{J_{k,l}\}$ that becomes ready after t_d has a CPU to be scheduled on (the same CPU where its predecessor was completed) when it becomes ready (illustrated in Figure 3.1 for $J_{k,l}$). Therefore, if t_d is not a busy time instant, then any later time instant is not busy, as jobs from $\Psi \cup \{J_{k,l}\}$ occupy fewer than m CPUs. Let t' be the first time instant when $J_{k,l}$ becomes ready.

If $t' \leq t_d$, then $J_{k,l}$ is ready at t_d. Thus, $J_{k,l}$ is scheduled at t_d (it is not a busy instant). Because all time instants after t_d are non-busy instants (all ready jobs in $\Psi \cup \{J_{k,l}\}$ are scheduled), the execution of $J_{k,l}$ cannot be postponed, and the response-time of $J_{k,l}$ is bounded by $T_k + C_k \leq x + T_k + C_k$.

If $t' > t_d$ (as shown in Figure 3.1), then $J_{k,l}$ becomes ready upon completion of $J_{k,l-P_k}$ (or an earlier job of τ_k, as P_k jobs of τ_k can be scheduled in parallel), which was released by time $t_d - T_k - P_k \cdot T_k$. By the

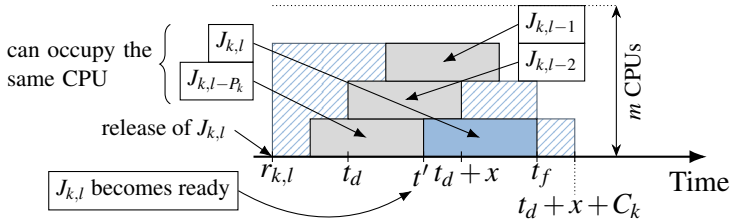


Figure 3.1: Lemma 3.2 illustration ($P_k = 3$).

lemma statement, $J_{k,l-P_k}$ must complete by time $t_d - T_k - P_k \cdot T_k + x + C_k + T_k = t_d + x + C_k - P_k \cdot T_k$. By (3.1), $C_k \leq P_k \cdot T_k$ (task set feasibility), so $t' \leq t_d + x$. As $J_{k,l}$ is scheduled immediately upon becoming ready (*e.g.*, occupying the same CPU as $J_{k,l-P_k}$), it completes by time $t_d + x + C_k$, within $x + T_k + C_k$ time units from $r_{k,l}$. □

We now consider the case when t_d is busy.

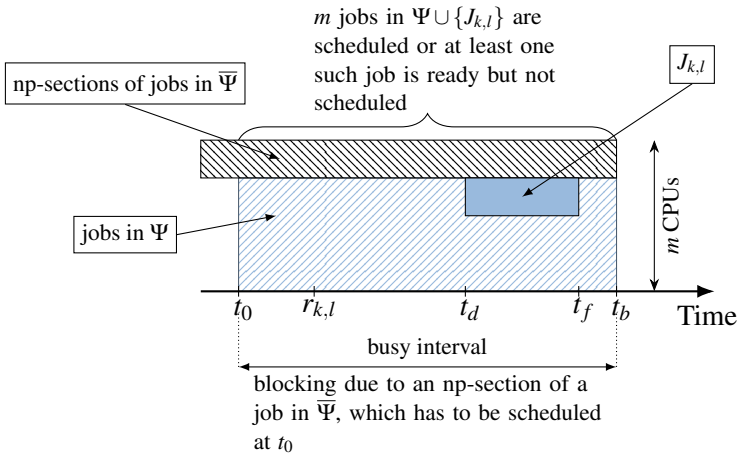
Definition 3.4. Let t_0 denote the first busy instant such that $[t_0, t_d)$ is a busy interval. Let t_b denote the last time instant such that $[t_d, t_b)$ is a busy interval.

The following lemma limits the number of lower-priority jobs in $\overline{\Psi}$ that can affect the execution of higher-priority ones.

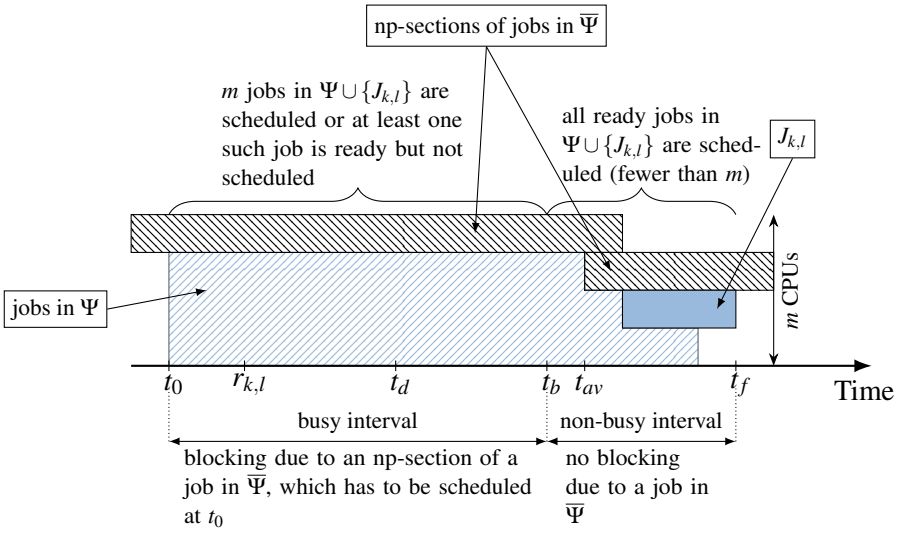
Lemma 3.3. *A non-preemptive section of a job $J_{i,j}$ in $\overline{\Psi}$ may block the execution of ready jobs in $\Psi \cup \{J_{k,l}\}$ within $[t_0, t_f)$ only if that section is scheduled at t_0. Moreover, such blocking may occur only within $[t_0, t_b)$.*

Proof. Consider the interval $[t_0, t_f)$, depicted in Figure 3.2 for two cases, (a) $t_b > t_f$ and (b) $t_b \leq t_f$ (note that t_{av} is defined later in Lemma 3.5). We begin by showing, in both cases, that all time instants after t_b are non-busy. By Definition 3.3, at most $m-1$ ready jobs in $\Psi \cup \{J_{k,l}\}$ are scheduled at t_b. By Lemma 3.1, the number of ready jobs in $\Psi \cup \{J_{k,l}\}$ does not increase after t_d. Thus, if a job $J_{g,h} \in \Psi \cup \{J_{k,l}\}$ becomes ready at some time $t > t_d$, then $J_{g,h-P_g}$ must have completed, and the CPU upon which it executed is available at t. Additionally, as jobs in $\Psi \cup \{J_{k,l}\}$ have higher priority than those in $\overline{\Psi}$, they remain scheduled until they complete, so no time instant after t_b is busy.

By Definition 3.3, if $J_{i,j} \in \overline{\Psi}$ blocks a job in $\Psi \cup \{J_{k,l}\}$ at $t' \in [t_0, t_f)$, then t' is a busy instant. As no time instant after t_b is busy, $t' \in [t_0, t_b)$. $J_{i,j}$ has lower priority than any job in $\Psi \cup \{J_{k,l}\}$, so it must therefore execute non-preemptively at every instant in $[t_0, t']$, or else it would be preempted. Thus, the non-preemptive section scheduled at t' must also be scheduled at t_0, and blocking by $J_{i,j}$ occurs only within $[t_0, t_b)$. □



(a) $t_b > t_f$.



(b) $t_b \leq t_f$.

Figure 3.2: Important time points in the analysis.

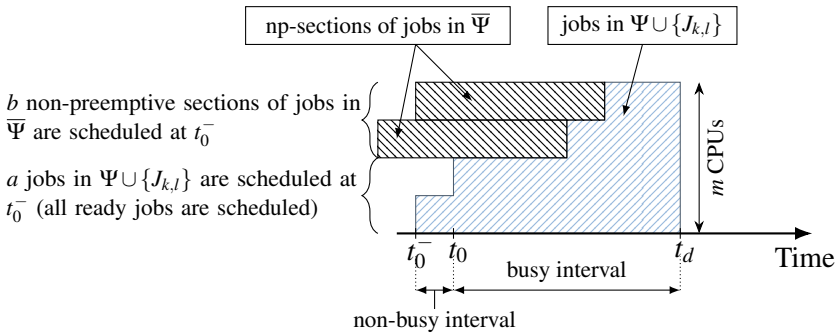


Figure 3.3: Lemma 3.4 illustration.

Let W_d be C_k plus the total workload that can potentially prevent the execution of $J_{k,l}$ at t_d. By Lemma 3.3, W_d includes the workload of non-preemptive sections of jobs in $\overline{\Psi}$ that are scheduled at t_0 and the workload of all jobs in $\Psi \cup \{J_{k,l}\}$.

By Lemma 3.4, given below, $L(x)$, defined next, is an upper bound for W_d.

$$L(x) = (m-1)C_{max} + B_{max} + \max_{\substack{\tau^* \subseteq \tau \text{ s.t.} \\ \sum_{\tau_i \in \tau^*} P_i \leq m-1}} \left(\sum_{\tau_i \in \tau^*} (u_i x + 2C_i) \right) \tag{3.2}$$

Lemma 3.4. *If t_d is a busy time instant, and the response time of each job $J_{i,j} \in \Psi$ is at most $x + T_i + C_i$, then $W_d \leq L(x)$.*

Proof. Let t_0^- be $t_0 - \varepsilon$ for an arbitrarily small $\varepsilon > 0$ such that $[t_0^-, t_0)$ is a non-busy interval, as illustrated in Figure 3.3. (If $t_0 = 0$, then we can conceptually view $[-\varepsilon, 0)$ as an interval where no work is scheduled.) Because ε is arbitrarily small, no scheduling events (jobs completions or releases) occur within $[t_0^-, t_0)$. To upper bound W_d, we first bound the workload at t_0 of jobs released before t_0^- in Claims 3.1 and 3.2 (all jobs in $\Psi \cup \{J_{k,l}\}$ that are ready at t_0^- are scheduled). Then, we bound the workload of jobs released within $[t_0, t_d)$ in Claim 3.3. Finally, we bound the workload completed over $[t_0, t_d)$ in Claim 3.4. (For clarity, claim proofs end with ■ while other proofs end with □.)

Let a (resp., b) be the number of jobs in $\Psi \cup \{J_{k,l}\}$ (resp., $\overline{\Psi}$) that are scheduled at t_0^-.

Claim 3.1. *Consider the jobs that are scheduled at t_0^-. Their total incomplete workload at t_0 is at most $aC_{max} + bB_{max}$.*

Proof. By Lemma 3.3, only non-preemptive sections of jobs in $\overline{\Psi}$ can block the execution of jobs in $\Psi \cup \{J_{k,l}\}$. The maximal workload of any job in $\Psi \cup \{J_{k,l}\}$ is bounded by C_{max}, and the number of such jobs scheduled at t_0^- is a. The total incomplete workload due to these jobs is upper bounded by $aC_{max} + bB_{max}$. ■

Let τ^* be the set of all tasks that have jobs in $\Psi \cup \{J_{k,l}\}$ that are pending but not ready at t_0^-.

Claim 3.2. *Consider the pending jobs in $\Psi \cup \{J_{k,l}\}$ that are not ready at t_0^-. Their total workload at t_0 is at most $\sum_{\tau_i \in \tau^*}(u_i x + 2C_i)$.*

Proof. Let s_i be the number of jobs of a task $\tau_i \in \tau^*$ that are pending at t_0^-. By the definition of τ^*, some jobs of τ_i are pending but not ready at t_0^-. Thus, certain preceding jobs of τ_i are incomplete at t_0^-. By the definition of P_i and job readiness, the first P_i pending jobs of τ_i are ready, because P_i jobs of τ_i can be scheduled in parallel. Thus, $s_i > P_i$. Note that the first P_i of these jobs are scheduled at t_0^- (t_0^- is a non-busy instant). Let $J_{i,j}$ be the earliest pending job of τ_i at t_0^-. Then $J_{i,j}$ is ready at t_0^-, and $J_{i,j} \neq J_{k,l}$, or else $\tau_k \notin \tau^*$ (as all pending jobs of τ_k in $\Psi \cup \{J_{k,l}\}$ would be ready). Thus, $J_{i,j} \in \Psi$. Also, because s_i jobs of τ_i are pending at t_0^-,

$$r_{i,j} \leq t_0^- - (s_i - 1)T_i. \tag{3.3}$$

Since $J_{i,j} \in \Psi$, it is completed by time $r_{i,j} + x + T_i + C_i$. Because $J_{i,j}$ is pending at t_0^-, $r_{i,j} + x + T_i + C_i \geq t_0^-$, or

$$r_{i,j} \geq t_0^- - x - C_i - T_i. \tag{3.4}$$

Combining (3.3) and (3.4), $t_0^- - x - C_i - T_i \leq t_0^- - (s_i - 1)T_i$, which implies $s_i - 2 \leq (x + C_i)/T_i$, which in turn implies

$$s_i \leq x/T_i + u_i + 2. \tag{3.5}$$

As the first P_i pending jobs of τ_i at t_0^- are ready, the total workload at t_0 of the jobs of τ_i pending but not ready at t_0^- is

$$\begin{aligned}
(s_i - P_i)C_i &\leq \{\text{by (3.5)}\} \\
&\quad (x/T_i + u_i + 2 - P_i)C_i \\
&= \{C_i/T_i = u_i\} \\
&\quad u_i x + 2C_i + (u_i - P_i)C_i
\end{aligned}$$

$$\leq \{\tau \text{ is feasible, so by (3.1), } u_i \leq P_i\}$$

$$u_i x + 2C_i.$$

Combining over all tasks in τ^*, we have a total workload of at most $\sum_{\tau_i \in \tau^*} (u_i x + 2C_i)$, as claimed. ■

Claim 3.3. *Consider the jobs in $\Psi \cup \{J_{k,l}\}$ that are not released at t_0^-. Their total generated workload over $[t_0, t_d)$ is at most $U(t_d - t_0)$.*

Proof. All jobs in $\Psi \cup \{J_{k,l}\}$ have deadlines at or before t_d. The jobs of a task τ_i with releases and deadlines within $[t_0, t_d)$ generate a workload of at most $\lfloor (t_d - t_0)/T_i \rfloor C_i \leq u_i(t_d - t_0)$. Summing over all such jobs of all tasks in τ yields the claim. ■

Claim 3.4. *The workload completed in $[t_0, t_d)$ is $m(t_d - t_0)$.*

Proof. By Definition 3.4, $t_0 \leq t_d$ and $[t_0, t_d)$ is a busy interval, so the total completed workload is $m(t_d - t_0)$. ■

Now we can finally bound W_d:

$$\begin{aligned}
W_d = \quad & \text{Workload at } t_0 \text{ of jobs scheduled at } t_0^- \\
& + \text{Workload at } t_0 \text{ of jobs pending but not ready at } t_0^- \\
& + \text{Workload at } t_d \text{ of jobs released after } t_0^- \\
& - \text{Workload completed within } [t_0, t_d) \\
\leq \ & \{\text{by Claims 3.1-3.4}\} \\
& aC_{max} + bB_{max} + \sum_{\tau_i \in \tau^*} (u_i x + 2C_i) \\
& + U(t_d - t_0) - m(t_d - t_0) \\
\leq \ & \{\tau \text{ is feasible, so } U \leq m\} \\
& aC_{max} + bB_{max} + \sum_{\tau_i \in \tau^*} (u_i x + 2C_i) \qquad (3.6)
\end{aligned}$$

Note that, by the definition of t_0^-, at least one CPU is not occupied with a job from $\Psi \cup \{J_{k,l}\}$ at t_0^-, so $a \leq (m-1)$. Additionally, the total number of scheduled jobs at t_0^- cannot exceed m, so $a + b \leq m$. Thus,

because $B_{max} \leq C_{max}$, we have

$$aC_{max} + bB_{max} \leq (m-1)C_{max} + B_{max}. \tag{3.7}$$

Also, any task $\tau_i \in \tau^*$ has exactly P_i ready jobs scheduled at t_0^-, while their total number is at most $a \leq m-1$. Thus,

$$\sum_{\tau_i \in \tau^*} P_i \leq m-1. \tag{3.8}$$

Combining (3.6), (3.7) and (3.8), and recalling (3.2), we get

$$W_d \leq (m-1)C_{max} + B_{max} + \sum_{\tau_i \in \tau^*} (u_i x + 2C_i) \leq L(x). \qquad \square$$

The obtained upper bound on $L(x)$ lets us define the sufficient value of x to establish the response-time bound of $x + T_k + C_k$ for the job of interest $J_{k,l}$.

Lemma 3.5. *If t_d is a busy time instant, and the response time of each job $J_{i,j} \in \Psi$ is at most $x + T_i + C_i$, where*

$$mx \geq L(x), \tag{3.9}$$

then the response time of $J_{k,l}$ is bounded by $x + T_k + C_k$.

Proof. Note that under G-EDF, $J_{k,l}$ cannot be preempted after its deadline t_d (which is T_k time units after $J_{k,l}$'s release). Thus, it is enough to prove that $J_{k,l}$ is scheduled at some point within $[t_d, t_d + x]$.

Let t_{av} ("av" means a CPU is available—see Figure 3.2(b)) denote the first time instant after t_d such that some CPU exists that is not executing a job in $\Psi \cup \{J_{k,l}\}$ or any non-preemptive section of a job in $\overline{\Psi}$ that is scheduled at time t_0 (and hence executes continually in $[t_0, t_{av}]$). Note that $t_b \leq t_{av}$. We consider three cases, depending on how much CPU allocation $J_{k,l}$ receives within $[t_0, t_{av})$.

Case 1. $J_{k,l}$ is completed before t_{av}.

In this case, the response time of $J_{k,l}$ is bounded by $t_{av} - r_{k,l} = t_{av} - t_d + T_k$.

$$\begin{aligned} t_{av} - t_d &\leq \{W_d \text{ is scheduled on } m \text{ CPUs at least over the interval } [t_d, t_{av})\} \\ &\quad W_d/m \\ &\leq \{\text{by Lemma 3.4}\} \end{aligned}$$

$$
\begin{aligned}
& L(x)/m \\
\leq\ & \{\text{by (3.9)}\} \\
& mx/m \\
=\ & x.
\end{aligned}
$$

This ensures a response-time bound of $x + T_k + C_k$ for $J_{k,l}$.

Case 2. $J_{k,l}$ is ready at t_{av}.

Let δ denote the remaining amount of execution for $J_{k,l}$ at t_{av}. Because the total remaining workload from jobs in $\Psi \cup \{J_{k,l}\}$ at t_d is W_d, at most $W_d - \delta$ of this workload can be completed within $[t_d, t_{av})$. Hence, $t_{av} - t_d \leq (W_d - \delta)/m$. By Lemma 3.4, $W_d \leq L(x)$, so $t_{av} - t_d \leq (L(x) - \delta)/m$. By Lemma 3.3, $J_{k,l}$ cannot be blocked by jobs or non-preemptive sections that do not contribute to W_d, so $J_{k,l}$ is scheduled in $[t_{av}, t_{av} + \delta)$, and $t_{av} + \delta - r_{k,l} = t_{av} + \delta - t_d + T_k$ is the response time of $J_{k,l}$. Because

$$
\begin{aligned}
t_{av} - t_d + \delta + T_k &\leq (L(x) - \delta)/m + \delta + T_k \\
&= L(x)/m + \delta(1 - 1/m) + T_k \\
&\leq \{\text{by (3.9)}\} \\
&\quad mx/m + \delta(1 - 1/m) + T_k \\
&\leq \{\delta \leq C_k\} \\
&\quad x + C_k + T_k,
\end{aligned}
$$

the response time of $J_{k,l}$ is at most $x + T_k + C_k$.

Case 3. $J_{k,l}$ is not ready at t_{av}.

In this case, $J_{k,l-P_k}$ (which is in $\Psi \cup \{J_{k,l}\}$) is not finished by t_{av}. This predecessor is released at the latest by time $t_d - (P_k + 1) \cdot T_k$. By the lemma statement, $J_{k,l-P_k}$ completes at the latest by $t_d - (P_k + 1) \cdot T_k + x + T_k + C_k = t_d + x - P_k \cdot T_k + C_k$. By (3.1), $C_k \leq P_k \cdot T_k$, so $J_{k,l}$ is ready at the latest by $t_d + x$. By Lemma 3.3, $J_{k,l}$ is not blocked by any job at t_{av}, because $t_b \leq t_{av}$. That ensures the response-time bound. □

We now can conclude both the busy and the non-busy t_d cases in the following theorem.

Theorem 3.1. *Every job $J_{i,j}$ of every task $\tau_i \in \tau$ completes within $x + T_i + C_i$ time units after its release for any $x > 0$ such that x satisfies (3.9).*

Proof. Follows by induction over $\prec$, applying Lemma 3.2 or Lemma 3.5. □

Early releasing. Early releasing does not affect the response-time analysis for rp-sporadic tasks because every job's actual release time and deadline (used in $\prec$ to define job priorities) are unaltered by early releasing. Thus, $\Psi \cup \{J_{k,l}\}$ is unaffected, as well as the upper bound on its demand $L(x)$. Therefore, the actual response-time bound $x + T_k + C_k$ is unaffected.

3.3 Closed Form Bound

We now introduce some terminology that is used in obtaining a closed-form expression for x that is relevant in the context of graph-based tasks (*e.g.,* the graph model we consider uses the same parallelization level for all their nodes).

Definition 3.5. We call a task τ_i *p-restricted* (parallelism-restricted) if $P_i < m$, and *non-p-restricted* if $P_i = m$. Also, let

$$U^b_{res} = \sum_{\substack{\text{b largest values} \\ \tau_i \text{ is p-restricted}}} u_i \qquad \text{and} \qquad C^b_{res} = \sum_{\substack{\text{b largest values} \\ \tau_i \text{ is p-restricted}}} C_i,$$

and let $U_{res} = U^n_{res}$ (the total utilization of all p-restricted tasks) and $C_{res} = C^n_{res}$ (the sum of WCETs of all p-restricted tasks). We assume $U^b_{res}, U_{res}, C^b_{res}, C_{res}$ to be 0 if no p-restricted tasks are present in the system.

Corollary 3.1. *The response time of any task $\tau_i \in \tau$ is bounded by $x + T_i + C_i$, where*

$$x = \frac{(m-1)C_{max} + B_{max} + 2C_{res}}{m - U_{res}}. \tag{3.10}$$

Furthermore, if there exists $P_{min} \geq 1$ such that for every p-restricted task τ_i, $P_i \geq P_{min}$, then U_{res} and C_{res} in (3.10) can be replaced with U^ℓ_{res} and C^ℓ_{res}, where $\ell = \lfloor (m-1)/P_{min} \rfloor$.

Proof. Note that the task subset τ^* in (3.2) consists of only p-restricted tasks, because $\sum_{\tau_i \in \tau^*} P_i \leq m-1$ (see (3.8)), while $P_i = m$ for any non-p-restricted task. Thus,

$$\begin{aligned}
&\max_{\substack{\tau^* \subseteq \tau \text{ s.t.} \\ \sum_{\tau_i \in \tau^*} P_i \leq m-1}} \left(\sum_{\tau_i \in \tau^*} (u_i x + 2C_i) \right) \\
&= \max_{\substack{\tau^* \text{ consists of p-restricted tasks} \\ \sum_{\tau_i \in \tau^*} P_i \leq m-1}} \left(\sum_{\tau_i \in \tau^*} (u_i x + 2C_i) \right) \\
&\leq \sum_{\tau_i \text{ is a p-restricted task}} (u_i x + 2C_i) \\
&= U_{res} x + 2C_{res}.
\end{aligned}$$

Hence, by (3.2), $L(x) \leq (m-1)C_{max} + B_{max} + U_{res}x + 2C_{res}$. Because, by (3.10), $mx = (m-1)C_{max} + B_{max} + U_{res}x + 2C_{res} \geq L(x)$, x as defined in (3.10) satisfies (3.9). Therefore, by Theorem 3.1, $x + T_i + C_i$ is a response-time bound for any task τ_i.

If for every p-restricted task τ_i, $P_i \geq P_{min}$, then, because $\sum_{\tau_i \in \tau^*} P_i \leq m-1$, $|\tau^*| \leq (m-1)/P_{min}$. Because $|\tau^*|$ is integer, $|\tau^*| \leq \lfloor (m-1)/P_{min} \rfloor$. In this case, only the $\lfloor (m-1)/P_{min} \rfloor$ p-restricted tasks with the highest corresponding values have to be considered in U_{res} and C_{res}. □

3.4 Improved Bounds

The basic bound derived in Section 3.2 can be improved via several techniques that we omitted above to simplify the analysis. We describe these techniques in this section.

Accurate accounting of ready jobs. In Claim 3.1 of Lemma 3.4, we bounded the maximal workload of any ready job at t_0^- as C_{max}. However, this could be reduced with a more precise accounting of ready jobs, yielding a small improvement for task sets for which the highest-WCET tasks are p-restricted.

Busy time instant. Let $m^+ = \lceil U \rceil$. If $U \leq m-1$, then $m^+ < m$; otherwise $m^+ = m$. This definition was used in (Devi and Anderson, 2008; Erickson and Anderson, 2011). Under this approach the definition of a busy instant (Definition 3.3) is modified in the following way: a time instant is busy if at least m^+ jobs in $\Psi \cup \{J_{k,l}\}$ are scheduled, or there is a non-scheduled ready job in $\Psi \cup \{J_{k,l}\}$. This approach results in the

What Changed	Basic	Improved (Reduced)
(3.7)	$(m-1)C_{max}+B_{max}$	$(m^+-1)C_{max}+(m-m^++1)B_{max}$
(3.8)	$\sum_{\tau_i\in\tau^*} P_i \leq m-1$	$\sum_{\tau_i\in\tau^*} P_i \leq m^+-1$
ℓ of Corollary 3.1	$\ell = \lfloor (m-1)/P_{min} \rfloor$	$\ell = \lfloor (m^+-1)/P_{min} \rfloor$
(3.10)	$x = \frac{(m-1)C_{max}+B_{max}+2C^\ell_{res}}{m-U^\ell_{res}}$	$x = \frac{(m^+-1)C_{max}+(m-m^++1)B_{max}+2C^\ell_{res}}{m-U^\ell_{res}}$

Table 3.1: Bound improvement due to change in Definition 3.3.

changes that are found in Table 3.1. These changes yield a significant bound improvement for low-utilization task sets.

Compliant-vector analysis. We considered every task to have the same value x. We could instead apply compliant-vector analysis (Erickson et al., 2010; Erickson and Anderson, 2011), which assigns a distinct x_i to each task τ_i. This approach yields lower response-time bounds.

3.5 Chapter Summary

In this chapter, we have considered G-EDF scheduling of rp-sporadic tasks with non-preemptive sections on an identical multiprocessor. In doing so, a necessary and sufficient feasibility condition for rp-sporadic tasks was derived. An upper bound on the workload competing with the task of interest was proven. From this upper bound, we derived a response-time bound for feasible tasks sets. We computed a closed form of the bound, and proposed a few ways to improve it.

Acknowledgements. The initial work on the rp-sporadic task model presented in Section 3.2 and the bounds listed in this chapter were the result of a collaboration between Tanya Amert and Sergey Voronov (Amert et al., 2019). Voronov derived the bounds and provided an initial implementation of the graph-generation scripts, and Amert worked on the implementation of a real application to show the applicability of the rp-sporadic model to real applications (some applications allow various parallelization levels).

CHAPTER 4: SRT GRAPH SCHEDULING[1]

In this chapter, we describe how to compute a response-time bound for an SRT graph task on an identical multiprocessor; the considered approach was briefly outlined in Section 2.5.2. We call this approach the *offset-based* approach. The offset-based approach is used widely in SRT graph research.

We first present the approach itself in Section 4.1 with an example. Then, we discuss its drawbacks in Section 4.2. Later, we propose node merging with heuristics as a method to address the discussed drawbacks in Section 4.3. We present an experimental evaluation of the proposed merging heuristics and conclude in Sections 4.4 and 4.5, respectively.

4.1 The Offset-Based Approach

Historically, a typical non-graph SRT response-time analysis produces *offset-independent* bounds: they do not change significantly (or at all) when the releases of all jobs of a task τ_i are shifted by some constant Φ_i (called the *offset* of τ_i). Examples include (Devi and Anderson, 2005; Leontyev and Anderson, 2010; Elliott et al., 2014; Yang and Anderson, 2017; Yang et al., 2018). The main idea behind the SRT graph scheduling approach can be formulated as follows:

Regular dependencies among graph nodes can be satisfied with a special choice of task offsets.

A similar idea is used in HRT scheduling to satisfy sporadic graph/node dependencies (achieved through a choice of node/graph deadline model; see details in Section 2.5.1).

The approach was proposed by Liu and Anderson (2010) for sporadic DAG tasks, and later extended by Yang *et al*. (2015) to support sporadic graph tasks with delay dependencies. The SRT approach includes three main steps (shown in Figure 2.9).

1. Convert graph tasks to DAG tasks (independently).

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

Voronov, S., Tang, S., Amert, T., and Anderson, J. H. (2021b). AI meets real-time: Addressing real-world complexities in graph response-time analysis. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 82–96.

2. Convert DAG tasks into a set of rp-sporadic tasks with offsets (offsets are determined using per-task response-time bounds).

   [combine rp-sporadic task sets from different DAGs into a single task set]

3. Compute per-graph response-time bounds (independently).

Step 1 considers a single graph task conversion. Multiple graphs are converted independently; their rp-sporadic sets are merged together before computing bounds. Note that we use the rp-sporadic task model as the sequential task model of graph nodes. However, the sporadic or npc-sporadic task models can be used instead if the results are acceptable.

We explain each of the three steps of computing graph response-time bounds in detail, and illustrate the process with an example using a single graph task G defined in Example 2.6 and shown in Figure 2.7a.

4.1.1 Step 1: Convert a Graph into a DAG

This step was proposed by Yang *et al*. (2015). The authors provided a set of rules for converting a task graph into a sporadic DAG task (which requires eliminating delay dependencies; see Section 2.4.1 for definitions).

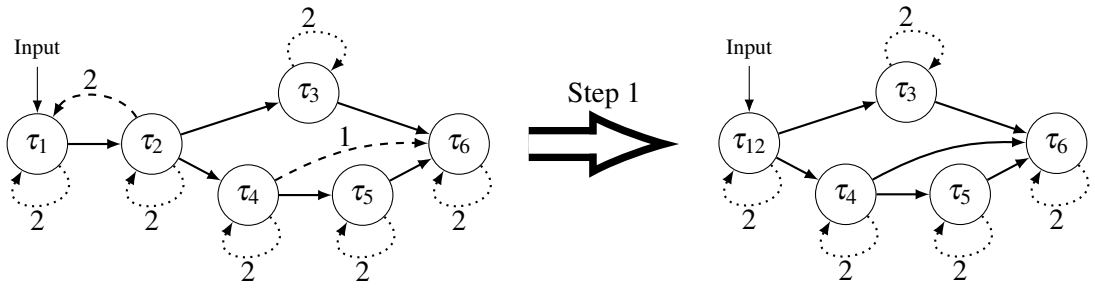
- A forward dependency $\tau_a \to \tau_b$ of any level should be replaced with a regular dependency $\tau_a \to \tau_b$.
- A backward dependency $\tau_a \to \tau_b$ of any level should be replaced by a regular dependency $\tau_a \to \tau_b$. The resulting cycle in the graph of regular dependencies should be combined into a single supernode with a WCET identical to the sum of the WCETs of the replaced nodes.

Yang *et al*. assumed that each sporadic dependency has a level of one (*i.e.,* the sporadic task model). Under this assumption, the utilization of a supernode may easily exceed 1.0, making the system unschedulable. We consider the rp-sporadic task model; the parallelization level of the supernode is defined as

$$P_{super} = \min(\text{parallelization level of graph}, \text{level of backward dependency}),$$

which is may be greater than 1.0.

Example 4.1 (see Figure 4.1). Consider the graph task G defined in Example 2.6 and shown in Figure 4.1a. G has two delay dependencies: $\tau_4 \to \tau_6$ (forward) and $\tau_2 \to \tau_1$ (backward). The corresponding DAG task G'



(a) Graph task (b) DAG task

Figure 4.1: The transformation of the graph task G into the DAG task G'.

is shown in Figure 4.1b. By the transformation rules, $\tau_4 \to \tau_6$ is replaced by a regular dependency $\tau_4 \to \tau_6$, and nodes τ_1 and τ_2 are merged into a single node τ_{12} to eliminate $\tau_2 \to \tau_1$. $\Diamond$

4.1.2 Step 2: Convert DAGs into a Set of Tasks

Consider the DAG task G' obtained in Step 1. Liu and Anderson (2010) proposed to schedule jobs produced by nodes of G' as jobs of independent sporadic tasks with some offsets (*i.e.,* they did not consider any delay dependencies of G); each independent task τ_i shares the graph period T_G and has an offset Φ_i.

Let $\tau_{G'}$ be the set of nodes of G' with regular dependencies removed, preserving WCETs, releases, deadlines, and parallelization level with period equal to $T_{G'} = T_G$. Thus, $\tau_{G'}$ is a set of independent rp-sporadic tasks. Let

$$\tau = \bigcup_{\text{all } G' \text{ in system}} \tau_{G'},$$

and R_i be the response-time bound of $\tau_i \in \tau$ obtained by applying an SRT response-time analysis to τ. We use the analysis from Chapter 3, but any offset-independent analysis can be used. Define a recursive function (assuming $\max(\emptyset)$ is zero)

$$\Phi_i = \max_{\substack{\tau_j \to \tau_i \text{ is a regular} \\ \text{dependency in } G'}} (\Phi_j + R_j). \tag{4.1}$$

Since G' is a DAG task, its regular dependencies have no cycles. Thus, the values Φ_i are computable (*e.g.,* in topological order over G'). Liu and Anderson (2010) showed that (4.1) ensures the satisfaction of the regular dependencies of G; the sporadic dependencies are preserved by the task model. We provide a sketch of the proof.

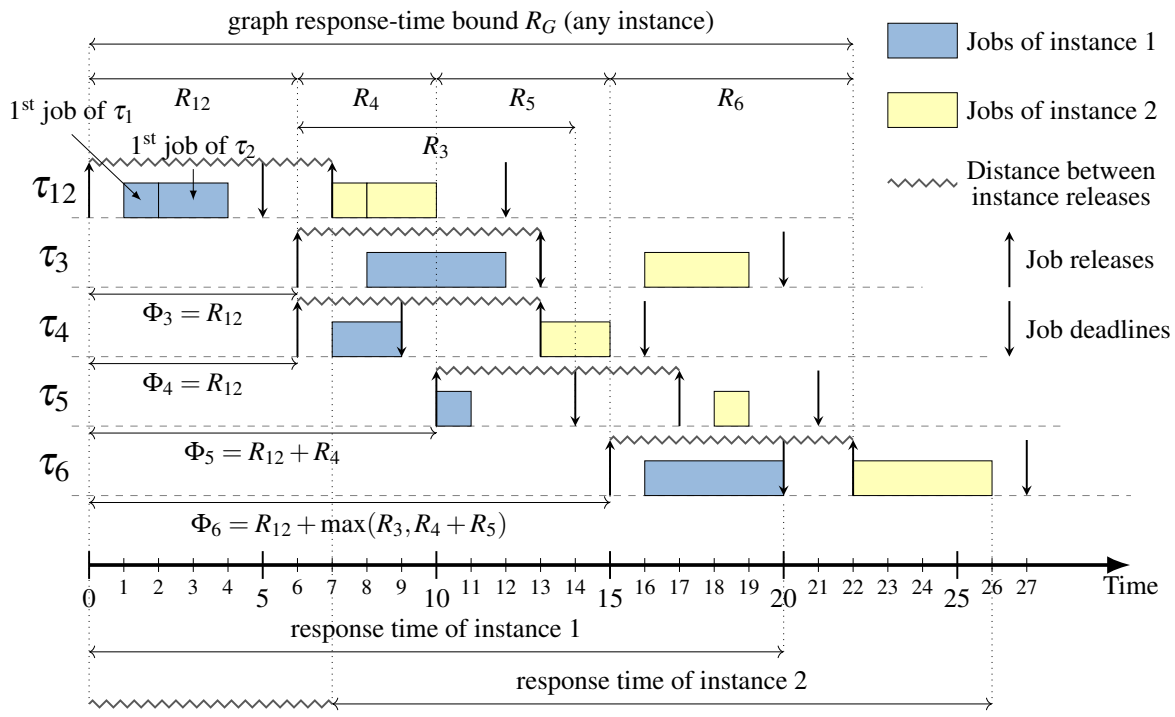


Figure 4.2: Schedule of the first and second instances of DAG task G' under offset-based approach.

The bounds of the offset-independent analysis are independent of the offsets. Thus, Φ_i can be viewed as constants. Consider the first instance of the DAG G' and a topological ordering (Kahn, 1962) of its nodes. Then the first node τ_1 has no dependencies and starts at 0. All nodes that depend on τ_1 have $\Phi_i \geq R_1$ (by 4.1). Thus, these nodes are released after the completion of the first job of τ_1 (by the definition of R_1—the response-time bound of τ_1). This reasoning can be extended by the principle of induction, first to all nodes of the first instance of G', second to all instances of G', and finally to all graphs in the system.

Example 4.1 (cont'd, see Figure 4.2). DAG task G' (depicted in Figure 4.1b) is converted into the set of independent tasks $\tau = \{\tau_{12}, \tau_3, \tau_4, \tau_5, \tau_6\}$ in Step 2. The schedule of jobs of the first two instances of G is shown in Figure 4.2. The first jobs of $\tau_{12}, \tau_3, \tau_4, \tau_6$ and the second jobs of τ_3, τ_5 are not scheduled immediately after their releases because of other workloads unrelated to G' in the system. Note that jobs of τ_4 are completed long before releases of jobs of τ_5 due to the offsets. ◊

4.1.3 Step 3: Compute Response-Time Bound

Assuming an offset of zero for the tasks with no incoming regular dependencies, $\Phi_i + R_i$ provides a response-time bound for any DAG node τ_i. Then the response-time bound R_G of DAG task G' can be

computed as the maximal bound over all its nodes

$$R_G = \max_{\tau_i \in G'}(\Phi_i + R_i). \tag{4.2}$$

The same response-time bound applies to the initial graph task G.

Example 4.1 (cont'd). Recall that G' is depicted in Figure 4.1b. τ_{12} has no incoming dependencies, so $\Phi_{12} = 0$. Both τ_3 and τ_4 depend only on τ_{12}, so $\Phi_3 = \Phi_4 = R_{12} + \Phi_{12} = R_{12}$. τ_5 depends only on τ_4, so $\Phi_5 = \Phi_4 + R_4 = R_{12} + R_4$. τ_6 depends on τ_3 and τ_5, so $\Phi_6 = \max(\Phi_3 + R_3, \Phi_5 + R_5) = \max(R_{12} + R_3, R_{12} + R_4 + R_5) = R_{12} + \max(R_3, R_4 + R_5)$. These offsets are shown in Figure 4.2. $\Diamond$

4.2 Approach Drawbacks

As described below, the standard approach to obtain SRT response-time bounds for a graph is to analytically convert the graph into a set of sporadic tasks and define per-task offsets. Despite being a standard method, this approach has two major drawbacks.

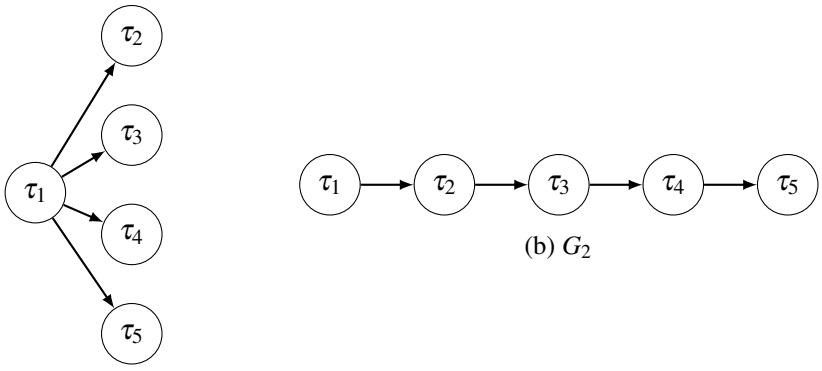
Actual schedule modification. The proposed approach modifies the actual schedule of the initial system (introducing per-task offsets). Such a modification can negatively affect the actual system performance by postponing releases of jobs, causing system idleness. Fortunately, if the analysis used to compute R_i supports early releasing (see Section 2.5.2), offset-related release changes can be ignored. However, offsets are still used to properly define jobs' deadlines (*i.e.,* job priorities under G-EDF). Thus, offsets change G-EDF prioritization but not the tasks themselves. At the same time, supernodes change how the tasks are executed (*i.e.,* tasks within are forced to be scheduled sequentially).

Poor bound scaling. Under the offset-based approach, the graph response-time bound is computed as

$$R_{G'} = \max_{\text{path of } G'} \sum_{\tau_i \in \text{path}} R_i, \tag{4.3}$$

using per-task response-time bounds R_i. Thus, two graphs with the same set of nodes may have significantly different response-time bounds. We illustrate this with the following example.

Example 4.2 (see Figure 4.3). Consider two task systems τ^1 and τ^2. Each of them has a single graph G_i that contains the same five nodes $\tau_1, ..., \tau_5$, where the regular dependencies are the only difference. Then



(a) G_1 (b) G_2

Figure 4.3: Two graphs types.

$R_{G_1} = R_1 + \max(R_2, R_3, R_4, R_5)$, while $R_{G_2} = R_1 + R_2 + R_3 + R_4 + R_5$. Because the nodes of both graphs are identical, the per-task bounds are identical. Thus, the second graph may have a significantly larger response-time bound. ◊

If we consider a random graph G, R_G will be proportional to the length of *the longest graph path*. Thus, graphs with many nodes generally have higher response-time bounds (because R_i is lower bounded by the graph period under any SRT analysis). As we can see, the obtained response-time bound may scale poorly with the graph size. We propose a solution to this problem in the next section using node *merging*.

4.3 Node Merging

The graph response-time bound under the offset-based approach is defined by (4.3). The main source of pessimism in it is the general looseness of the per-task response-time bounds. Most papers deriving SRT response-time bounds for non-graph tasks $\tau = \{\tau_i\}$ (Devi and Anderson, 2005; Erickson et al., 2010; Leontyev and Anderson, 2010; Erickson and Anderson, 2011; Tong and Liu, 2015; Yang and Anderson, 2017; Amert et al., 2019) have $R_i = C_i + T_i + x_i$ with $x_i \geq 0$ (x_i may or may not be the same for all tasks). Let *length* of a path in a graph G be the sum of R_i over the nodes of the path. Then merging two nodes in the longest path reduces R_G by about T_i time units, if other paths have smaller lengths and R_i does not change significantly due to the merging. We illustrate this with the following example.

Example 4.3 (see Figure 4.4). Consider a simple graph G depicted in Figure 4.4a; its response-time bound is $R_1 + R_2 + R_3$. Assume that τ_2 and τ_3 have small utilizations, so $u_2 + u_3$ does not exceed the upper limit on a node's utilization (which may depend on the task model). Suppose we merge tasks τ_2 and τ_3 (which forces

(a) Intial graph. (b) Nodes τ_2, τ_3 merged.

Figure 4.4: Node merging example.

jobs of τ_2 and τ_3 to be always scheduled one after another) into a single task τ_{23} (see Figure 4.4b for the resulting graph). Typically, R_2 and R_3 are smaller than R_{23} but $R_2 + R_3$ is bigger than R_{23}, so we effectively reduce the response-time bound of the whole graph from $R_1 + R_2 + R_3$ to $R_1 + R_{23}$ (which may be smaller by up to T_G). $\Diamond$

From the above discussion, it may seem desirable to merge many nodes into one (*e.g.,* merge all nodes of the DAG into one node). However, the ordinary sporadic task model limits merging possibilities: for the system to remain schedulable, the total utilization of a supernode must be at most 1.0. Fortunately, the rp-sporadic task model we use relaxes the utilization constraints: the total utilization of a supernode is limited by the graph parallelization level.

4.3.1 Detailed Merging Process

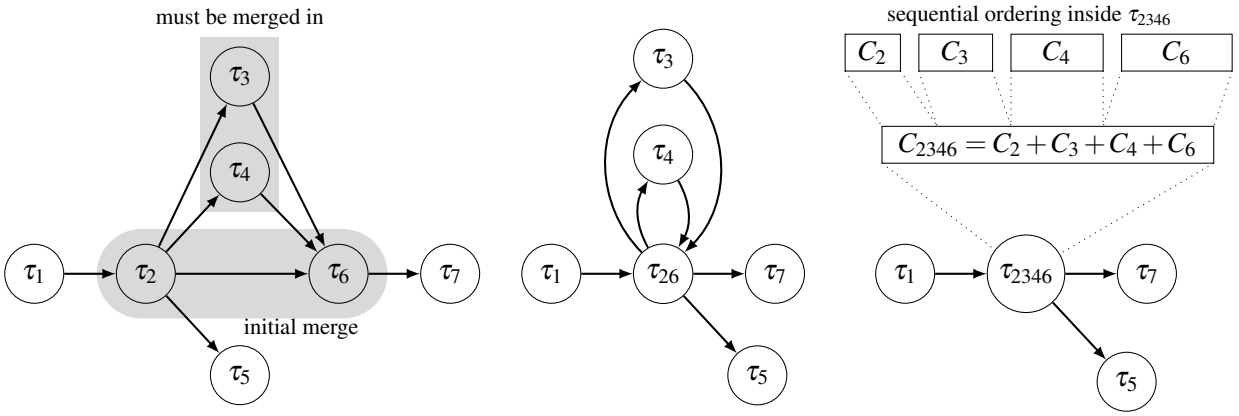
To explain in detail the merge-related problems, we first formally define the merging process. We want to emphasize that node merging is an *offline* bound reduction technique and does not require significant changes at runtime.

Definition 4.1. Let $A = \{\tau_{a_1}, ... \tau_{a_s}\}$ be the set of nodes of the graph G we want to merge (we assume they share the graph period T_G). Parameters of τ_{super} are defined as follows:

$$C_{super} = \sum_{\tau_{a_i} \in A} C_{a_i},\ U_{super} = \sum_{\tau_{a_i} \in A} u_{a_i},\ P_{super} = \min_{\tau_{a_i} \in A} P_{a_i}.$$

We replace nodes in A with a single supernode τ_{super} in G. The workload of nodes in A is scheduled sequentially inside τ_{super} (node-by-node in some order). To preserve dependencies in G, for any node τ_x that has an edge $\tau_x \to \tau_{a_i}$ (resp., $\tau_{a_i} \to \tau_x$), we replace it with an edge $\tau_x \to \tau_{super}$ (resp., $\tau_{super} \to \tau_x$).

Unfortunately, the node merging process may be more complicated than the merging provided in Example 4.3. For example, the definition above does not specify the node ordering inside τ_{super}. Moreover, cycles can be created in the graph when merging without a careful consideration of the nodes to be merged. We illustrate these issues with the following example.



(a) Intial graph. (b) τ_2 and τ_6 merged. (c) $\tau_2, \tau_3, \tau_4, \tau_6$ merged.

Figure 4.5: Merging additional nodes.

Example 4.4 (see Figure 4.5). Consider a graph G depicted in Figure 4.5a. Assume we want to merge nodes τ_2 and τ_6 (*e.g.,* we want to check the response-time bound of the system after the merge).

Consider the case where we merge *only* τ_2 and τ_6 (*i.e.,* $A = \{\tau_2, \tau_6\}$); we call τ_{super} as τ_{26} in this case. The obtained graph G' is shown in Figure 4.5b. As graph regular dependencies have to be preserved, and G has an edge $\tau_2 \to \tau_3$, G' has edge $\tau_{26} \to \tau_3$. At the same time G has an edge $\tau_3 \to \tau_6$, so G' has edge $\tau_3 \to \tau_{26}$. Thus G' has circular dependency and cannot be scheduled (without extra information about dependencies in G).

Note that all cycles in Figure 4.5b ($\tau_{26} \rightleftarrows \tau_3$ and $\tau_{26} \rightleftarrows \tau_4$) contain nodes laying on some path $\tau_2 \to ... \to \tau_6$. Thus, an example solution for this problem (shown in Figure 4.5c) is to additionally merge in all nodes on any path $\tau_2 \to ... \to \tau_6$. In the case of G, these paths are $\tau_2 \to \tau_3 \to \tau_6$ and $\tau_2 \to \tau_4 \to \tau_6$. Thus, the merging of two nodes τ_2, τ_6 actually leads to the merging of at least $\{\tau_2, \tau_3, \tau_4, \tau_6\}$ with supernode $\tau_{super} = \tau_{2346}$. The sequential ordering of merged tasks inside the supernode is also shown in Figure 4.5c. $\Diamond$

The solution for the example above can be generalized into the following lemma.

Lemma 4.1. *To merge nodes τ_a and τ_b in a DAG G, all nodes laying on any path $\tau_a \to ... \to \tau_b$ or $\tau_b \to ... \to \tau_a$ must be included into the supernode τ_{super} to keep the graph acyclic. The merged nodes can be serialized inside τ_{super} preserving their dependencies.*

Proof. Consider a node τ_w that lies on path $\tau_a \to ... \to \tau_w \to ... \to \tau_b$. If τ_w is not included into τ_{super}, in the graph after merging there is a path $\tau_{super} \to ... \to \tau_w$ (because of the path $\tau_a \to ... \to \tau_w$), and a path $\tau_w \to ... \to \tau_{super}$ (because of the path $\tau_w \to ... \to \tau_b$). Thus, the graph after merging has a cycle, and τ_w must

be included into τ_{super}. Note that all graph dependencies are preserved by the definition of node merging: τ_{super} has all dependencies of merged nodes as its own dependencies .

To serialize nodes inside τ_{super}, we consider a topological ordering $\mathbb{T}$ of the nodes of G. For example, $(\tau_1, \tau_2, \tau_3, \tau_4, \tau_6, \tau_7, \tau_5)$ and $(\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7)$ are valid topological orderings of G. By the definition of $\mathbb{T}$, if τ_x appears in $\mathbb{T}$ before the node τ_y, no path $\tau_y \to ... \to \tau_x$ exists in G. At least one $\mathbb{T}$ always exists in a graph without cycles (can be found with a single depth-first search). We serialize the execution of the merged nodes inside τ_{super} with respect to $\mathbb{T}$ (*e.g.,*$(\tau_2, \tau_3, \tau_4, \tau_6)$). Thus, no jobs inside τ_{super} can be started before the completion of their dependencies of the same graph instances. □

The lemma above specifies the full merging set for any pair of nodes we want to merge. Thus, from the graph point of view, any pair of nodes can be merged. However, the excessive merging is bounded by the system schedulability condition:

$$U_{super} \leq P_{super},$$

where U_{super} may be large for some pairs of nodes (*e.g.,* the merging of τ_1 and τ_7 in Figure 4.5a leads to a merging of six nodes $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_6, \tau_7\}$).

Existing work on node merging. Task/node merging has been considered before, but mostly for reasons orthogonal to the response-time bound reduction, such as to reduce task-to-task communication costs (Aronsson and Fritzson, 2003; Zhou et al., 2017; Faragardi et al., 2014), enable task clustering (Ebaid et al., 2010; Bhuiyan et al., 2018; Danne and Platzner, 2005; Xia et al., 2010), or reduce energy consumption (Qiu et al., 2006; Rong and Pedram, 2008). Yang *et al*. (2015) used node merging to eliminate cycles. (Jiang et al., 2022a) used node merging to compare results with an older model (the new model considers node-shared critical sections, while the old model considers per-node critical sections).

In fact, with respect to response-time bounds and schedulability, the conventional wisdom seems to favor task splitting, not merging (Chen and Liu, 2014; Erickson and Anderson, 2013; Jiang et al., 2016, 2017; Soliman and Pellizzoni, 2019; Ferry et al., 2013; Júnior et al., 2013; Brandenburg and Gül, 2016).

An example without a polynomial solution. While node merging can decrease end-to-end response-time bounds, the choice of which nodes to merge to minimize the longest path is complex. We illustrate this complexity with an example.

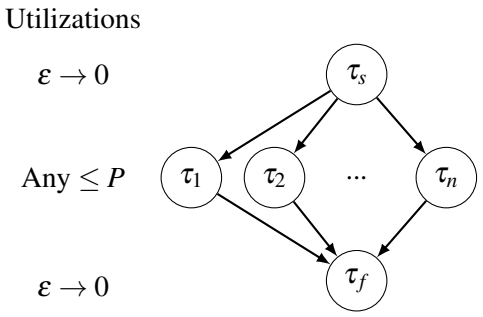


Figure 4.6: The shortest path elimination example.

Example 4.5 (see Figure 4.6). Consider a simple DAG task with nodes $\tau_s, \tau_1, ..., \tau_n, \tau_f$ shown in Figure 4.6 with two low-utilization nodes τ_s and τ_f. We assume that the graph parallelization level is P.

Let the total utilization of all graph nodes be in $(P, 2P]$. Such a graph cannot be merged into a single supernode due to the schedulability constraints. Thus, the shortest path in the graph is at least two. Such two-node lengths can be reached if some nodes of $\tau_1, ..., \tau_n$ are merged into τ_s, while other nodes are merged into τ_f.

Note that this is a 2-partitioning problem, which known to be NP-complete (Garey and Johnson, 1975).

$\Diamond$

In the example above, we assume that per-task response-time bounds do not change significantly after merging nodes. However, this assumption does not hold for some task sets; in this case, the problem becomes even harder.

4.3.2 Proposed Solution

The usual way to solve a problem that is unlikely to have a polynomial-time solution is to use heuristics. A base heuristic algorithm can be found in Listing 1. The base heuristic traverses over all pairs of nodes to merge, proposed by a heuristic, and merges them repeatedly until no new pair can be selected (*i.e.,* the response-time bound R would not decrease any further). The choice of pairs of nodes is heuristic-specific and described later.

Note that a typical task set contains multiple DAGs; we use the largest response-time bound of these DAGs as the system's response-time bound.

Listing 1 Base heuristic pseudocode.

```
b_prev step ← current system bound
b_found ← current system bound
while b_found ≤ b_prev step do
    foreach DAG G in the task set do
        foreach (τ1, τ2) ∈ ChosenNodePairs (G) do        ▷ Heuristic-specific function
            S ← nodes to merge with τ1 and τ2            ▷ According to Lemma 4.1
            P12 ← min_{τi∈S}(Pi)
            U12 ← Σ_{τi∈S}(ui)                           ▷ Total utilization of the merged nodes
            if U12 ≤ P12 then                            ▷ Nodes can be merged
                b_current ← response-time bound after merge
                b_found ← min(b_current, b_found)
            end
        end
    end
    merge the nodes leading to the bound of b_found
end
return b_prev step
```

heuristic	pairs of nodes to consider
LongestPath	all pairs of consecutive nodes in the longest path
ElementaryPair	all pairs of nodes in graph connected by an **edge**
BestPair	all pairs of nodes in graph connected by a **path**

Table 4.1: Heuristics

Proposed heuristics. We consider three heuristics: *LongestPath*, *ElementaryPair*, and *BestPair*. We assume that our task set has n nodes and E edges. Short descriptions are presented in Table 4.1; the implementations of `ChosenNodePairs` can be found in Listing 2.

LongestPath computes the longest path L in any graph of the system, weighting each of the n system nodes by its individual response-time bound R_i. Then, LongestPath traverses over pairs of consecutive nodes in L. Because our graphs are acyclic (Step 1 described in Section 4.1.1), the computation of L can be done with a single breadth-first search in $O(E)$ time.

ElementaryPair traverses over all pairs of nodes that are connected by a direct edge. It can also be done in $O(E)$ time. BestPair traverses over all pairs of nodes that are connected by any path (*i.e.,* are in the same "connected component"). It requires $O(n^2)$ time. ElementaryPair is a simplified version of BestPair.

Note that every merging in LongestPath requires to merge only two nodes. Assume $\tau_a \to \tau_b$ is an edge in the longest path L. If there is a path $\tau_a \to ... \to \tau_b$, then L is not the longest path. At the same time, a

Listing 2 `ChosenNodePairs` implementation for each heuristic.

Function `LongestPath_ChosenNodePairs(`*DAG* G`):`

```
ℓ_pairs ← empty list
topologically sort G                                   ▷ If G has an edge τ_i → τ_j then i < j
d ← (0,...,0)                                          ▷ The length of the longest path that ends in τ_i
par ← (0,...,0)                                        ▷ The parent of τ_i in the longest path that ends in τ_i
n ← the number of nodes in G
for i = 1..n do
    if τ_i has no incoming edges in G then
        d[i] ← R_i                                     ▷ only one-node path {τ_i} ends in τ_i
        par[i] ← 0
    foreach edge τ_i → τ_j in G do
        if d[j] < d[i] + R_j then                      ▷ try to improve the longest path ending in τ_j
            d[j] ← d[i] + R_j
            par[j] ← i
        end
    end
end
q ← arg max_{i=1..n}(d[i])                             ▷ q is the index of the last node of the longest path
path ← the longest path in G                           ▷ Reconstructed with par starting with q
ℓ_pairs ← all pairs of consecutive nodes in path
return ℓ_pairs
```

Function `BestPair_ChosenNodePairs(`*DAG* G`):`

```
ℓ_pairs ← empty list
foreach τ_i ∈ G do
    foreach τ_j ∈ G do
        Add pair (τ_i, τ_j) to ℓ_pairs
    end
end
return ℓ_pairs
```

Function `ElementaryPair_ChosenNodePairs(`*DAG* G`):`

```
ℓ_pairs ← empty list
foreach τ_i ∈ G do
    foreach τ_j ∈ G do
        if G has edge τ_i → τ_j then
            Add pair (τ_i, τ_j) to ℓ_pairs
        end
    end
end
return ℓ_pairs
```

merging in BestPair (and, sometimes in ElementaryPair) requires merging $O(n)$ nodes into a single node (*e.g.,* merging τ_1 and τ_7 in Figure 4.5 would result in the merging of 6 nodes into τ_{123467}).

4.4 Experimental Evaluation

We conducted several synthetic experiments to evaluate the efficacy of the proposed heuristics. In this section, we consider CPU-only graph workloads (CPU+HAC workloads are considered in Chapter 5).

Workload generation. A task set is a collection of several DAGs. Nodes of the same DAG represent tasks with regular dependencies defined by the edges. We assumed that nodes of the same DAG share the same period and parallelization level; their utilizations are generated using the Dirichlet-Rescale algorithm (Griffin et al., 2020). This algorithm generalizes UUnifast (Bini and Buttazzo, 2005) for multiprocessor task sets with various utilization bounds. The obtained utilization vectors are uniformly distributed in the available space.

To generate a DAG of k nodes we first generated a random tree over these nodes to make the DAG connected. To generate such a tree we used either the Barabási–Albert model (Barabási and Albert, 1999), or a randomly generated tree with a predefined longest path length (additional nodes are attached on random layers of the tree). The remaining edges were generated using the Erdős-Rényi model (Erdős and Rényi, 1959) (with a constant probability of 0.4). Each edge creates a dependency by making the lower-indexed task a predecessor of the higher-indexed task (to ensure that the generated graph is DAG).

We generated several independent DAGs for each task set. The number of nodes per DAG was generated with a multinomial distribution using the total number of nodes. For each figure, we generated 8,000 task sets (for each line in the following plots) with $P = 2$ unless stated otherwise.

Metrics. We measured the efficacy of the proposed solution with two metrics: the share of improved task sets (SITS) and the relative end-to-end bound (RB). SITS shows the percentage of task sets which get a response-time bound improvement of at least 50% (as in most of our experiments, the share of graphs with any bound improvement is close to 100%).

A task set that has no mergable nodes cannot get any improvement. At the same time, a graph such as that in Figure 4.3b with randomly generated utilization on average should get a bound improvement due to having few edges (hence, merging is not expected to merge extra nodes). RB measures how good the

response-time bound improvement is (lower is better):

$$\text{RB} = \frac{R_{fin}}{R_{init}},$$

where R_{init} is the initial graph response-time bound, and R_{fin} is the obtained response-time bound after node merging.

4.4.1 Early-Stop

The first experiment considered a small modification to the base heuristic. The base heuristic (see Listing 1) terminates at the first merging step that does not lead to an immediate bound improvement (we call this *early-stop*). However, a single node merging may affect per-node response-time bounds. For example, (3.10) depends on C_{max}. If C_{max} increases as a result of the merging, all response-time bounds increase. Typically, the increase is relatively small, but if the two longest paths have approximately the same length in the task set, any single merging increases the response-time bound. Thus, we change the termination condition in the base heuristic to allow for a minor increase in the system response-time bound (hoping that additional iterations will cause a net decrease in the bound). The updated heuristic is presented in Listing 3. For simplicity, in the experiments we use BestPair to represent these changes; the other heuristics' behaviors are similar.

Listing 3 Base heuristic pseudocode (disabled early-stop).

```
b_prev step ← current system bound;
b_found ← current system bound;
while b_found ≤ 1.5 · b_prev step do                ▷ Early-stop condition: b_found ≤ b_prev step
    foreach considered pair of nodes (τ1, τ2) do
        S ← nodes to merge with τ1 and τ2
        P12 ← min_{τi∈S}(Pi)
        U12 ← Σ_{τi∈S}(ui)
        if U12 ≤ P12 then
            b_current ← response-time bound after merge
            b_found ← min(b_current, b_found)
        end
    end
    merge the nodes leading to the bound of b_found
end
return best observed bound during the merging       ▷ Early-stop return: b_prev step
```

A comparison of the old (Listing 1, early-stop) and new (Listing 3, no early-stop) base heuristics is shown in Figure 4.7a. However, the updated heuristic may not lead to an improvement in some cases (*e.g.,* for small sets and systems—see Figure 4.7b). To avoid this problem, we returned the minimal observed bound over all merges as a result of the base heuristic instead of the last observed bound (as in Listing 1).

We conducted an experiment with 5,000 generated task sets (set parameters: 4 graphs, 75 total nodes, 24 CPUs) to evaluate if the additional time spent for merges after the bound increase leads to a better bound in the end; it is shown in Figure 4.8. This figure supports the following observation.

Observation 4.1. The disabled early-stop condition improves the efficacy of the heuristics significantly. The effect is higher for task sets with lower utilization.

At the same time, we returned the best computed response-time bound (over all bounds reachable with early-stop) as output. Thus, we do not worsen the returned response-time bound regardless of heuristic. In later experiments, we assume early-stop to be disabled.

4.4.2 Graph Size

In this section, we considered the dependence of heuristics' efficacy on varying graph sizes.

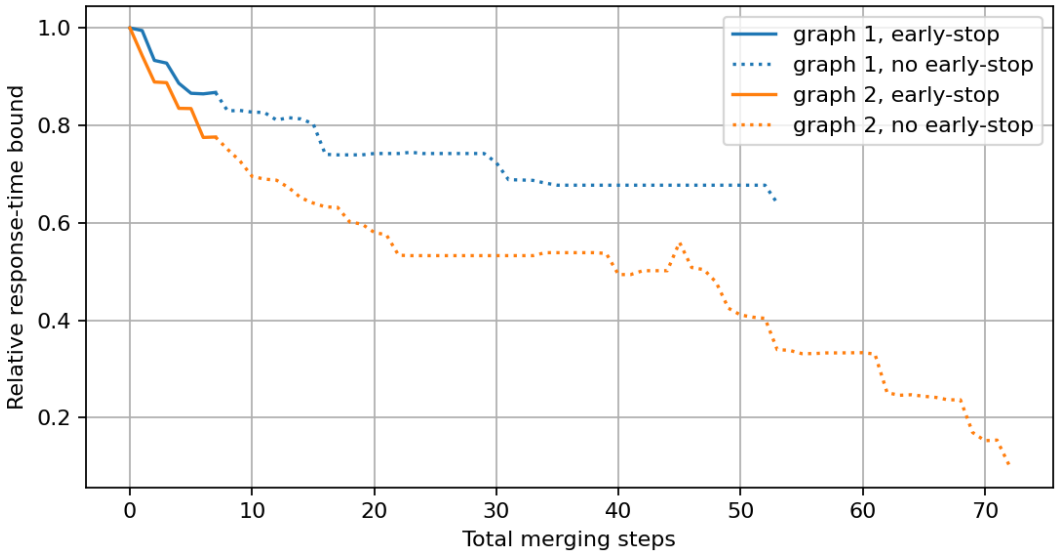
In the first experiment, we fixed the number of graph nodes in the task set and varied the utilization. We considered three types of workloads: small graphs (20 nodes, shown in Figure 4.9a), medium graphs (55 nodes, shown in Figure 4.9b), and large graphs (100 nodes, shown in Figure 4.9c) on a platform with 24 CPUs.

In the second experiment, we fixed the total utilization in the task set and varied the total number of nodes. We considered three types of workloads: low utilizations (35% of the system capacity, shown in Figure 4.10a), medium utilizations (65% of the system capacity, shown in Figure 4.10b), and high utilizations (95% of the system capacity, shown in Figure 4.10c) on a platform with 24 CPUs. Each task set has a single DAG.

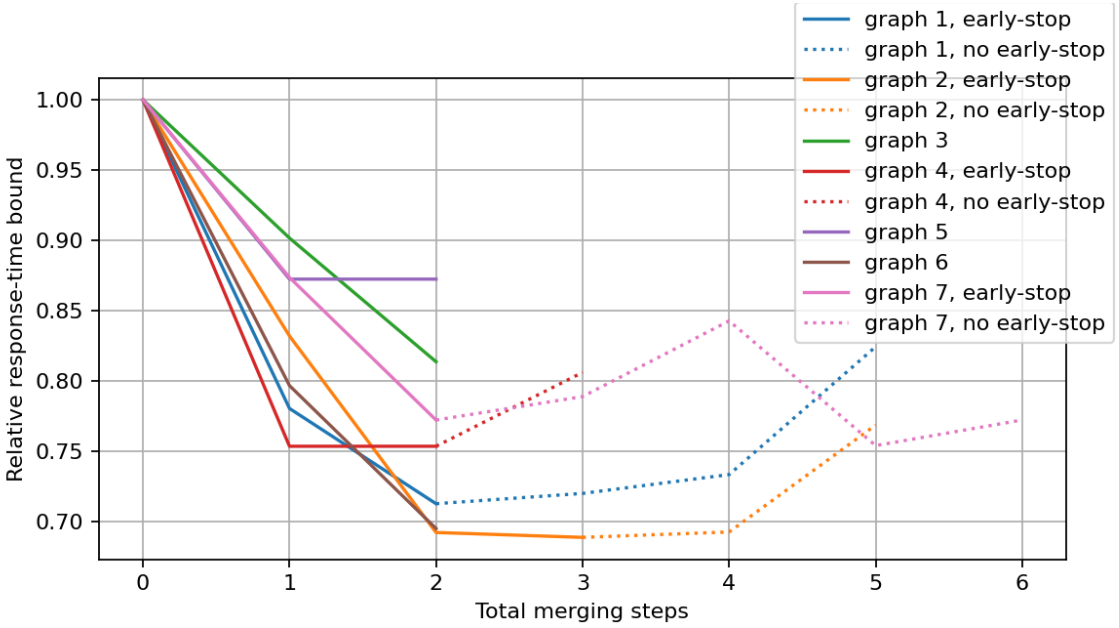
We make several observations based on these two experiments.

Observation 4.2. The heuristics we propose generally improve the response-time bound of the task set. The response-time bound can be reduced by a factor of 5-10 for some task sets.

In our experiments, most of the tasks set achieved a response-time bound reduction. This reduction is effectively free for SRT systems. Recall that node merging is an *offline* process, and can be done once for



(a) Two random sets (set parameters: 5 DAGs, 100 total nodes, 16 CPUs).



(b) Seven small random sets with a single graph (set parameters: 1 DAG, 10 total nodes, 4 CPUs).

Figure 4.7: BestPair heuristic with and without early-stop on several random sets. Solid lines represent improvement before the first response-time bound increase, dotted lines represent the rest of the process. The Y axis represents the lowest reached response-time bound during the merging (relative to the initial bound).

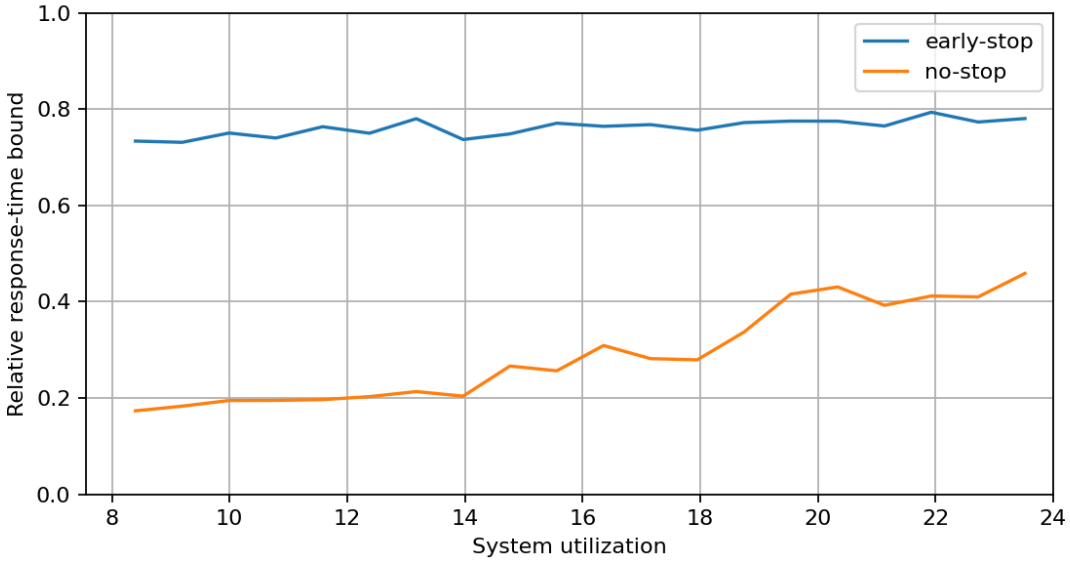


Figure 4.8: BestPair heuristic with and without early-stop on various task sets (set parameters: 4 DAGs, 75 total nodes, 24 CPUs). The Y axis represents the lowest reached response-time bound during the merging (relative to the initial bound).

the considered task set. A runtime implementation of an altered task set only requires that the execution of merged nodes be completed in a predefined order, which should not slow down the actual scheduler.

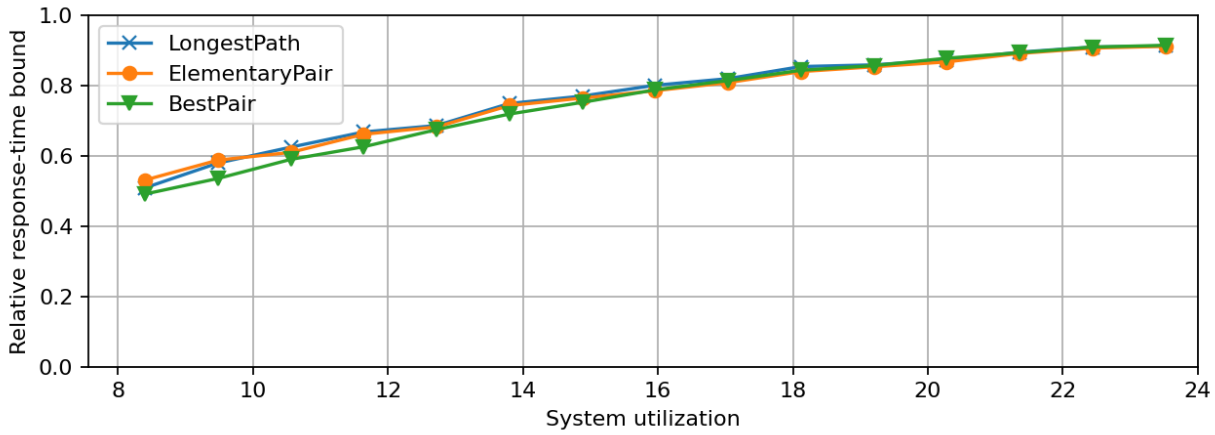
Observation 4.3. BestPair is the best heuristic, with LongestPath being second best.

BestPair considers merging all possible pairs, so it often catches several "'optimal"' merges. LongestPath focuses on reducing the longest path, and may merge the wrong node pairs too early. It is worth noting that LongestPath is significantly faster (compared to BestPair).

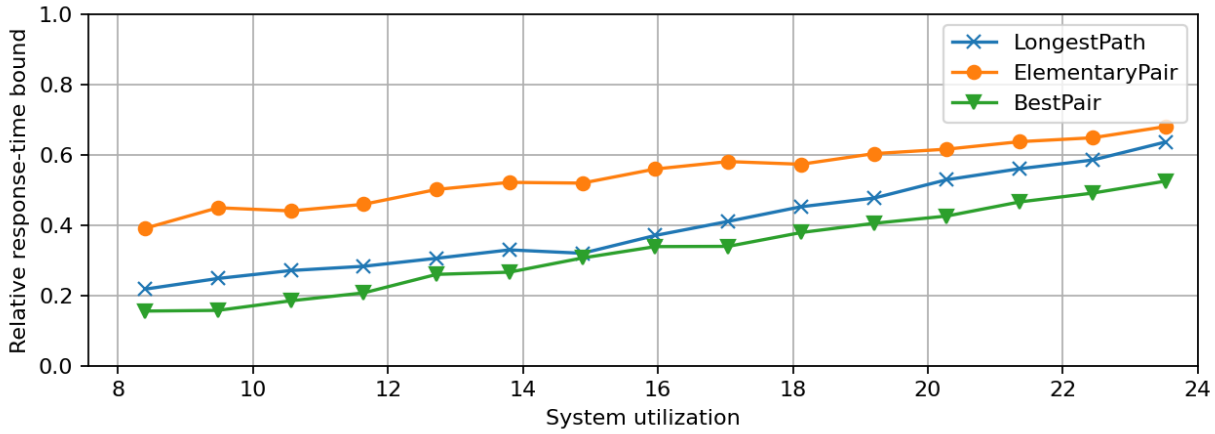
Observation 4.4. The difference between the efficacy of the heuristics is smaller for graphs with few nodes.

Task sets generated with a small number of nodes have high average node utilization (*e.g.,* a 20-node graph with system utilization of 16.0 has an average node utililzation of 0.8). If the expected utilization of a pair of nodes exceeds the parallelization level, almost no nodes can be merged. Thus, heuristics perform in a similar way for task sets with limited merging options.

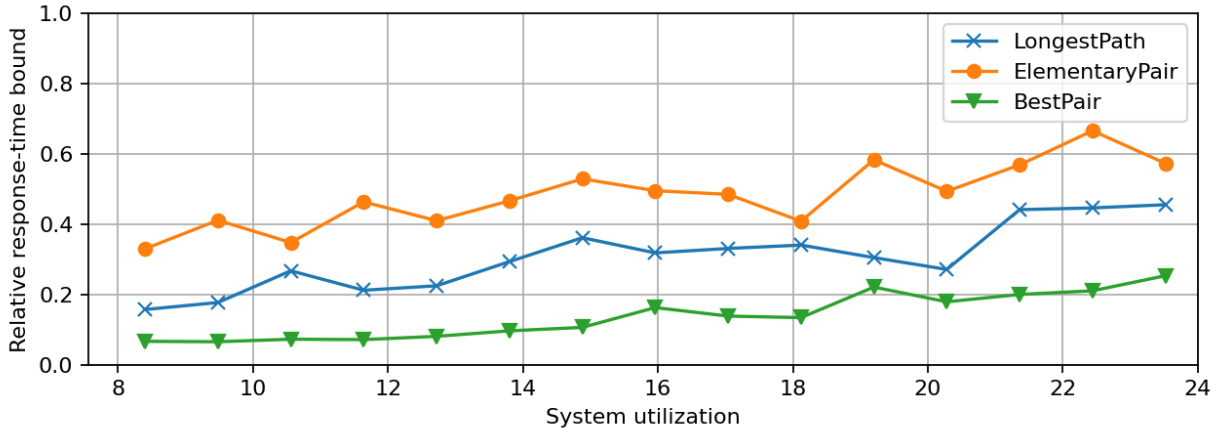
Observation 4.5. The efficacy of all heuristics is higher for task sets with lower utilizations and larger numbers of nodes.



(a) Small graphs (set parameters: 2 DAGs, 20 total nodes).

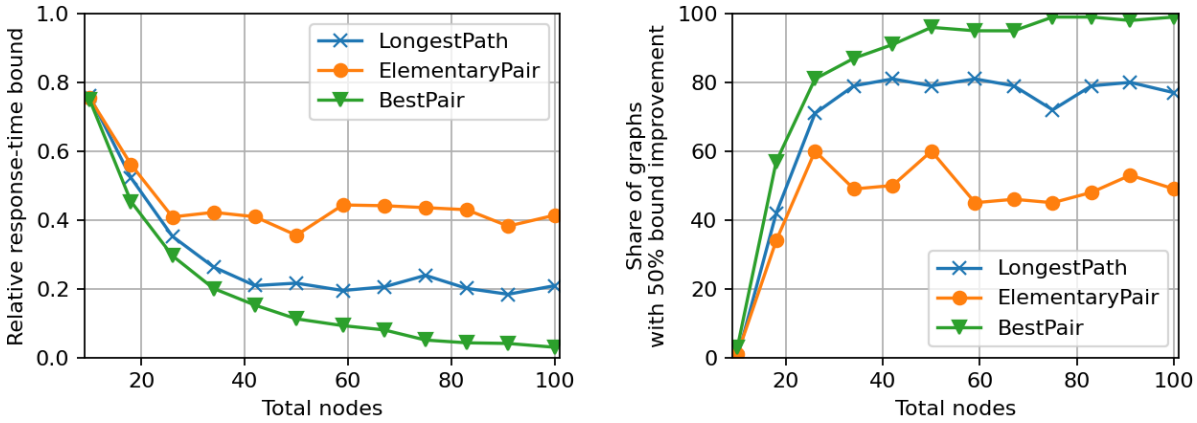


(b) Medium graphs (set parameters: 2 DAGs, 55 total nodes).

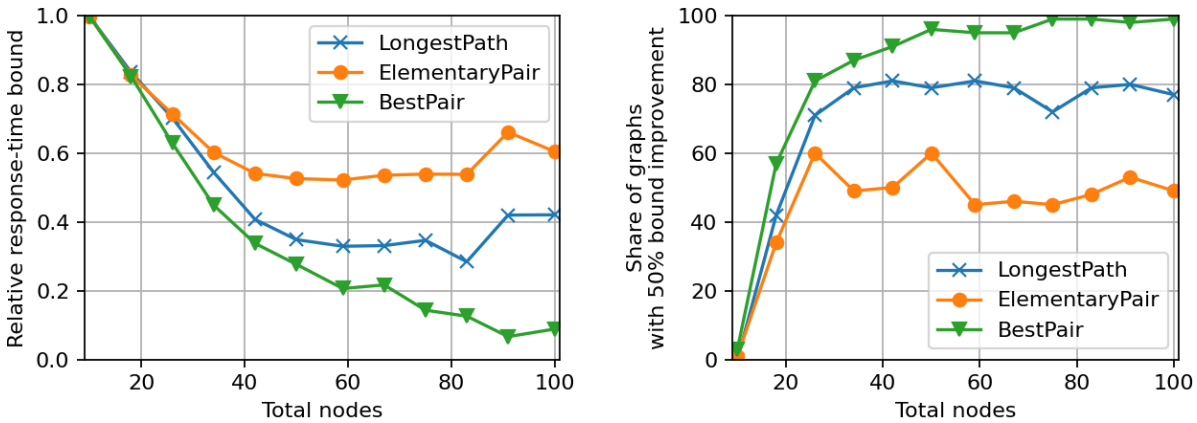


(c) Large graphs (set parameters: 2 DAGs, 100 total nodes).

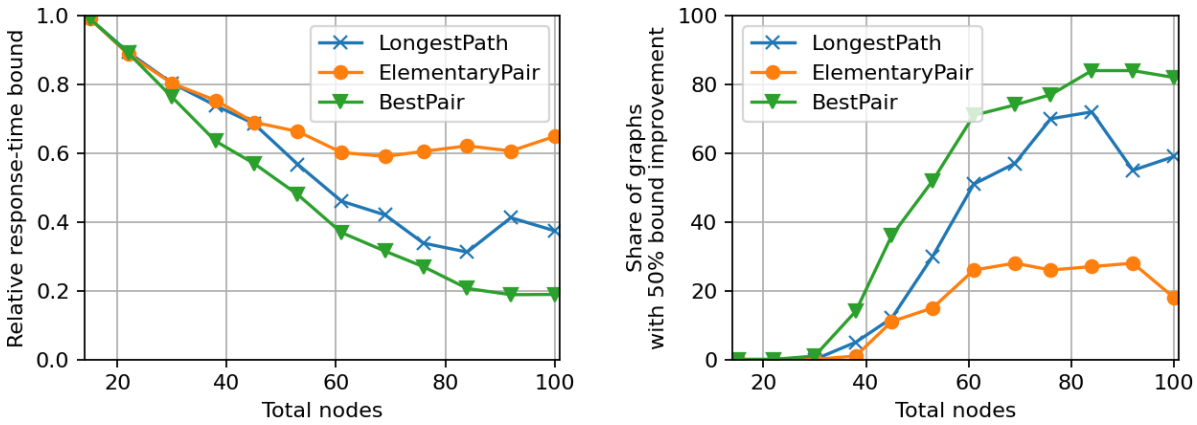
Figure 4.9: Efficacy of three heuristic several random sets on a system with 24 CPUs varying the total utilization. The Y axis represents the lowest reached response-time bound during the merging (relative to the initial bound).



(a) Low utilization (35% of the system capacity).



(b) Medium utilization (65% of the system utilization).



(c) High utilization (95% of the system utilization).

Figure 4.10: Efficacy of the three heuristic on several random sets on a system with 24 CPUs while varying the number of the nodes (single DAG per set). The left Y axis represents the lowest reached response-time bound during the merging (relative to the initial bound). The right Y axis represents the share of graphs that get at least 50% bound reduction with the heuristic.

In both cases (lower utilizations and larger numbers of nodes) the average utilization of nodes decreases. Then more pairs of nodes can be merged, as the schedulability condition for the supernode $U_{super} \leq P_{super}$ is more relaxed on average.

4.4.3 Parallelization Level

In this section, we considered the dependence of heuristic efficacy on varying the parallelization level of the generated tasks. For simplicity, we used the BestPair heuristic as the others' results are similar (but slightly worse).

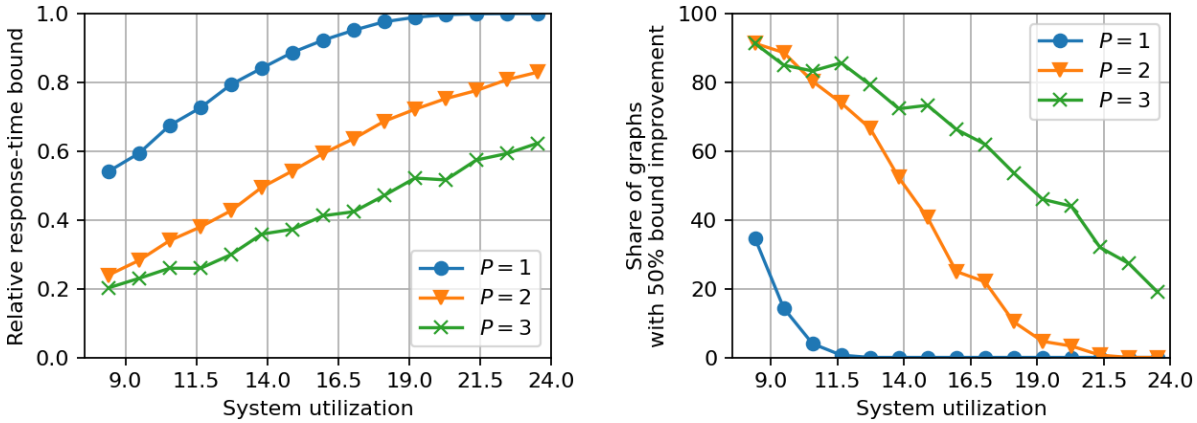
In the first experiment, we considered tasks with $P = 1, 2, 3$ on a platform with 24 CPUs. We consider three types of workloads: small graphs (35 nodes, shown in Figure 4.11a), medium graphs (50 nodes, shown in Figure 4.11b), and large graphs (80 nodes, shown in Figure 4.11c). We conducted this experiment to compare the standard real-time sporadic task model ($P = 1$) and the rp-sporadic task model with low parallelization levels. Amert *et al*. (2021b) showed that the rp-sporadic model with low parallelization levels may be used in place of the sporadic model with small accuracy drops (especially, if no bounds may be guaranteed for the initial system). This experiment leads to the following observation.

Observation 4.6. A higher parallelization level enables more merging possibilities to be considered by the heuristic.

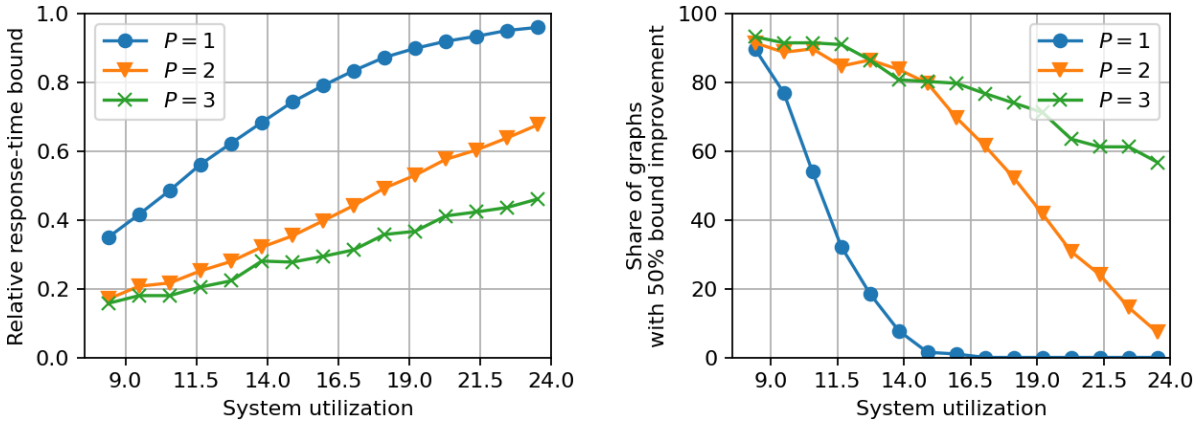
In the second experiment, we considered tasks with $P = 1, 2, 4, 8, 12$ to understand the efficacy of the heuristic for larger parallelization levels, which may not be available for every application because of algorithmic constraints. Results can be found in Figure 4.12. As expected, very high levels lead to better heuristic performance. Note that we do not consider node merging of different graphs between each other, as we expect each graph in the system to has reasonable large utilization (as otherwise it can be used as a single node by the system designer).

4.4.4 The Longest Path

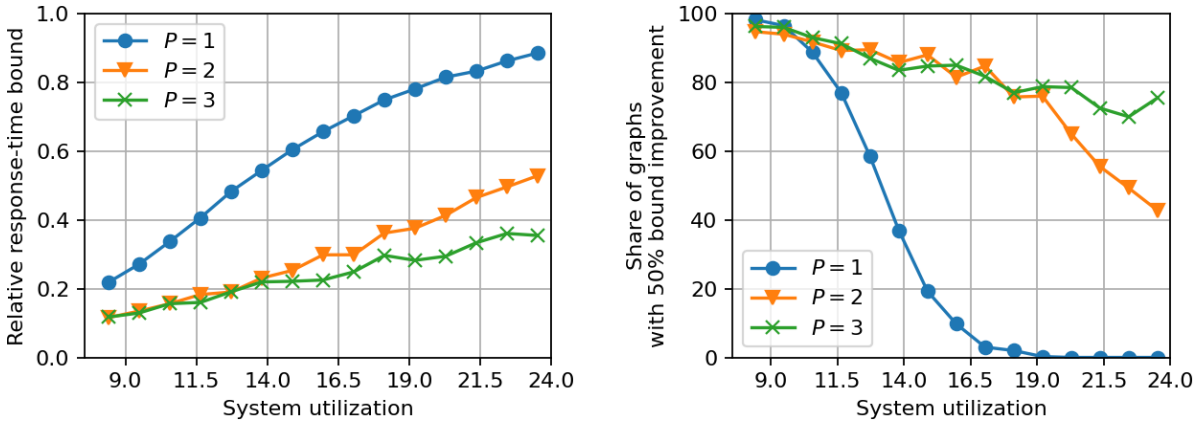
In this section, we considered the dependence of heuristic efficacy on varying the longest path L (in nodes) of the generated tasks. For simplicity, we used the BestPair heuristic as the others' results are similar (but slightly worse). To ensure the length of L we considered the additional edges probability $p = 0.0$ in



(a) Small graphs (set parameters: 2 DAGs, 35 total nodes).



(b) Medium graphs (set parameters: 2 DAGs, 50 total nodes).



(c) Large graphs (set parameters: 2 DAGs, 80 total nodes).

Figure 4.11: Efficacy of the BestPair heuristic on several random sets with different parallelization levels on a system with 24 CPUs varying the total utilization. The left Y axis represents the lowest reached response-time bound during the merging (relative to the initial bound). The right Y axis represents the share of graphs that get at least 50% bound reduction with the heuristic.

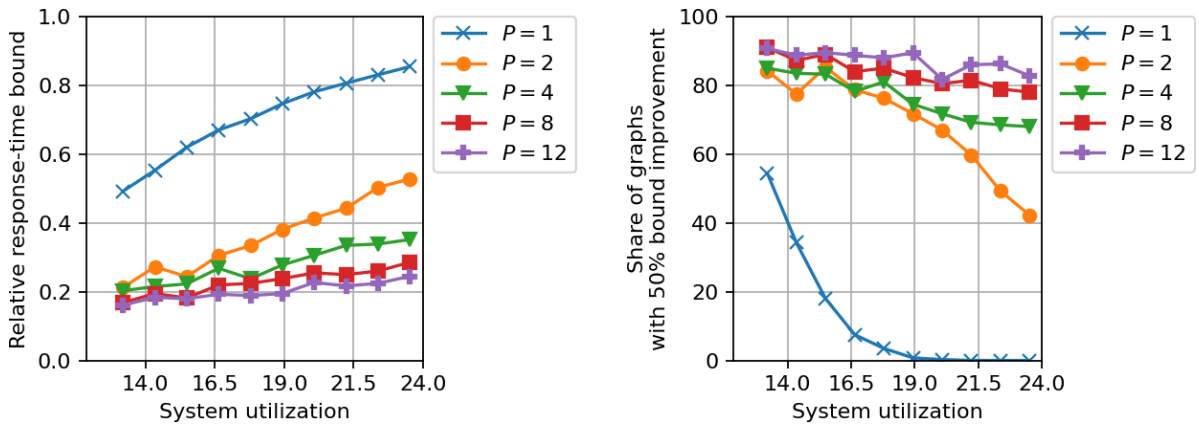


Figure 4.12: Efficacy of the BestPair heuristic on several random sets on a system with 24 CPUs varying the total utilization. The left Y axis represents the lowest reached response-time bound during the merging (relative to the initial bound). The right Y axis represents the share of graphs that get at least 50% bound reduction with the heuristic.

section (*i.e.,* generated graphs are trees with the fixed length of the longest path). Examples of the generated trees can be found in Figure 4.13.

We considered tasks with $L = 4,5,6,7,8,9,10$ and parallelization level $P = 2$ on a platform with 12 CPUs. We considered three types of workloads: small graphs (35 nodes, shown in Figure 4.14a), medium graphs (50 nodes, shown in Figure 4.14b), and large graphs (75 nodes, shown in Figure 4.14c). We conduct this experiment to check if we gain higher improvement from the graphs with a longer paths.

Observation 4.7. The final relative response-time bound is lower for graph with longer paths (*i.e.,* these graphs get larger improvement). This effect is observed for all graphs sizes.

4.5 Chapter Summary

In this chapter, we have presented the offset-based approach for graph response-time bound computation. This approach is the standard SRT graph scheduling approach; graph nodes are shifted with precomputed offsets to satisfy the regular dependencies of the graph and analyzed independently. We described an important drawback of this approach: poor scaling with a graph size. We proposed node merging: an offline algorithm to reduce the response-time bound of SRT graph tasks. We discussed the specific challenges associated with such merging, and proposed several heuristics to reduce the bound. We demonstrated the response-time bound reduction through experiments.

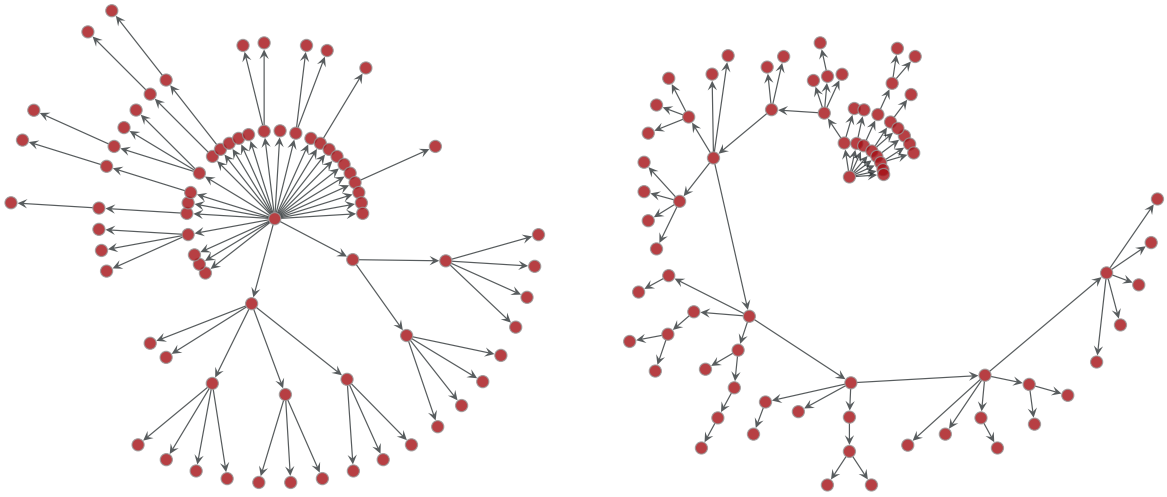
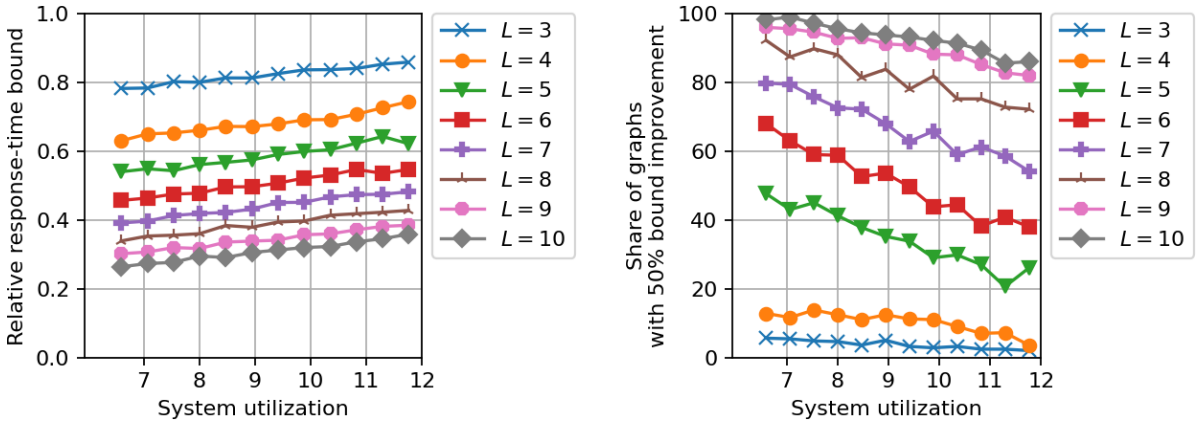
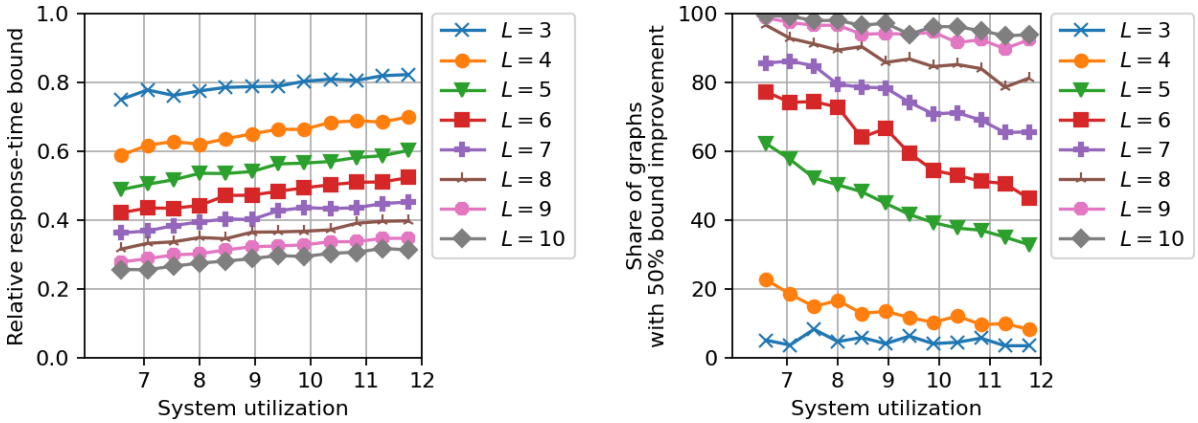


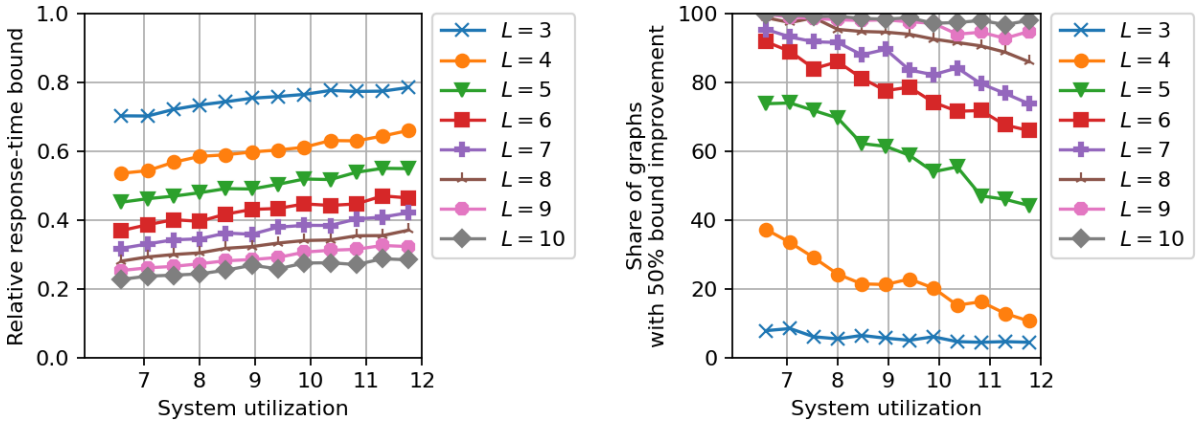
Figure 4.13: Examples of graphs with the fixed length of the longest path. The left graph has $L = 4$, the right graph has $L = 10$.



(a) Small graphs (set parameters: 2 DAGs, 35 total nodes).



(b) Medium graphs (set parameters: 2 DAGs, 50 total nodes).



(c) Large graphs (set parameters: 1 DAGs, 80 total nodes).

Figure 4.14: Efficacy of the BestPair heuristic on several random sets with different longest path lengths on a system with 12 CPUs varying the total utilization. The left Y axis represents the lowest reached response-time bound during the merging (relative to the initial bound). The right Y axis represents the share of graphs that get at least 50% bound reduction with the heuristic.

CHAPTER 5: A SINGLE SYSTEM COMPONENT[1]

In this chapter, we focus our attention on scheduling within a single system component. There are two important problems related to such scheduling: providing temporal isolation for a single system component and response-time analysis of workloads within a single component Ω. Recall that we explained how to deal with graphs in Section 4.1 (see Figure 5.1); the approach does not make any assumptions about response-time bound computations. Thus, we henceforth consider scheduling a set of rp-sporadic tasks with HAC accesses inside reservation Λ of component Ω. We include Figure 5.1 here for a better visualization of our steps.

First, we describe the isolation problem and its solution in Section 5.1. Second, we explain how to deal with HAC accesses in Section 5.2. Third, we provide the response-time analysis for the whole component Ω in Section 5.3. Fourth, in Section 5.4, we propose a way to improve response-time bounds for components with heavy access requests. Fifth, we describe the whole bound computation process in Section 5.5. Then, we conduct experiments in Section 5.6. Finally, we discuss alternative approaches for the analysis in Section 5.7 and conclude in Section 5.8.

5.1 Isolation Problem

We consider a system with a two-level scheduling hierarchy. The top-level scheduler allocates the partitions for the components. As discussed in Section 2.6, we consider a periodic reservation model, where computing resources are made available through periodic reservations. Thus, the top-level scheduler is

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

Amert, T., Voronov, S., and Anderson, J. H. (2019). OpenVX and real-time certification: The troublesome history. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 312–325.

Voronov, S., Tang, S., Amert, T., and Anderson, J. H. (2021b). AI meets real-time: Addressing real-world complexities in graph response-time analysis. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 82–96,

Amert, T., Tong, Z., Voronov, S., Bakita, J., Smith, F. D., and Anderson, J. H. (2021a). Timewall: Enabling time partitioning for real-time multicore+accelerator platforms. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 455–468.

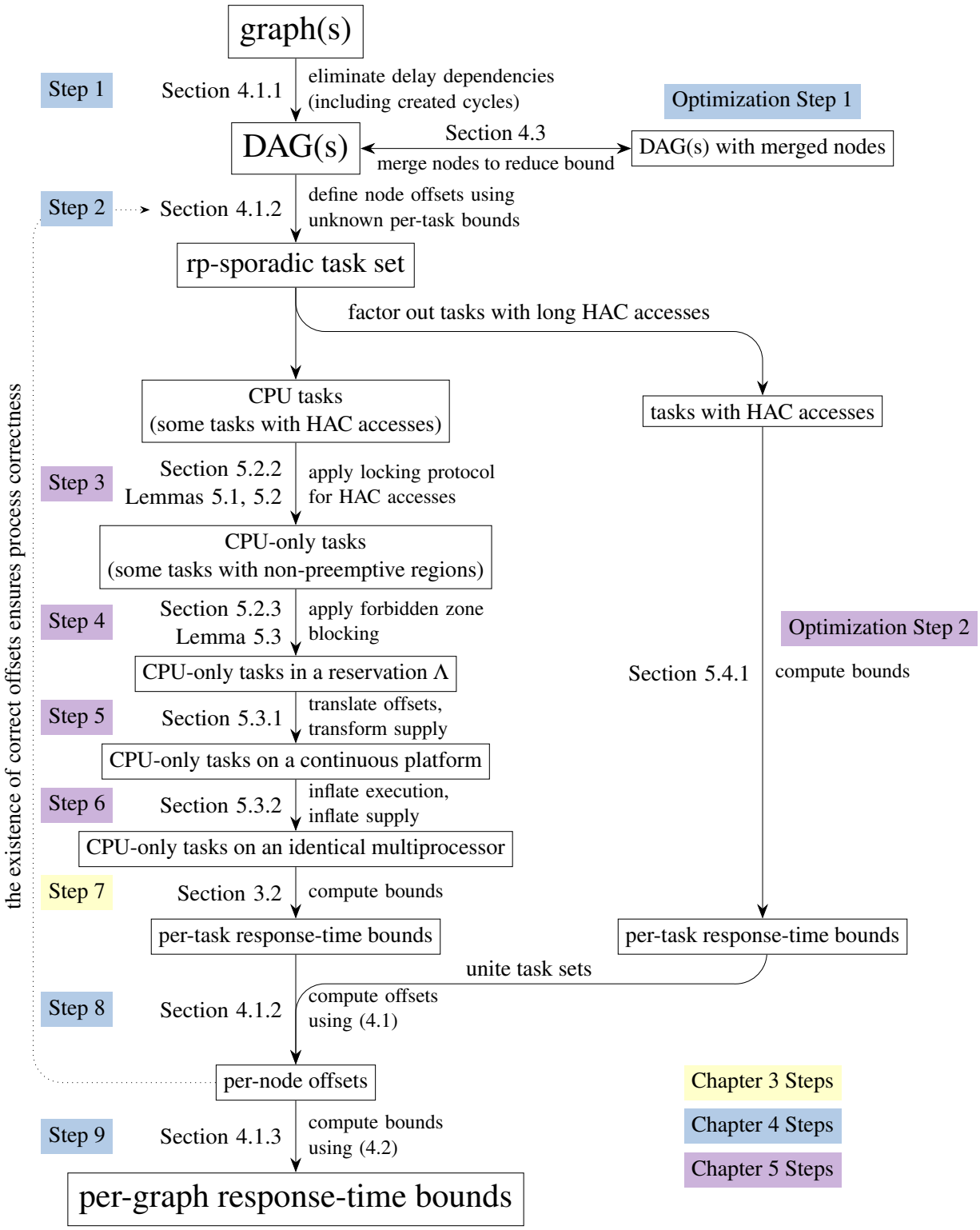


Figure 5.1: The response-time bound computation for the component Ω.

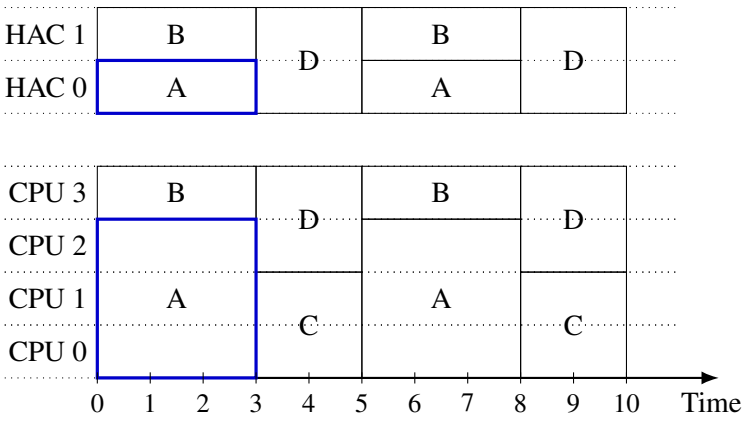


Figure 5.2: System with four components A, B, C, and D, four CPUs and two HACs (rectangles represent component reservations).

effectively a table-driven scheduler, where the table is determined offline. An example of multiple reservations in a system can be found in Figure 5.2.

The bottom-level scheduler is the in-partition scheduler. In this dissertation, we consider G-EDF ("global" within component Ω), but any G-EDF-like (Leontyev et al., 2011; Erickson and Anderson, 2011; Ward et al., 2013; Ahmed and Anderson, 2021) scheduler can be applied with small changes to the analysis.

The important property in such systems is component isolation. Such isolation assumes that every component keeps its workload strictly within its allocated reservation of resources (regardless of the reservation model). If all system reservations are assigned without intersections, this property ensures the temporal isolation of components. Thus, we need to ensure that a single component of interest respects reservation bounds.

Our task model assumes preemptive (*e.g.,* CPU) and non-preemptive (*e.g.,* GPU) workloads. Any preemptive workload can be preempted at the reservation end with relatively low overhead (by the definition of preemptivity). Unfortunately, non-preemptive workloads cannot be processed the same way. An example of an isolation break is shown in Figure 2.12. To avoid such breaks, we use the idea of the forbidden zone (Holman and Anderson, 2006): a non-preemptive workload is prohibited from beginning execution near the end of a reservation. More formally, a forbidden zone is the time interval in which the access may not be initiated (as it may not complete before the end of the reservation). Because of the isolation property, *the accelerator usage of other system components does not affect the forbidden zone of the component of interest*. Note that the use of forbidden zones in Ω requires that no accelerator access in Ω takes more than Θ time units (the length of Λ).

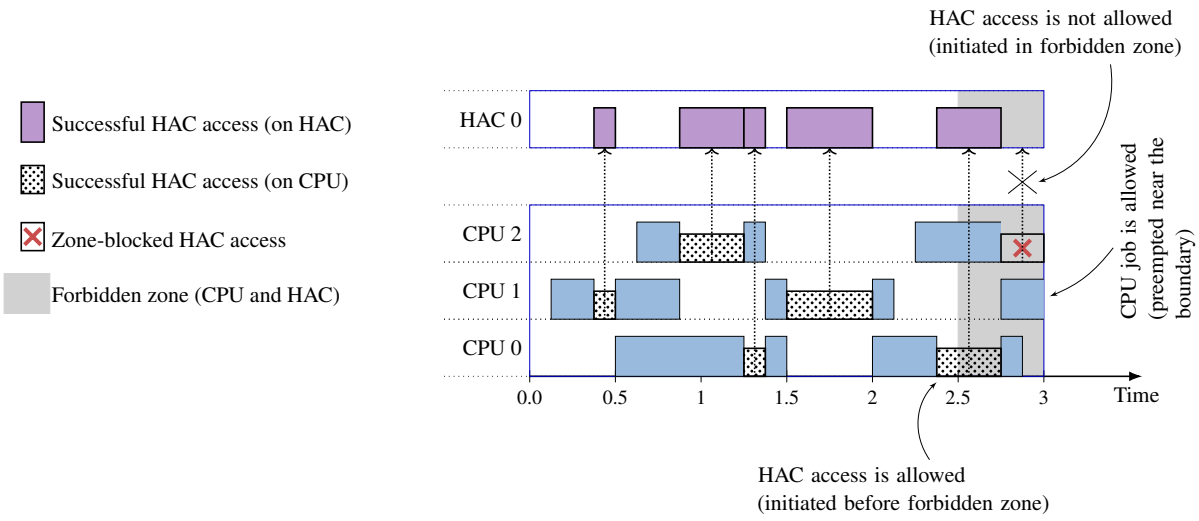


Figure 5.3: Example of the forbidden zone of reservation Λ_A of component A (see Figure 5.2).

Example 5.1 (see Figure 5.3). Consider a single reservation of the component A of the system shown in Figure 5.2. We detail a schedule inside this component in Figure 5.3. The forbidden zone is shown in grey (starts at 2.5). The HAC access request issued at 2.35 on CPU 0 is allowed (initiated before the forbidden zone starts); the HAC access request issued at 2.75 on CPU 2 is blocked (initiated within the forbidden zone). The CPU job executed in $[2.75, 3)$ is preempted at the reservation boundary. ◊

Forbidden zones can be defined on a *per-reservation* basis (equal to the longest HAC access in a reservation) or on a *per-access* basis (equal to the HAC access' worst-case length). For simplicity, we consider forbidden zones defined on a per-reservation basis. In the next section, we explain how to deal with HAC accesses.

5.2 Dealing with HAC Accesses

There are two major approaches to deal with HAC accesses: locking protocols and independent scheduling. The first approach assumes that HAC accesses are arbitrated through a locking protocol; the total time required for these accesses inflates the WCETs of the accessing tasks. Generally speaking, this approach is CPU-centric (the scheduling process is viewed from the CPUs' point of view). The second approach considers independent scheduling of CPU and HAC nodes. As offset-based graph scheduling (see Section 4.1.2) requires only per-task response-time bounds to compute offsets, CPU-task and HAC-task bounds may use

different analyses. We describe the locking protocol approach (with necessary analysis) in this section; the second approach is explained in Section 5.4.

For simplicity, we first assume a single HAC in reservation Λ of component Ω (and explain how to deal with several identical HACs later).

5.2.1 Locking Protocol

We arbitrate accesses to each HAC with a multiprocessor locking protocol. In this chapter, we consider the global ("global" within component Ω) OMLP (Brandenburg and Anderson, 2010), though alternative locking protocols could be used.[2] We also contrast against the non-preemptive FIFO spinlock to highlight difference between spin-based and suspension-based locking protocols. Spin-based protocols usually have lower overheads (compared to the cost of suspending and resuming tasks), but waiting jobs waste CPU capacity.

We consider access *execution* time (resp., access *blocking* time) as non-preemptive (resp., preemptive) CPU execution under the OMLP.[3] Thus, we assume that all tasks are CPU-only tasks, with their WCETs inflated to include HAC blocking and execution times, and that tasks can contain non-preemptive regions due to said locking protocol.

When used on m_Ω processors, the global OMLP ensures $O(m_\Omega)$ pi-blocking by utilizing a dual-queue structure, with an m_Ω-element FIFO queue fed into by a priority queue, as depicted in Figure 5.4. When the access request at the head of the FIFO queue (i.e., the lock holder) completes, it is dequeued, and the next access request (if any) in the FIFO queue becomes satisfied; if the priority queue is not empty, the highest-priority access request is moved from the priority queue to the tail of the FIFO queue. Note that jobs in both queues (except the lock holder) are suspended under the global OMLP.[4]

Our motivation to use the global OMLP is based on the choice of in-partition scheduler (G-EDF): the OMLP has optimal priority-inversion blocking (pi-blocking) under suspension-oblivious analysis, which is the suspension-accounting method usually used under G-EDF. Under suspension-oblivious analysis, a job

[2]We consider only locking protocols with non-preemptive critical sections to support non-preemptive HAC accesses.

[3]Both execution and blocking time are considered non-preemptive CPU execution under the FIFO spinlock.

[4]The FIFO spinlock has one m_Ω-element FIFO queue. Jobs in this queue occupy their CPUs while waiting for the lock holder. Thus, the FIFO spinlock may show worse results at runtime.

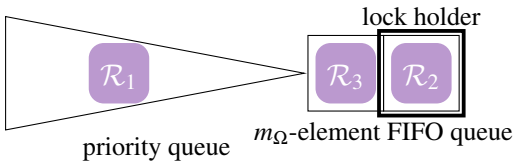


Figure 5.4: The global OMLP state representation ($m_\Omega = 2$).

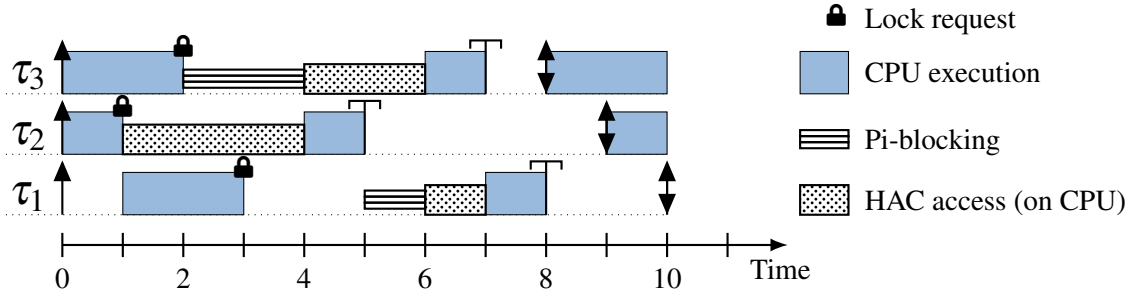


Figure 5.5: Three tasks accessing one shared HAC via the global OMLP with $m_\Omega = 2$. This figure is inspired by Figure 1 in (Brandenburg and Anderson, 2010).

in Ω is only pi-blocked if it is one of the m_Ω highest-priority active jobs but is not scheduled. We explain pi-blocking with the following example.

Example 5.2 (see Figures 5.5 and 5.6). Consider a schedule of first jobs J_1, J_2, J_3 of tasks τ_1, τ_2, τ_3; each job has a HAC access. We assume $m_\Omega = 2$ and a single HAC; also, the reservation does not end within the first 10 time units (*i.e.,* the forbidden zone is not present in $[0, 10]$).

Job J_1 is blocked during the entire interval $[3, 6)$, but is only pi-blocked in the interval $[5, 6)$, as only then is J_1 one of the $m_\Omega = 2$ highest-priority active jobs. In checking schedulability, both pi-blocking times and HAC execution times are analytically viewed as CPU execution time, as depicted in Figure 5.6 (the execution of job J_1 is shifted in the analysis because only two CPUs are available). Note that this execution-time inflation may not include all task suspension time, but only that occurring while a task is actually pi-blocked.

Assume that job J_i tries to requests a HAC access (call it $\mathcal{R}_i$). Requests $\mathcal{R}_2$ and $\mathcal{R}_2$ are enqueued in the FIFO queue upon issuance. Request $\mathcal{R}_1$ is enqueued in the priority queue, as the FIFO queue is full. The state of both queues of the global OMLP at time 3.5 is shown in Figure 5.4. $\Diamond$

The next question we consider is the effect of locking protocol on the workload in Ω.

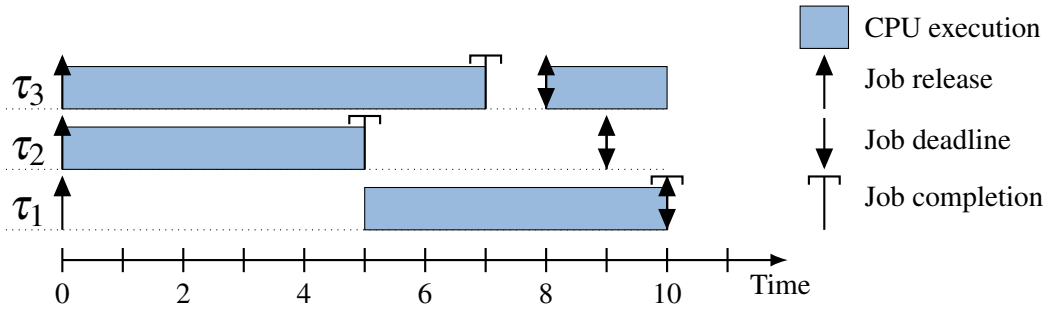


Figure 5.6: Illustration of suspension-oblivious analysis under the global OMLP.

5.2.2 Abstracting HAC Accesses

In this subsection, we consider Step 3 of Figure 5.1. We determine upper bounds on the necessary protocol-related WCET inflation. Note that the response-time bound analysis we use in Step 7 of Figure 5.1 (see Section 3.2) supports non-preemptive job regions, so we do not need to account for release blocking in this section.

Definition 5.1. Let B_{max} be the longest duration of a HAC access in Ω, and X be the maximum pi-blocking time of a HAC access.

The locking protocol approach abstracts each HAC access of a task τ_i by inflating the access length by X time units (C_i and u_i are also inflated). The following lemmas provide an upper bound on X for the global OMLP and non-preemptive FIFO spinlock.

Lemma 5.1 (Theorem 1 in (Brandenburg and Anderson, 2013))**.** *$X_{OMLP} \leq (2m_\Omega - 1)B_{max}$ under global OMLP on an identical multiprocessor with m_Ω CPUs without time partitioning.*

Lemma 5.2. *$X_{FIFO} \leq (m_\Omega - 1)B_{max}$ under FIFO spinlock on an identical multiprocessor with m_Ω CPUs without time partitioning.*

Proof. While holding a FIFO spinlock, a job remains non-preemptive on its CPU for the total duration of its spinning and HAC access. This busy waiting on the CPU ensures that at most m_Ω HAC requests may be simultaneously in progress. Thus, each individual request waits behind at most $m_\Omega - 1$ other requests in the FIFO queue. □

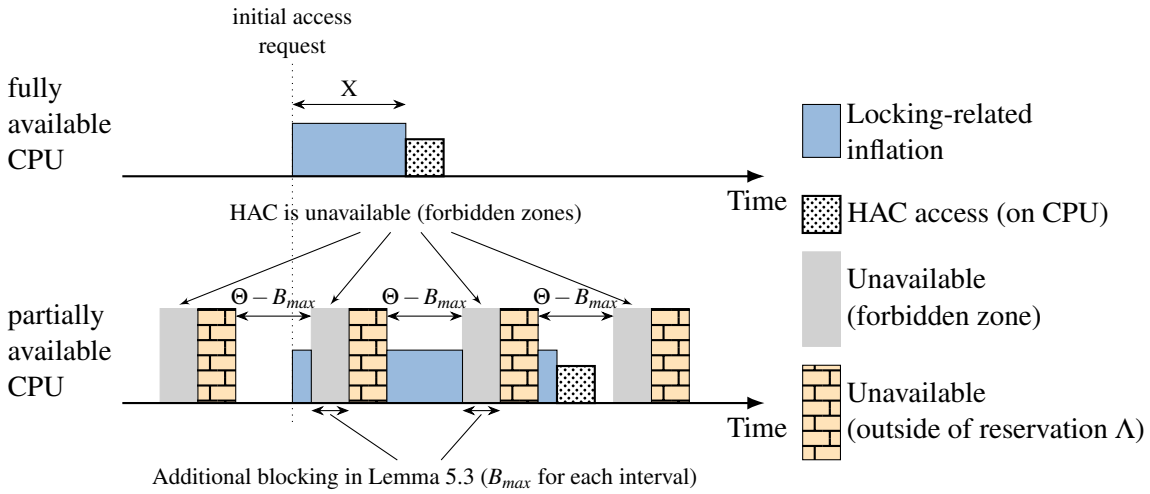


Figure 5.7: Lemma 5.3 illustration.

5.2.3 Abstracting Forbidden Zones

In this subsection, we consider Step 4 of Figure 5.1. We determine upper bounds on the necessary forbidden zone-related WCET inflation. We augment the blocking analysis for locking protocols to consider forbidden zones with Lemma 5.3. Note that zone-related inflation is caused by the non-preemptivity of accesses, so we augment X for both the OMLP and the FIFO spinlock.

Lemma 5.3. *The total pi-blocking introduced by the management of non-preemptive HAC accesses is at most* $X + \left\lceil \frac{X + B_{max}}{\Theta - B_{max}} \right\rceil \cdot B_{max}$ *time units for each lock request by a task in Component* Ω.

Proof. In each reservation of length Θ, at least $\Theta - B_{max}$ time units are available for a job to initiate an access to a HAC (because B_{max} is the worst-case duration of a forbidden zone—see Figure 5.7). In executing this work, and while the access is unfinished, additional blocking of up to B_{max} time units may be incurred for each reservation boundary crossed. Thus, we upper bound the number of such boundaries that may be crossed between the initiation of the request and the completion of the access. The worst case occurs when the request is initiated right before a reservation boundary, in which case $\left\lceil \frac{X + B_{max}}{\Theta - B_{max}} \right\rceil$ boundaries are crossed. □

Skipping ahead. The bound of Lemma 5.3 can be improved with per-access forbidden zones and the *skipping ahead* approach. If the next HAC access of the job at the head of the OMLP's FIFO queue is requested within its forbidden zone, then we can allow other access requests to "skip ahead" of that access until the beginning of the next reservation invocation. This corresponds to the *Skip Protocol* proposed previously (Holman

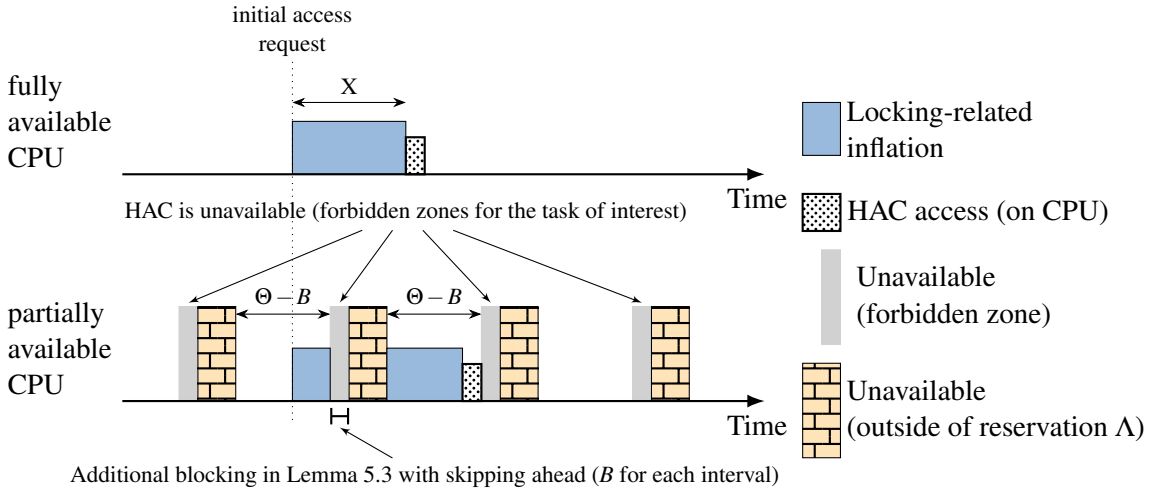


Figure 5.8: Skipping ahead illustration. A task with the same locking-related inflation as in Figure 5.7 but with a smaller access length.

and Anderson, 2006). With skipping, the total blocking bound of an access of length B can be reduced to $X + \left\lceil \frac{X+B}{\Theta - B} \right\rceil B$, because the accesses ahead in the FIFO queue must utilize at least $\Theta - B$ time units in each reservation instance of size Θ or let the access of interest skip to the head of the queue. The change is illustrated in Figure 5.8.

To convert (analytically) pi-blocking to a CPU-only workload, we must add to the WCET of each task in Ω the maximum blocking duration X and the duration of the HAC access itself for each such access. Note that this added CPU execution may be preemptive (under OMLP) or non-preemptive (under the FIFO spinlock). This WCET inflation may cause task utilizations to exceed 1.0, which is an additional motivation for the rp-sporadic task model instead of the standard sporadic model.

5.2.4 Extending to Multiple HACs

We consider two variants of multiple HACs per component: several independent HACs (which may even have different types) or several identical HACs.

In the first case, each HAC is governed by its own lock within the component. From the point of view of the analysis, B_{max} and X are defined on a per-HAC basis, and each access is inflated using these B_{max} and X.

If a set of k identical HACs is interchangeable (any HAC in the set can service a request), then k HACs can instead be managed by a k-exclusion locking protocol (which allows up to k "lock holders"). This roughly

divides X by a factor of k in the analysis:

$$X^k_{FIFO} \leq \left(\left\lceil \frac{m_\Omega}{k} \right\rceil - 1 \right) B_{max} \qquad \text{and}$$

$$X^k_{OMLP} \leq \left(2 \left\lceil \frac{m_\Omega}{k} \right\rceil - 1 \right) B_{max} \text{ (by Theorem 3 in (Brandenburg and Anderson, 2013)).}$$

5.3 Response-Time Bounds in a Reservation

In this section, we abstract the reservation model and convert a task set scheduled in the reservation Λ of the component Ω to the task set scheduled on an identical multiprocessor (recall that forbidden zones were abstracted away with HAC accesses in Section 5.2). This pertains to Steps 5 and 6 of Figure 5.1. We assume a set of CPU-only tasks (*i.e.,* tasks with WCETs inflated using Lemma 5.3). We seek to leverage existing response-time analysis for rp-sporadic tasks on a multiprocessor platform described in Chapter 3 for which (unlike our considered platform) all CPUs are fully available; our results are applicable for any such analysis that supports early-releasing (like most G-EDF analyses do).

Steps 5 and 6 of Figure 5.1 define the sequence of processing-supply transformations, starting with the supply given by Λ and ending with a supply corresponding to a fully available platform. Step 5 transforms Λ to a reservation with the same schedule and a continuous supply; this is discussed in Section 5.3.1. Step 6 inflates task execution times to apply the analysis of Section 3.2; this is discussed in Section 5.3.2. We perform these transformations such that no job's response time decreases, and we track task utilization changes as we transform. It is important to track utilizations so that the schedulability of the task set is preserved during transformations.

5.3.1 Abstracting Partial Supply

Recall that PCR Λ is parameterized by Θ (length), Π (period), and m_Ω (number of CPUs). In this subsection, we describe Step 5 of Figure 5.1. We transform Λ to a continuous processing supply without changing the total processing capacity supplied over long time intervals (*e.g.,* the hyperperiod of all system reservations).

Definition 5.2. Let Φ_{res} be the start of the first invocation of the reservation Λ of Ω. Because Λ is periodic, it is available during intervals $[\Phi_{res} + i\Pi, \Phi_{res} + i\Pi + \Theta)$ for $i \in \{0, 1, 2, ...\}$ and $\Phi_{res} + \Theta \leq \Pi$.

Definition 5.2 allows us to describe the transformed CPU platform.

Definition 5.3. Define a new platform with m_Ω CPUs, each with speed Θ/Π, that begins supplying processing time at time Φ_{res}. We call this platform the *reduced-speed platform*.

The total supply provided to Ω by the initial and reduced-speed platforms is the same over any time interval of length $i\Pi$ for $i \in \{0,1,2,...\}$ that starts after Φ_{res}. The processing supply provided by both platforms is depicted in Figure 5.9a.

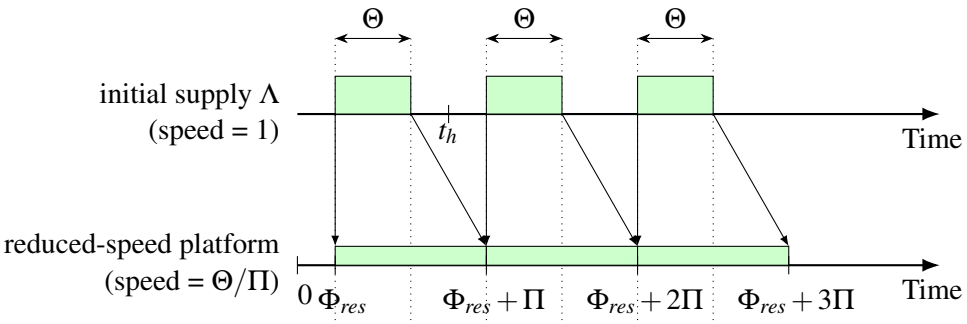
Although both platforms deliver equal processing supply in the long run, the change in how processing supply is provided changes the schedule significantly. In particular, job completion times in the initial and reduced-speed schedules may differ greatly. For example, consider a task in Ω that releases a job in the interval between two invocation of Λ (*e.g.,* t_h in Figure 5.9a, initial supply). On the initial platform, such a job must wait until the invocation of Λ to be considered for execution, whereas on the reduced-speed platform, it would execute immediately if there are free CPUs. To avoid this issue, we transform job releases and deadlines of the tasks in Ω.

Definition 5.4. Define the piecewise-linear function $F(\cdot)$, plotted in Figure 5.9b (solid line), as follows:

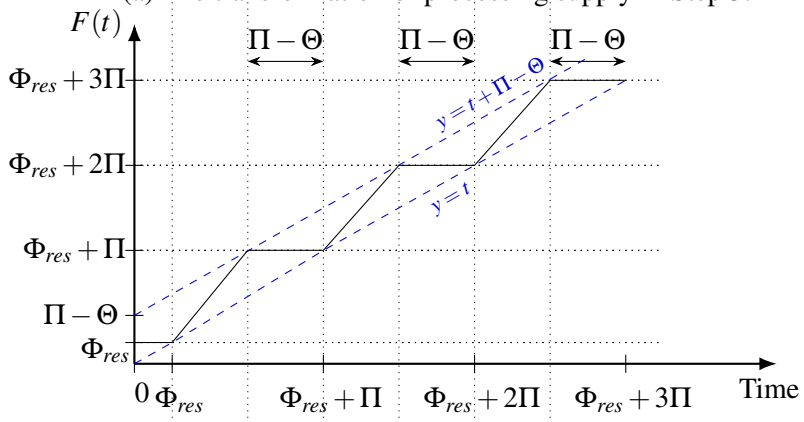
$$F(t) = \begin{cases} \Phi_{res} & \text{if } t \in [0, \Phi_{res}), \\ \Phi_{res} + i\Pi + z\Pi/\Theta, & \text{if } t \in [\Phi_{res} + i\Pi, \Phi_{res} + i\Pi + \Theta), \\ \Phi_{res} + (i+1)\Pi, & \text{if } t \in [\Phi_{res} + i\Pi + \Theta, \Phi_{res} + (i+1)\Pi), \end{cases}$$

where $z = t - \Phi_{res} - i\Pi$.

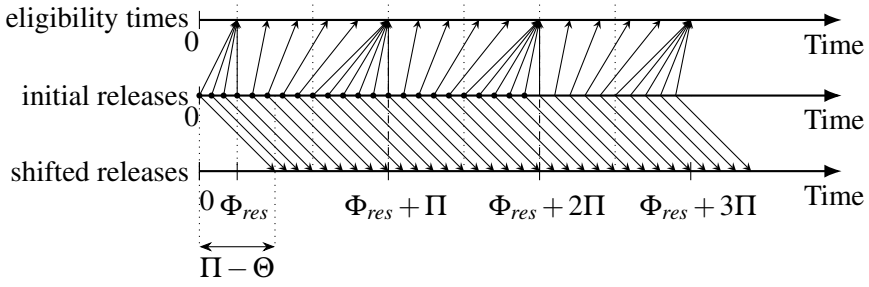
To circumvent the issue of inconsistent allocations noted above, we first shift all job releases and deadlines on the reduced-speed platform into the future by $\Pi - \Theta$ time units compared to the initial platform (see Figure 5.9c). Then, we allow the early releasing of jobs on the reduced-speed platform. Specifically, letting $r_{i,j}$ be the release time of a job on the initial platform, we define its release time on the reduced-speed platform to be $(r_{i,j} + \Pi - \Theta)$ and its eligibility time to be $F(r_{i,j})$ (we explain below why this is "early" after Lemma 5.4), as illustrated in Figure 5.9c. The motivation of the exact definition of $F(\cdot)$ is to force ready times of jobs on the reduced-speed platform to follow the transformation pattern of the initial to the reduced-speed platforms.



(a) The transformation of processing supply in Step 5.



(b) Plot of $F(\cdot)$.



(c) The transformation of releases and eligibility times.

Figure 5.9: Transformation clarification figures.

To preserve the scheduling order, we break deadline ties the same way as before the transformation. Thus, the priority order of jobs does not change after this transformation. To explore how schedules on the initial and reduced-speed platforms are now connected, we first need bounds on $F(\cdot)$.

Lemma 5.4. $t \leq F(t) \leq t + (\Pi - \Theta)$.

Proof. Figure 5.9b illustrates the lemma statement. First, consider $t \in [0, \Phi_{res})$. By Definition 5.4, $t < F(t)$. Furthermore, because the first reservation invocation is $[\Phi_{res}, \Phi_{res} + \Theta)$, and the reservation is periodic, $\Phi_{res} + \Theta \leq \Pi$. Thus, $F(t) = \Phi_{res} \leq \Pi - \Theta \leq t + \Pi - \Theta$.

Second, consider $t \in [\Phi_{res} + i\Pi, \Phi_{res} + i\Pi + \Theta)$ for some integer i, and $z = t - \Phi_{res} - i\Pi$. In this case, by Definition 5.4,

$$
\begin{aligned}
t &= \Phi_{res} + i\Pi + z \\
&\leq \Phi_{res} + i\Pi + z \cdot \Pi/\Theta && \{= F(t)\} \\
&= \{\text{because } \Theta \leq \Pi \text{ and } z \geq 0\} \\
&\quad \Phi_{res} + i\Pi + z + z \cdot (\Pi/\Theta - 1) \\
&= t + z \cdot (\Pi/\Theta - 1) \\
&\leq \{\text{because } z \leq \Theta\} \\
&\quad t + \Theta \cdot (\Pi/\Theta - 1) \\
&= t + (\Pi - \Theta).
\end{aligned}
$$

Finally, if $t \in [\Phi_{res} + i\Pi + \Theta, \Phi_{res} + (i+1)\Pi)$, then by Definition 5.4, $F(t) = \Phi_{res} + (i+1)\Pi$, so $t < F(t)$. Additionally, $F(t) = \Phi_{res} + i\Pi + \Pi + (\Theta - \Theta) \leq t + (\Pi - \Theta)$, as $t \geq \Phi_{res} + i\Pi + \Theta$. □

By Lemma 5.4, on the reduced-speed platform, a job with a release at time $r_{i,j}$ can be scheduled at time $F(r_{i,j})$ due to its defined eligibility time, while its actual release happens at time $r_{i,j} + \Pi - \Theta \geq F(r_{i,j})$.

Definition 5.5. Let S_{in} be the schedule of Ω on the initial platform defined by Λ. Let S_{tr} denote the corresponding schedule on the reduced-speed platform, with job releases and deadlines adjusted as above.

The next lemma gives the schedule correspondence we seek.

Lemma 5.5. *A job is scheduled at time t in S_{in} if and only if it is scheduled at time $F(t)$ in S_{tr}.*

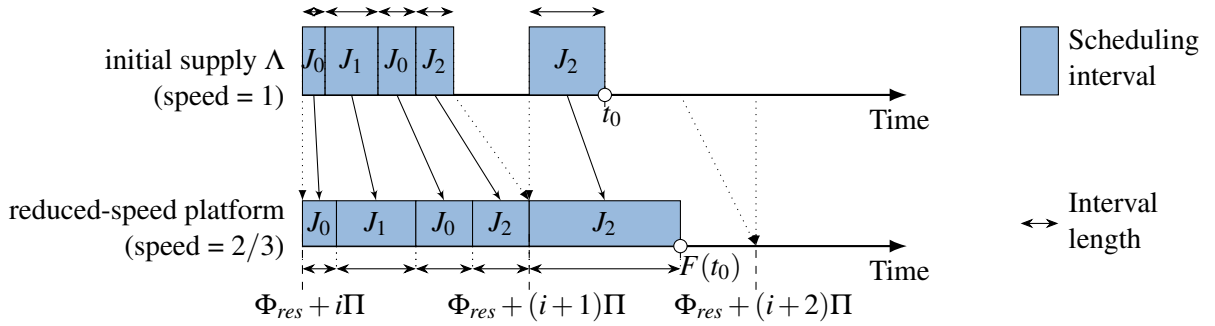


Figure 5.10: Lemma 5.5 intuition (assuming $\Theta/\Pi = 2/3$).

Proof. Figure 5.10 illustrates the proof. Assume, for the purpose of contradiction, that the lemma does not hold, *i.e.,* that there exists a time t such that the sets of scheduled jobs in S_{in} at t and S_{tr} at $F(t)$ differ. Let t_0 be the first such time instant.

By the definition of Φ_{res}, no jobs are scheduled in either S_{in} or S_{tr} within $[0, \Phi_{res})$, so $t_0 > 0$. By the definition of t_0, any scheduling interval of a job within $[0, t_0)$ in S_{in} is transformed into a scheduling interval of the same job in S_{tr} by $F(\cdot)$. By Definition 5.4, any such interval of length h is transformed into a scheduling interval of length $h \cdot \Pi/\Theta$, scheduled on a CPU with speed Θ/Π, because a job can be scheduled in S_{in} only during active reservation slices (see Figure 5.10). This results in the same total amount of completed work, h, for both the initial and transformed intervals. Thus, the amount of completed work for each job in S_{in} within $[0, t_0)$ is identical to the amount of completed work for the same job in S_{tr} within $[0, F(t_0))$. As the eligibility times of jobs are also transformed from the actual releases with $F(\cdot)$, the sets of uncompleted jobs at t_0 in S_{in} and at $F(t_0)$ in S_{tr} are identical. The transformation process does not affect the relative order of deadlines, so the set of scheduled jobs is the same in S_{in} at t_0 and in S_{tr} at $F(t_0)$, which contradicts the definition of t_0. □

Because $F(\cdot)$ directly transforms S_{in} into S_{tr}, we can bound the response time of an initial job in S_{in} via the response time of its transformed job in S_{tr}.

Theorem 5.1. *If a job has a response time of R_{tr} in S_{tr}, then its response time R_{in} in S_{in} is at most $R_{tr} + \Pi - \Theta$.*

Proof. Let $r_{i,j}$ be release time of the job in S_{in}, and let $f_{i,j}$ be its completion in S_{in}. Then $R_{in} = f_{i,j} - r_{i,j}$. By Lemma 5.5, all of this job's scheduling intervals in S_{in} are transformed via $F(\cdot)$ into scheduling intervals in S_{tr}. Thus, the job is completed at time $F(f_{i,j})$ in S_{tr}, and by definition of the transformed release time, $R_{tr} = F(f_{i,j}) - (r_{i,j} + \Pi - \Theta)$. By Lemma 5.4, $F(f_{i,j}) \geq f_{i,j}$, so $R_{tr} \geq f_{i,j} - (r_{i,j} + \Pi - \Theta) = R_{in} - (\Pi - \Theta)$. Thus, $R_{in} \leq R_{tr} + \Pi - \Theta$. □

Step 5 of Figure 5.1 does not affect task utilizations, so thus far, no changes to schedulability conditions are required. However, such changes are inevitable because the initial reservation restricts processing supply. These changes occur in the step discussed next.

5.3.2 Abstracting Reduced-Speed Platform

In this subsection, we describe Step 6 of Figure 5.1. We transform the reduced-speed platform the identical multiprocessor platform (changing utilizations of tasks). Recall that on the reduced-speed platform, each CPU has speed Θ/Π. Thus, we can easily rescale these speeds to 1.0 by multiplying by the factor Π/Θ, which requires correspondingly multiplying each WCET by Π/Θ (thus, the utilization of task τ_i on the identical multiprocessor platform is defined as $u'_i = u_i \cdot \Pi/\Theta$). This step allows us to completely abstract the reservation and consider the scheduling of Ω on an identical multiprocessor platform with m_Ω CPUs with WCETs that have been inflated to account for HAC accesses.

By Theorem 5.1, Step 5 preserves the system schedulability. Step 6 preserves the schedule, so it preserves schedulability too. Thus, the schedulability conditions for Ω before Steps 5 and 6 can be derived from those of the system after Step 6 with the following lemma.

Lemma 5.6. *The rp-sporadic task set obtained after Step 4 of Figure 5.1 is schedulable if and only if* $u_i \leq \Theta/\Pi \cdot P_i$ *and* $U \leq \Theta/\Pi \cdot m_\Omega$.

Proof. As mentioned before, the task set after Step 4 of Figure 5.1 is schedulable if and only if the task set obtained after Step 6 of Figure 5.1 is schedulable. The latter condition follows from (3.1): $\forall i\ u'_i \leq P_i$ and $U' \leq m_\Omega$, where $U' = \Pi/\Theta \cdot U$ is the modified system's utilization, and $u'_i = u_i \cdot \Pi/\Theta$ is the modified task τ_i's utilization. Rearrangement yields the lemma statement. □

To get the response-time bounds of the tasks in S_{tr}, we can use Theorem 3.1 or Corollary 3.1 from Chapter 3 assuming m_Ω fully available CPUs.

5.4 Factoring Out Tasks

We described how to work with HAC accesses in Section 5.2 using a locking protocol. Unfortunately, this approach may produce unsuitably large bounds in some specific cases. In this section, we describe

Optimization Step 2 of Figure 5.1, which may reduce such bounds or even make the system schedulable. We demonstrate one such specific case with the following example.

Example 5.3. Consider a system with a single component Ω comprised of 15 tasks with continuous access (a simplification to avoid forbidden zones) to eight CPUs and a single HAC. Assume that each task performs 1 time unit of CPU execution and has a period of 30 time units (thus, from the CPUs' point of view, the task set has CPU utilization of 0.5).

Additionally, assume that eight of the 15 tasks must also access the HAC once per job for 2 time units. Thus, we have seven CPU-only tasks ($C = 1, T = 30$) and eight CPU-and-HAC tasks ($C = 1, B = 2, T = 30$). According to Lemma 5.1, each HAC-accessing task has its WCET increased by $2 \cdot (2 \cdot 8 - 1) = 30$ under the global OMLP. Thus, each HAC-accessing task's utilization increases by $30/30 = 1.0$, leading to a total utilization increase of $8 \cdot 1.0 = 8.0$. The increase in total utilization due to blocking alone consumes the capacity of all eight allocated CPUs, making the task set unschedulable.

This would be exacerbated if supply was not continuous under Ω's reservation due to additional blocking from forbidden zones. $\Diamond$

The reason behind such a utilization explosion in Example 5.3 is that the increase in utilization caused by lock-related blocking scales with the number of HAC-accessing tasks *multiplied* by the number of CPUs. Moreover, this increase is proportional to B_{max} (Definition 5.1). Thus, a single long HAC access in Ω increases the task WCET inflation for all accesses (even for the short ones[5]). This encourages *factoring out* the problematic HAC and its associated tasks into a subsystem with the smallest possible CPU count, namely one. Tasks accessing that HAC are then scheduled non-preemptively and busy-wait on the dedicated CPU during their accesses, ensuring that jobs access the HAC immediately upon request (thus removing the need for a locking protocol).

Example 5.3 (cont'd). Suppose instead that the seven CPU-only tasks are scheduled on seven CPUs and the eight HAC-accessing tasks are scheduled exclusively on a dedicated CPU (we say they are factored out). Each factored out task has WCET equal to $1 + 2 = 3$ time units (CPU execution plus the HAC access time), and a utilization of $3/30$ (no blocking inflation is needed anymore), totaling to $8(3/30) = 24/30$ for all HAC-accessing tasks. The seven CPU-only tasks can clearly be scheduled on the seven remaining CPUs.

[5]Note that "skipping ahead" (mentioned in Section 5.2.3) improves only the forbidden-zone-related blocking time in Lemma 5.3 and does not affect the lock-related blocking X.

The dedicated CPU has greater capacity (1.0) than the HAC-accessing tasks (total utilization of $24/30$), so the system is schedulable (under NP-EDF). ◊

Of course, not all systems will have only one HAC as in Example 5.3. If, after factoring out a single HAC, total utilization remains high due to accesses to other HACs, then we can simply continue to factor out HACs onto other dedicated CPUs until the remaining subsystem becomes schedulable. We provide a response-time analysis needed for the tasks on dedicated CPUs in Section 5.4.1 (which addresses the need to account for the reservation model and forbidden zones).

Choosing HACs to factor out. Factoring out HACs has disadvantages in that some capacity will inevitably be lost due to partitioning tasks onto dedicated CPUs, making the choice of which HACs to factor out a trade-off. We assume that the number of HACs is small enough that considering every possibility of factoring out HACs is feasible (note that for any two HACs accessed by the same task, the fact that that task cannot be managed both by the OMLP and via a dedicated CPU means that either both or neither HAC must be factored out). Our proposed approach iterates through every possible factoring and chooses the factoring of HACs that best satisfies a given metric (*e.g.,* the lowest maximum or average response-time bound). However, if checking of all factorings is impossible, we propose to factor out tasks that make long HAC accesses.

The intuition behind preferentially assigning factored-out HACs to tasks with long accesses is to reduce B_{max} (as the logic behind factoring out itself is to reduce m_Ω) that results in reducing blocking under the global OMLP (Lemma 5.3). As such blocking inflates jobs execution times, reducing blocking should reduce the total utilization. Although our discussion here has focused on using the global OMLP, the principles above apply regardless of the choice of locking protocol.

Factoring assumption. Being able to factor out HACs is predicated upon the assumption that all tasks that access a specific HAC fit on a single CPU. This is not a strong assumption for some AI applications, such as deep learning (*e.g.,* (Bochkovskiy et al., 2020)), which tend to be dominated by GPU workloads and require negligible CPU execution in comparison. Furthermore, if a HAC's tasks overutilize a dedicated CPU and their CPU execution is negligible compared to their HAC execution, then the HAC must already be executing nearly continuously. This implies that HAC capacity is the limiting factor for schedulability, in which case adding more CPUs is ineffective.

Multiple HAC groups. Task groups corresponding to multiple HACs may be partitioned on the same CPU by logically combining the HACs into a single virtual HAC (HAC requests from tasks on a dedicated CPU are never simultaneous under our policy).

5.4.1 Response-Time Bounds of Factored-Out Tasks

In this subsection, we provide response-time analysis for the set of factored-out tasks τ_{fo} on a platform that has a single HAC $\mathcal{H}$ and its dedicated CPU $\mathcal{C}$. We assume the non-preemptive Earliest-Deadline-First (NP-EDF) scheduler. However, we only forbid tasks in said group from preempting each other; in-progress jobs are still preempted at the end of reservation boundaries. We still use forbidden zones to guarantee in-progress execution at the boundary is preemptible CPU work but incorporate them into the analysis rather than using Lemma 5.3.

We assume that tasks busy-wait on dedicated CPUs while accessing HACs. Note that, under uniprocessor NP-EDF scheduling, no busy-waiting occurs prior to an actual HAC execution. This means their WCETs must be inflated by their worst-case HAC access times. Also, any HAC request should be immediately satisfied on $\mathcal{H}$ (as any job that accesses it must have first occupied $\mathcal{C}$). As HAC accesses are abstracted as workload on $\mathcal{C}$, we analyze the scheduling on $\mathcal{C}$.

Definition 5.6. Let B^{fo}_{max} be the duration of the longest HAC access made by any job of a task in τ_{fo}, and C^{fo}_{max} be the longest WCET of a task in τ_{fo}.

Analysis is complicated by the fact that $\mathcal{H}$ may not be available while $\mathcal{C}$ is available due to forbidden zones. We abstract away from this complication by treating the dedicated CPU-HAC pair $\mathcal{C}$ and $\mathcal{H}$ as a single resource (though at runtime, CPU workloads may still be executed in the forbidden zone of $\mathcal{H}$). However, all the workload of tasks in τ_{fo} must fit on one CPU or else that CPU is overutilized.

Definition 5.7. Let $V = \Pi - \Theta + B^{\text{fo}}_{max}$. V denotes the longest duration of time when $\mathcal{H}$ is unavailable within a reservation period Π.

For the purpose of abstracting $\mathcal{C}$ and $\mathcal{H}$ as a single resource, we (analytically) assume that $\mathcal{C}$, like $\mathcal{H}$, may be unavailable for at most V time units per Π time units. This analytically wastes CPU capacity when $\mathcal{C}$ is available while $\mathcal{H}$ is in its forbidden zone, but this is not particularly damaging for AI algorithms that make extensive use of HACs.

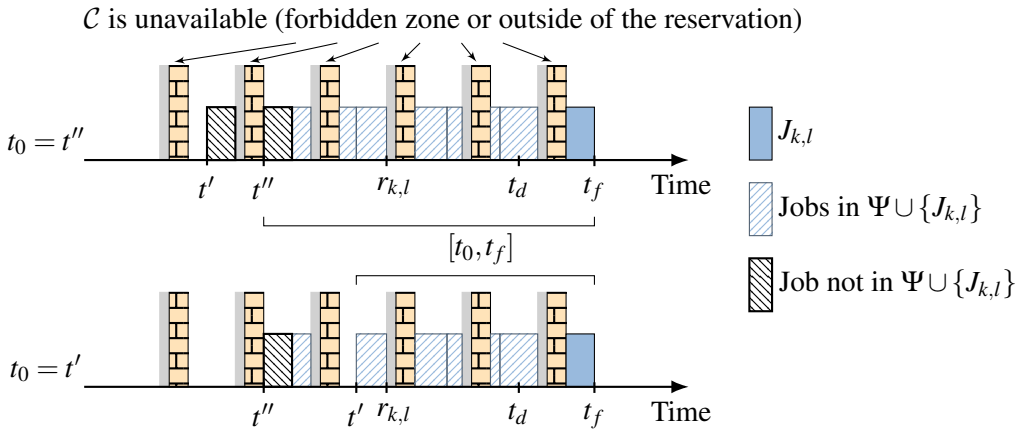


Figure 5.11: Important proof points.

Consider a job of interest $J_{k,l}$ scheduled on $\mathcal{C}$ with release time $r_{k,l}$, absolute deadline t_d, and completion time t_f. Our goal is to provide an upper bound on the response time of $J_{k,l}$, given by $t_f - r_{k,l} = t_f - t_d + T_k$. We prove our bound by contradiction in Theorem 5.2 after proving bounds on the amount of work that can delay $J_{k,l}$ (Lemma 5.7) and on the amount of work completed by time t_f (Lemma 5.8).

Definition 5.8. Denote the set of all jobs of tasks of τ_{fo} with priority higher than $J_{k,l}$ as Ψ.

By the definition of NP-EDF, all jobs in $\Psi \cup \{J_{k,l}\}$ are released before t_d. Thus, outside of one potential lower-priority job, discussed later, only jobs of $\Psi \cup \{J_{k,l}\}$ may delay the execution of $J_{k,l}$ inside the invocations of the reservation. We say that a job is *ready* at time t if it is released at or before t and not completed at t.

Definition 5.9. Let t' be the last time instant before t_d when $\mathcal{C}$ is idle[6] during τ_{fo}'s reservation invocation (or time 0 if no such time instant exists). Let t'' be the start of the last scheduling interval before t_d when a job not in $\Psi \cup \{J_{k,l}\}$ is scheduled on $\mathcal{C}$ (or time 0 if no such interval exists. t' and t'' are illustrated in Figure 5.11. Let $t_0 = \max(t', t'')$.

Note that no jobs in $\Psi \cup \{J_{k,l}\}$ are ready at t_0 (otherwise one of them would be scheduled). Thus, as $J_{k,l}$ is ready at $r_{k,l}$,

$$t_0 \leq r_{k,l} \tag{5.1}$$

The interval $[t_0, t_f]$ is also illustrated in Figure 5.11 and is referenced in the following proofs. We compute an upper bound on t_f with the following sufficient condition for $J_{k,l}$ to have been completed: if the total

[6] A more mathematically rigorous definition of t' would be "the last time instant before t_d when $\mathcal{C}$ is idle at t'^{-} ...".

workload of all jobs that can be scheduled within the interval $[t_0, t_f]$ is less than the provided capacity, then $J_{k,l}$ is completed within the interval $[t_0, t_f]$.

Definition 5.10. Let U_{fo} be the total utilization of tasks in τ_{fo}. Recall that a job of a task in τ_{fo} occupies $\mathcal{C}$ (busy waiting) during its HAC requests, so U_{fo} accounts for both CPU and HAC execution.

Lemma 5.7. *The total workload of jobs in $\Psi \cup \{J_{k,l}\}$ that can be scheduled within $[t_0, t_f]$ is at most $(t_d - t_0)U_{\text{fo}}$.*

Proof. By the definition of t_0 there were no ready jobs in $\Psi \cup \{J_{k,l}\}$ just before[7] t_0. By the definition of NP-EDF, no jobs of $\Psi \cup \{J_{k,l}\}$ have deadline later than t_d. Thus, all jobs in $\Psi \cup \{J_{k,l}\}$ that execute within $[t_0, t_f]$ have release and deadline within $[t_0, t_d]$.

Each task $\tau_i \in \tau_{\text{fo}}$ releases at most $\lfloor (t_d - t_0)/T_i \rfloor$ jobs that execute within $[t_0, t_f]$. Thus, the total workload of jobs in $\Psi \cup \{J_{k,l}\}$ over $[t_0, t_f]$ is the total workload of jobs in $\Psi \cup \{J_{k,l}\}$ released within $[t_0, t_d]$:

$$\sum_{\tau_i \in \tau_{\text{fo}}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor C_i \leq \sum_{\tau_i \in \tau_{\text{fo}}} \frac{t_d - t_0}{T_i} C_i = (t_d - t_0) U_{\text{fo}}. \qquad \square$$

Now we estimate the total available capacity of $\mathcal{C}$ within $[t_0, t_f]$ spent on jobs in $\Psi \cup \{J_{k,l}\}$.

Definition 5.11. Let $S_{\text{fo}} = 1 - V/\Pi$ be the long-run rate for which $\mathcal{H}$ and $\mathcal{C}$ are both available. Note that we require $U_{\text{fo}} \leq S_{\text{fo}}$, as otherwise $\mathcal{C}$ is overutilized.

Lemma 5.8. *The available capacity of $\mathcal{C}$ for jobs in $\Psi \cup \{J_{k,l}\}$ within $[t_0, t_f]$ is at least $(t_f - t_0)S_{\text{fo}} - V - C^{\text{fo}}_{max}$.*

Proof. The total capacity of $\mathcal{C}$ within this interval is $t_f - t_0$. By the definition of t_0, because the scheduler is NP-EDF, the first job executed in this interval may not be in $\Psi \cup \{J_{k,l}\}$ (*e.g.,* when $t_0 = t''$), which induces at most C^{fo}_{max} of capacity loss (see the black hatched execution in Figure 5.11). As we work within a periodic reservation, $\mathcal{C}$ may not be available for $\lceil (t_f - t_0)/\Pi \rceil$ intervals of length V (striped intervals in Figure 5.11). Thus, the guaranteed available capacity of $\mathcal{C}$ for jobs in $\Psi \cup \{J_{k,l}\}$ is at least

$$(t_f - t_0) - \left\lceil \frac{t_f - t_0}{\Pi} \right\rceil V - C^{\text{fo}}_{max}.$$

With $\left\lceil \frac{t_f - t_0}{\Pi} \right\rceil \leq \frac{t_f - t_0}{\Pi} + 1$ and the definition of S_{fo}, we get the lemma statement. $\square$

Now we can bound the response time of $J_{k,l}$.

[7] A more mathematically rigorous definition of "just before t_0" would be "at t_0^-".

Theorem 5.2. *If $S_{\text{fo}} \geq U_{\text{fo}}$, then the response time of any job of τ_k on $\mathcal{C}$ is at most*

$$T_k + \frac{V + C^{\text{fo}}_{max} - T_k(S_{\text{fo}} - U_{\text{fo}})}{S_{\text{fo}}}. \tag{5.2}$$

Proof. We prove the theorem by contradiction. Suppose otherwise that $t_f - r_{k,l}$ exceeds (5.2). Consider the interval $[t_0, t_f]$.

In the following derivations, we derive necessary and sufficient conditions (5.3) on t_f such that the guaranteed capacity $(t_f - t_0)S_{\text{fo}} - V - C^{\text{fo}}_{max}$ (Lemma 5.8) for jobs of $\Psi \cup \{J_{k,l}\}$ over $[t_0, t_f]$ exceeds the upper bound on the workload of jobs in $\Psi \cup \{J_{k,l}\}$ in this interval, $(t_d - t_0)U_{\text{fo}}$ (Lemma 5.7).

$$\begin{aligned}
&(t_d - t_0)U_{\text{fo}} && \leq (t_f - t_0)S_{\text{fo}} - V - C^{\text{fo}}_{max} \\
\Leftrightarrow &V + C^{\text{fo}}_{max} + (t_d - t_0)U_{\text{fo}} && \leq (t_f - t_0)S_{\text{fo}} \\
\Leftrightarrow &V + C^{\text{fo}}_{max} + (t_d - t_0)U_{\text{fo}} && \leq (t_f - t_d + t_d - t_0)S_{\text{fo}} \\
\Leftrightarrow &V + C^{\text{fo}}_{max} + (t_d - t_0)(U_{\text{fo}} - S_{\text{fo}}) && \leq (t_f - t_d)S_{\text{fo}} \\
\Leftrightarrow &\frac{V + C^{\text{fo}}_{max} + (t_d - t_0)(U_{\text{fo}} - S_{\text{fo}})}{S_{\text{fo}}} && \leq t_f - t_d \\
\Leftrightarrow &\frac{V + C^{\text{fo}}_{max} - (t_d - t_0)(S_{\text{fo}} - U_{\text{fo}})}{S_{\text{fo}}} && \leq t_f - r_{k,l} - T_k
\end{aligned} \tag{5.3}$$

Note that $t_f - r_{k,l}$ is the response time of $J_{k,l}$. By (5.1), $t_0 \leq r_{k,l}$. Thus, $t_d - t_0 \geq t_d - r_{k,l} = T_k > 0$. By Definition 5.11, $S_{\text{fo}} - U_{\text{fo}} \geq 0$. Thus,

$$-(t_d - t_0)(S_{\text{fo}} - U_{\text{fo}}) \leq -T_k(S_{\text{fo}} - U_{\text{fo}}). \tag{5.4}$$

By our assumption $t_f - r_{k,l}$ exceeds (5.2), so by (5.4), (5.3) holds. As (5.2) is a sufficient condition for the capacity provided by $\mathcal{C}$ to jobs of $\Psi \cup \{J_{k,l}\}$ to exceed the workload required by this job set over $[t_0, t_f]$, $\mathcal{C}$ must have been idle[8] in $[t_0, t_f]$. This contradicts the definition of t_0, meaning the response time of $J_{k,l}$ must be at most (5.2).

Note that (5.2) does not depend on job number, so it applies to any job of any task $\tau_k \in \tau_{\text{fo}}$. □

[8]We accounted for the first job potentially being not in $\Psi \cup \{J_{k,l}\}$ in Lemma 5.8; any other job not in $\Psi \cup \{J_{k,l}\}$ cannot be scheduled by the definition of t''.

Action	Section	Reference	Step
Transform the graph set Γ to the DAG set Γ'	Section 4.1.1	–	Step 1
Obtain the rp-sporadic task set τ from Γ'	Section 4.1.2	–	Step 2
Inflate WCET of each task in τ (locking + forbidden zones)	Section 5.2.1	Lemma 5.3	Steps 3, 4
Inflate WCET of each task in τ Π/Θ times	Section 5.3.2	–	Step 6
Compute per-task response-time bound	Section 3.2	Theorem 3.1	Step 7
Increase bound of each task in τ by $\Pi - \Theta$ (releases shift)	Section 5.3.1	Theorem 5.1	Step 5
Compute per-node offsets	Section 4.1.2	(4.1)	Step 8
Compute per-graph response-time bounds	Section 4.1.3	(4.2)	Step 9

Table 5.1: The bound computation in practice.

5.5 The Bound Computation Process

In this section, we describe the whole response-time bound computation process shown in Figure 5.1. We consider scheduling of a component Ω containing a graph task set in a reservation Λ.

Our computation process contains nine required and two optional steps. We designed it this way to simplify the usage of existing research. Consider, for example, Steps 5, 6, and 7. We separated them to ensure that an alternative response-time analysis for the identical multiprocessor can be used in place of our analysis proposed in Section 3.2. Alternatively, we could have proven the analysis proposed in Section 3.2 step-by-step already assuming partial supply (and forbidden zone usage), effectively merging Steps 5, 6, and 7 into a single step.

However, when a better analysis idea for some task model is discovered, authors usually consider the most general (or popular) task model that works with the found idea. Thus, if a better rp-sporadic analysis is proposed, we expect it to consider a continuous supply, meaning the new analysis can only be retrofitted[9] into our analysis because of the separation of Steps 5, 6, and 7. Moreover, our approach allows the usage of an alternative sequential real-time task model with a completely different[10] analysis.

Bound computation process review. We now review the logic behind the bound computation process step-by-step. We also mention bound computation steps needed in practice in Tbl. 5.1. All steps referenced here follow Figure 5.1.

[9]We require this new analysis to support early-releasing and non-preemptive sections.

[10]We also require this analysis to support early-releasing and non-preemptive sections.

We start with a graph task set Γ. In Step 1, we transform each graph into a DAG using the process described in Section 4.1.1: we replace forward and backward dependencies in the graph with regular dependencies. Then, if the obtained graph has any cycles, we merge them into supernodes.

We continue with the DAG task set Γ'. In Step 2, we define per-node offsets (as unknown but constant values). These offsets are used to abstract the DAG model, transforming each DAG into a set of independent tasks. If the set of independent tasks τ is schedulable under any release pattern, then (4.2) ensures the existence of per-node offsets, and the schedulability of the initial collection of DAGs.

We can also apply Optimization Step 1 to some DAGs (i.e., merging nodes) before the transformation to τ, as described in Section 4.3, to reduce the final response-time bound.

We continue with the set of independent rp-sporadic tasks τ. In Step 3, we abstract the HAC accesses with a locking protocol, transforming tasks in τ into CPU-only tasks with non-preemptive sections (Lemma 5.1). In Step 4, we abstract away temporal isolation requirements for HACs using forbidden zones (Lemma 5.3). In both steps, we inflate the WCETs of tasks.

We continue with the set of CPU-only rp-sporadic tasks τ'. In Step 5, we abstract the partial supply model, transforming partial supply into a continuous one. We simultaneously transform tasks releases and deadlines to ensure that response-time bounds under the original partial supply are bounded by response-time bounds under the continuous supply (Theorem 5.1). In Step 6, we abstract the reduced speed of the continuous supply by inflating WCETs and supply simultaneously.

We continue with the set of CPU-only rp-sporadic tasks τ'' on an identical multiprocessor. In Step 7, we directly apply the analysis of Section 3.2 to get the response-time bounds R_i of tasks in τ''.

We also can apply Optimization Step 2 to some tasks of τ (*i.e.,* factoring out tasks accessing HACs) before the transformation to τ', as described in Section 5.4, to reduce the final response-time bound. The response-time bounds of tasks factored out in Optimization Step 2 are computed independently from the remaining tasks.

Using bounds R_i in Steps 8 and 9, we compute offsets of DAGs and per-graph response-time bounds.

5.6 Experiments

We conducted several experiments to evaluate the efficacy of the approaches proposed in this chapter. We assumed a reservation ($\Theta = 20$ and $\Pi = 40$) with m_Ω (which varies over the experiments) CPUs and two HACs. By Lemma 5.6, the feasibility condition of the system is $U \leq \Theta/\Pi \cdot m_\Omega = 0.5 m_\Omega$.

Generation. We followed the generation procedure explained in Section 4.4 with minor changes. We consider an additional parameter p_g—the probability of a node to be a HAC node. A HAC node has a single HAC access equal to its generated WCET. For each experiment, we generated 8,000 task sets (for each line in the following plots).

Utilization. To apply our analysis, we need to inflate the WCETs of each HAC node after generation according to Lemma 5.3. However, in the following plots, we plotted based on the task set utilization before the inflation (the *generated utilization*) for a better visualization of results. Thus, multiple tasks sets with utilization close to the limit of $0.5 m_\Omega$ are not schedulable, as inflation makes their total utilization exceed the total capacity.

5.6.1 Schedulability

In the first experiment, we checked the schedulability of the generated task sets with $m_\Omega = 4, 6, 8$ and $p_g = 0.05, 0.1, 0.15, 0.2$. We used two DAGs with 60 total nodes as a workload. Results can be found in Figure 5.12.

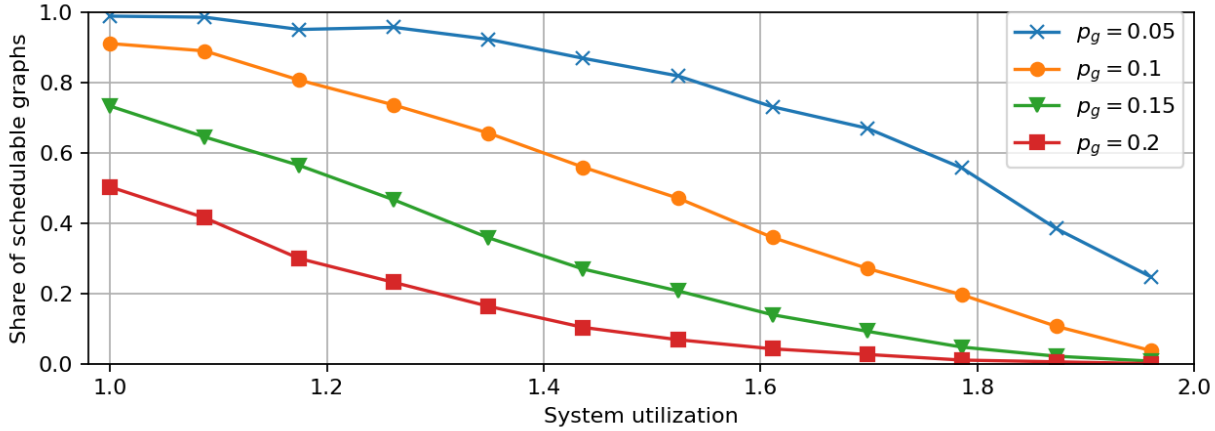
Observation 5.1. Task sets with $p_g = 0.05$ have the highest chance to be schedulable (very few HAC nodes) with any generated utilization; task sets with $p_g = 0.2$ are less than 50% (and for larger m_Ω, 20%) schedulable and are barely schedulable at all for large generated utilizations.

We expected this result: the higher p_g means a higher number of HAC nodes and, therefore, a larger total inflation.

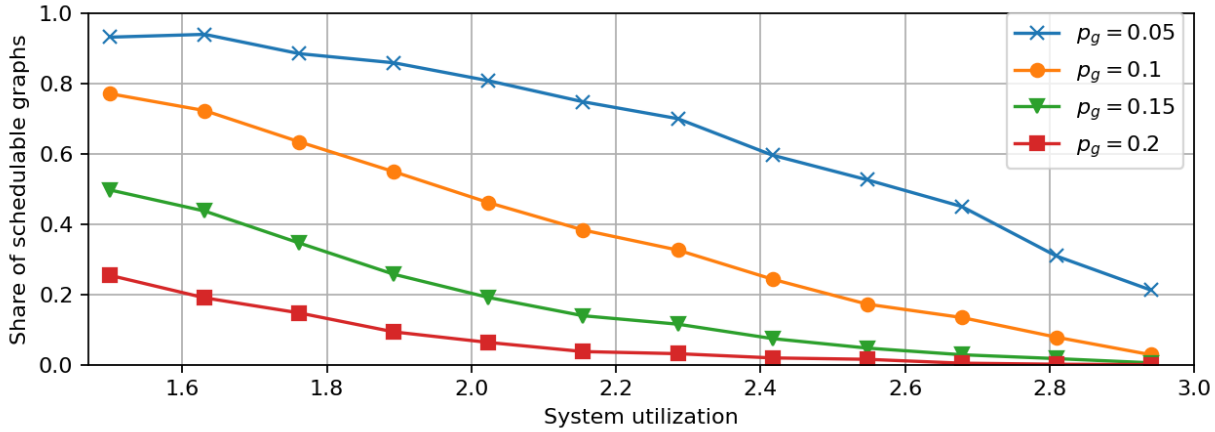
Observation 5.2. Assuming the same p_g, the share of schedulable task sets decreases with the increased number of CPUs.

The observation can be explained by Lemma 5.2: the inflation per HAC node is proportional to the number of CPUs in the reservation. The two observations above and Figure 5.12 result in the following.

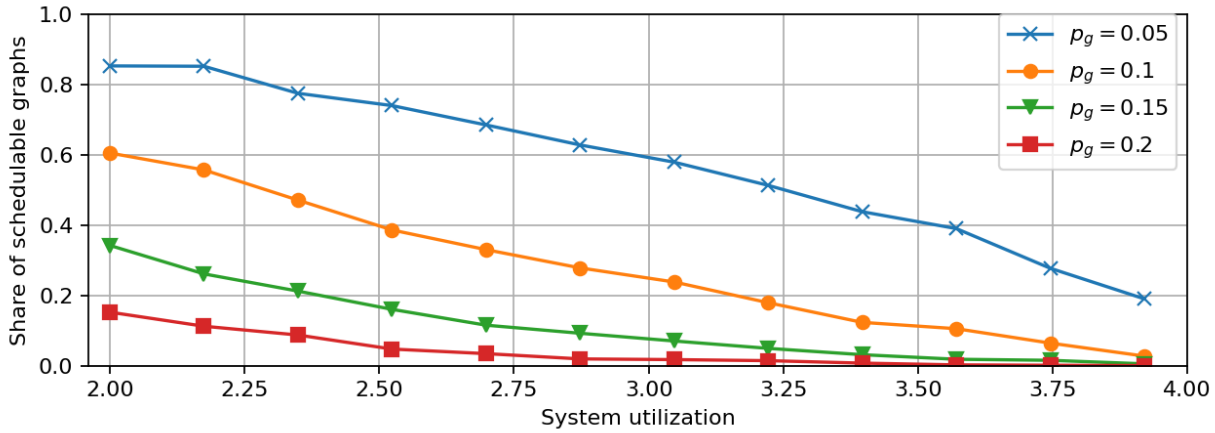
Observation 5.3. The WCET inflation is a significant factor negatively affecting task sets' schedulability.



(a) $m_\Omega = 4$.



(b) $m_\Omega = 6$.



(c) $m_\Omega = 8$.

Figure 5.12: The schedulabilty of task sets. The maximal possible utilization of a schedulable task set is $0.5m_\Omega$.

The schedulability effect of factoring out. In the second experiment, we evaluated the effect of "factoring out." We factored out one HAC with one CPU; we picked tasks with the longest WCETs to be factored out (the number of tasks to be factored out is determined by Definition 5.11). We generated task sets with $m_\Omega = 4, 6, 8$ and $p_g = 0.05, 0.1, 0.15, 0.2$. We used two DAGs with 60 total nodes as a workload. Results of the experiment are shown in Figure 5.13.

Observation 5.4. Factoring out even a single HAC *significantly* improves task set schedulability, effectively reducing the WCET inflation for most generated task sets.

Factoring out reduced the number of HAC nodes in the remaining part of the system, reducing the total inflation. Factoring out may increase the share of schedulable sets by up to 80%!

Observation 5.5. For task sets with utilizations close to the system limit of $0.5m_\Omega$, factoring out may decrease schedulablilty. The effect is stronger for lower values of m_Ω.

Recall that factoring out uses one HAC and one CPU for its scheduling. For systems with small m_Ω (*e.g.,* 4), it significantly reduces CPU computing capacity (*e.g.,* by 25%).

We also conducted the same experiment with fixed $m_\Omega = 6$ and varying the number of nodes. Results can be found in Figure 5.14.

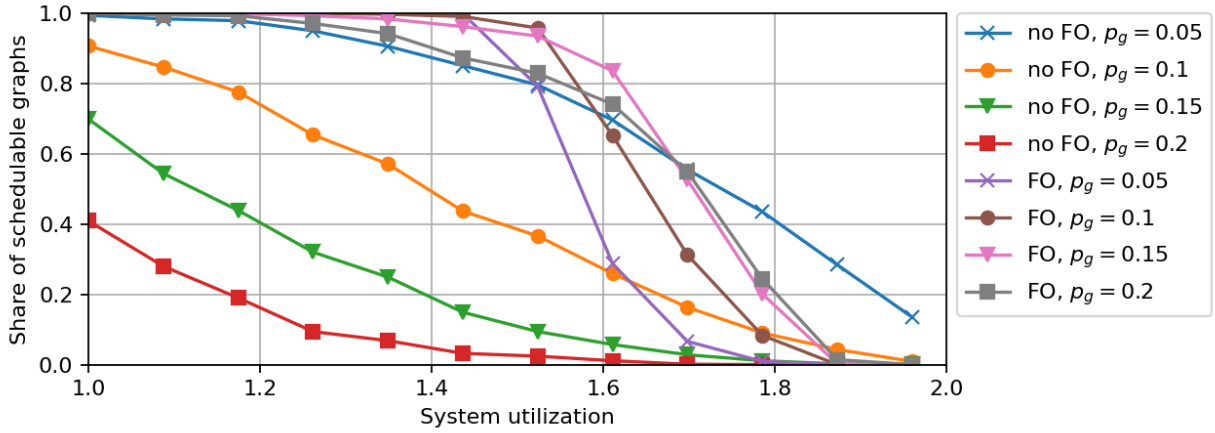
Observation 5.6. Improvement due to factoring out is smaller for smaller graphs.

The response-time bound effect of factoring out. In the experiment above we considered the schedulability effect of factoring out. However, the factored out nodes have a different per-node response-time bound (obtained with Theorem 5.2). Thus, for task sets that are schedulable with and without factoring out, we can compute the net change in bound for a graph after factoring out. Result are shown in Figure 5.15.

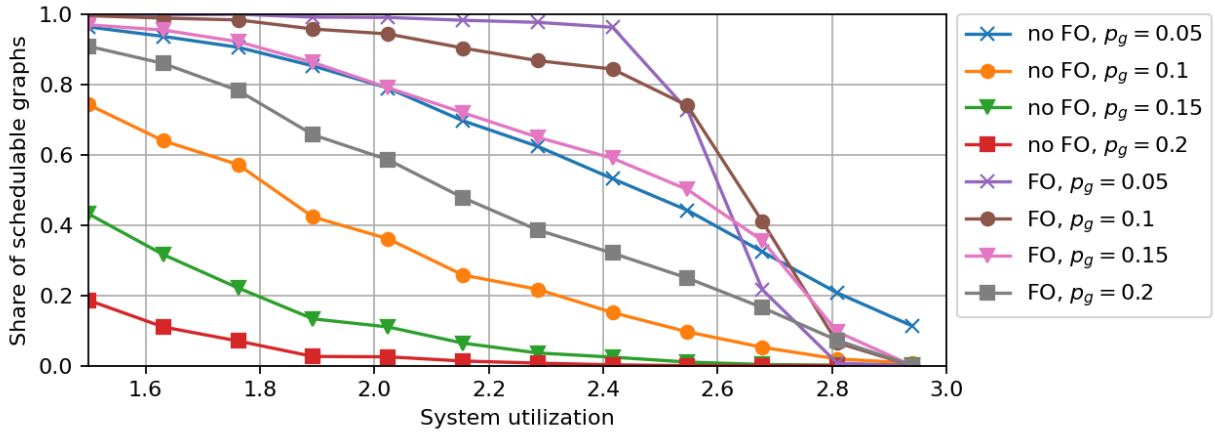
As we need to consider a significant number of task sets, we did not generate task sets with utilization near $0.5m_\Omega$.

Observation 5.7. Factoring out alone improves response-time bounds by 10-30%. The effect is stronger for systems with larger m_Ω.

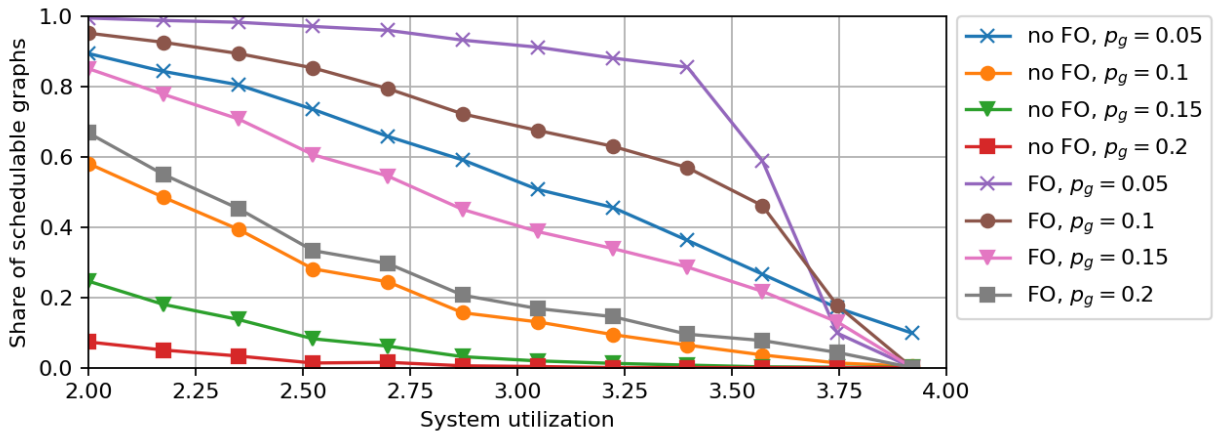
Factoring out reduces the total inflation of the task set at the cost of one HAC and one CPU. On average, factored out nodes have smaller response-time bounds.



(a) $m_\Omega = 4$.

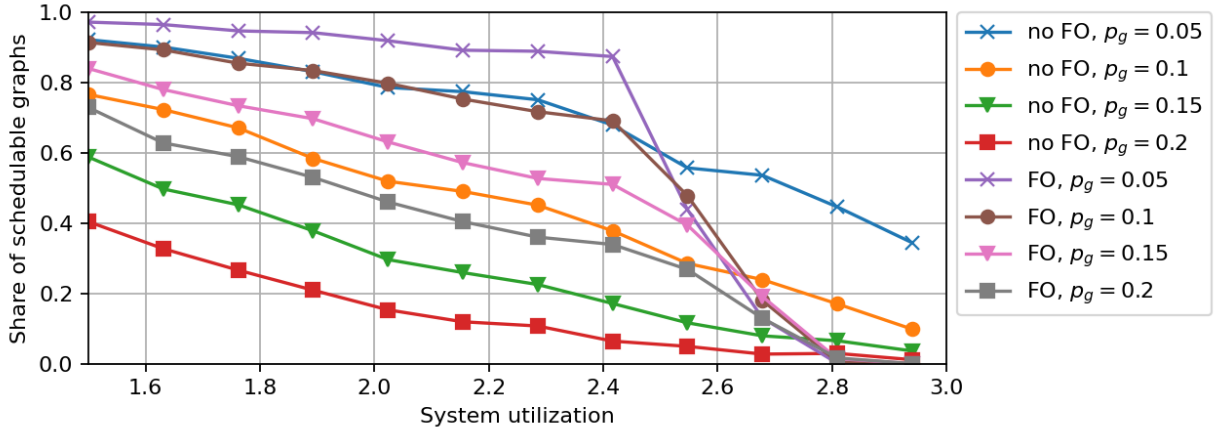


(b) $m_\Omega = 6$.

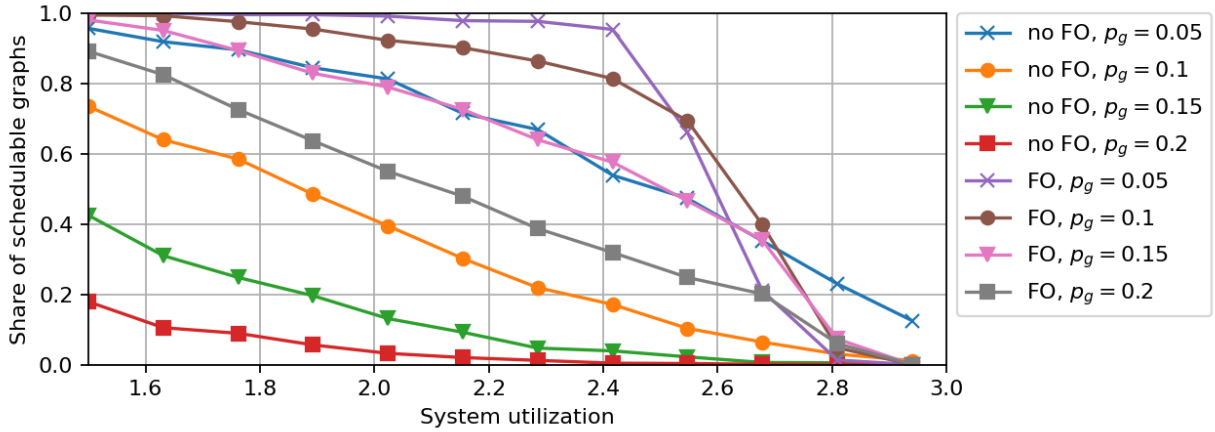


(c) $m_\Omega = 8$.

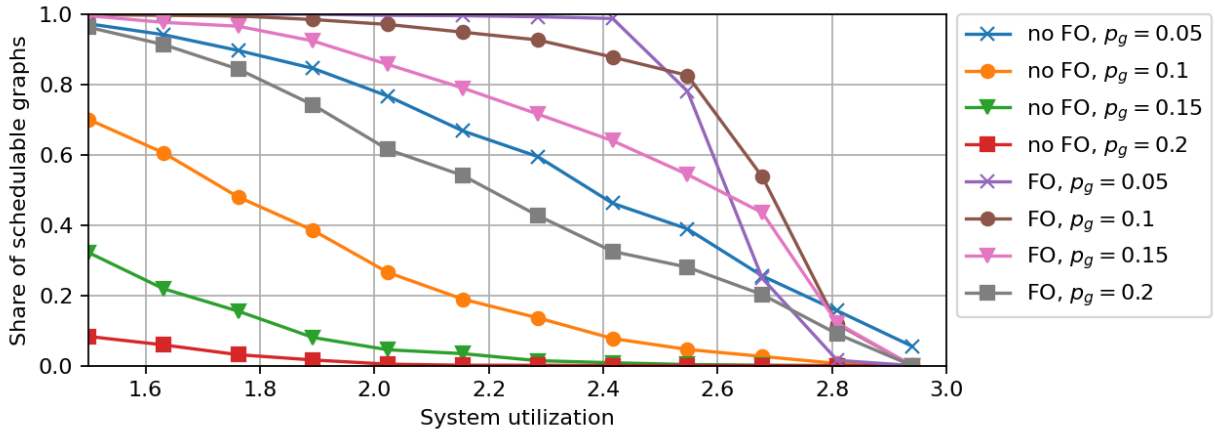
Figure 5.13: The effect of factoring out on the task set schedulability. The total number of nodes is 60.



(a) Small graphs (set parameters: 2 DAGs, 25 total nodes).

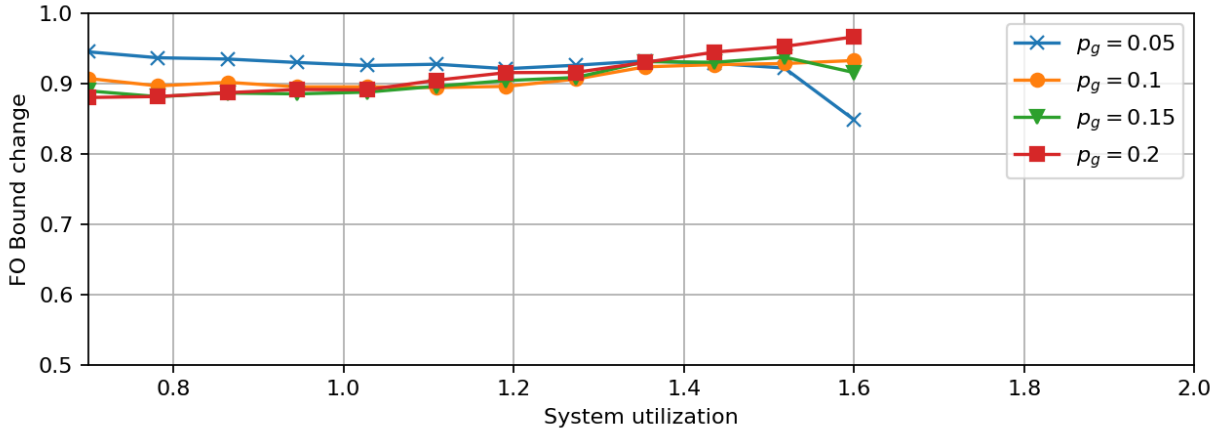


(b) Medium graphs (set parameters: 2 DAGs, 55 total nodes).

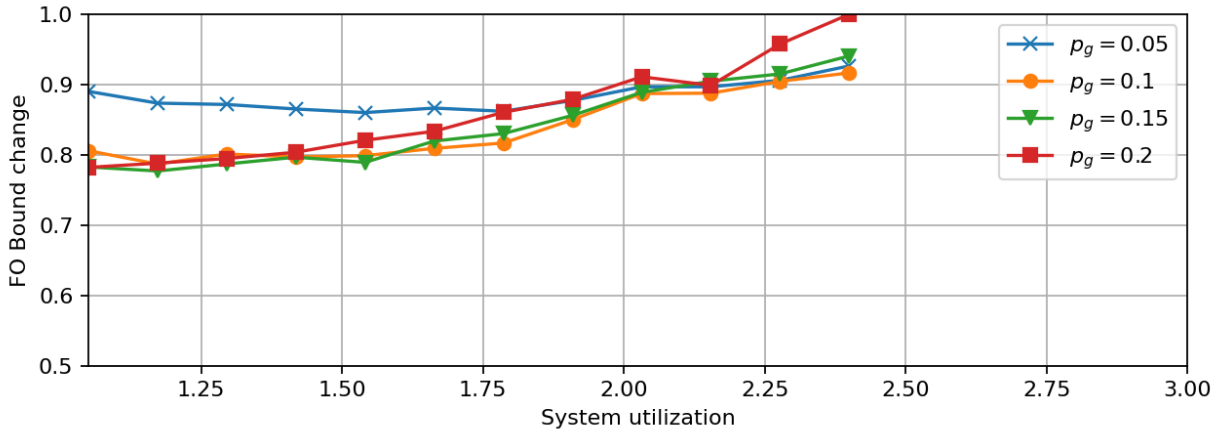


(c) Large graphs (set parameters: 2 DAGs, 80 total nodes).

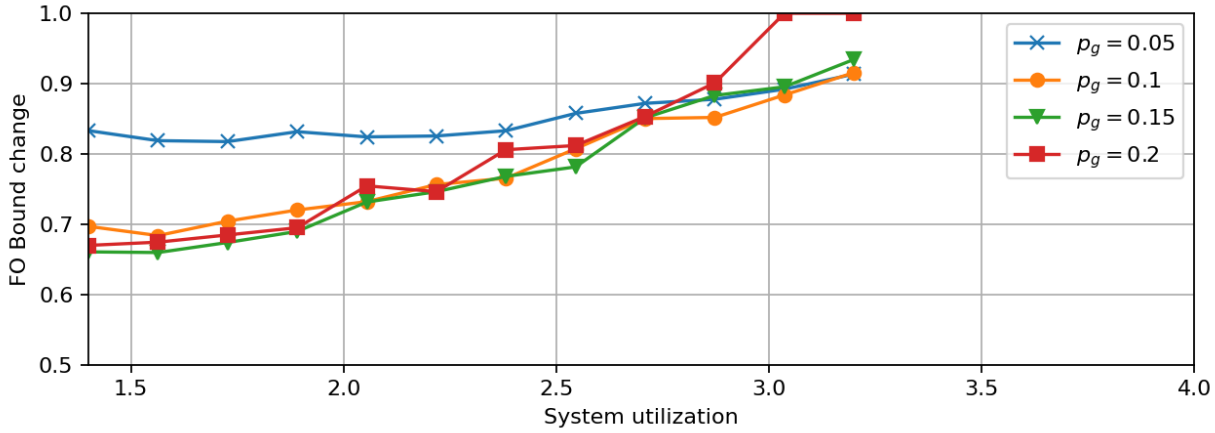
Figure 5.14: The effect of factoring out on the task set schedulability. $m_\Omega = 6$; maximal possible utilization of a schedulable task set is 3.0.



(a) Small graphs (set parameters: 2 DAGs, 25 total nodes).

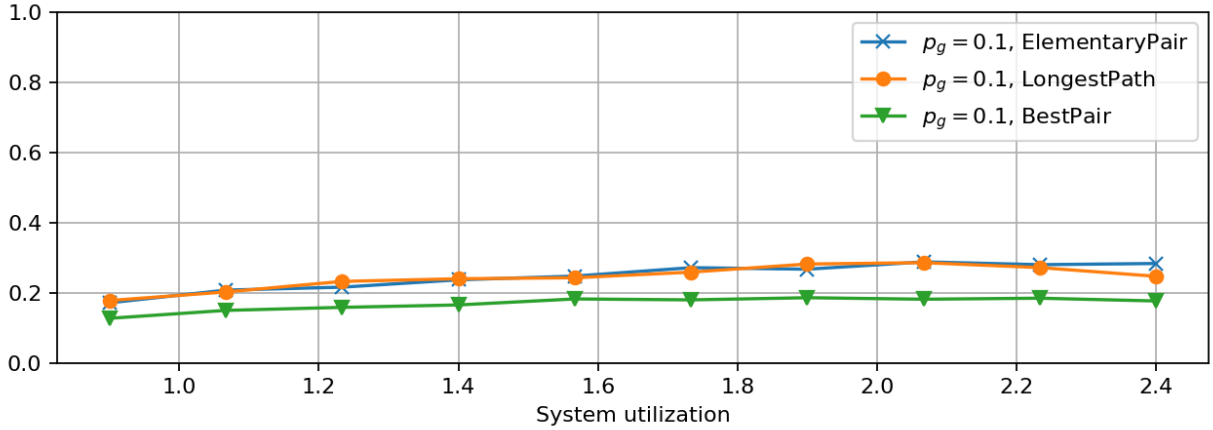


(b) Medium graphs (set parameters: 2 DAGs, 55 total nodes).

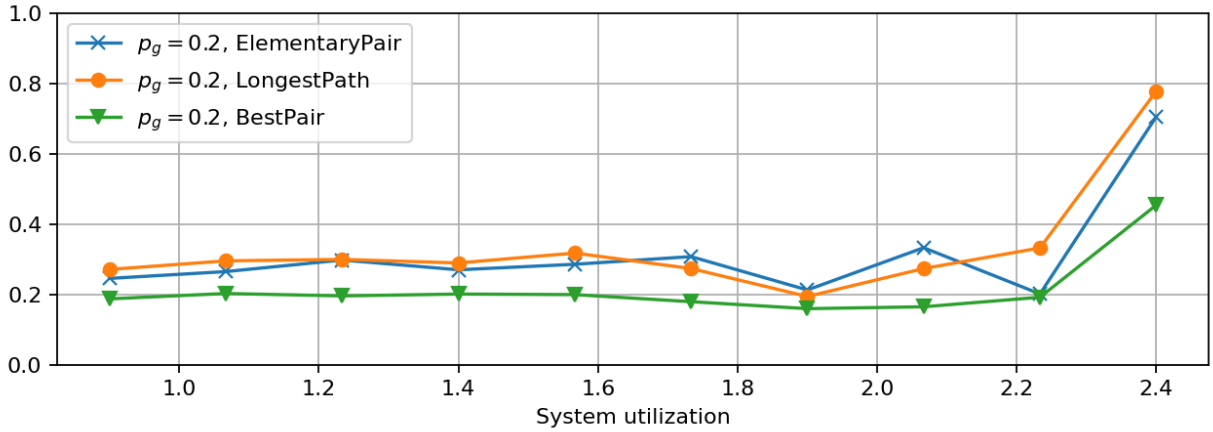


(c) Large graphs (set parameters: 2 DAGs, 80 total nodes).

Figure 5.15: The response-time bound compared to the bound before the factoring out



(a) $p_g = 0.1$.



(b) $p_g = 0.2$.

Figure 5.16: Heuristics performance.

5.6.2 Node Merging

In this section, we consider node merging in the context of periodic reservation. Recall that node merging does not improve schedulability. Because of the scheduling process described in Section 5.5, each task's response-time bound is inflated by $\Pi - \Theta$. Thus, each merging of two consecutive nodes (as in Example 4.3) improves the graph bound approximately by $T + \Pi - \Theta$ instead of T (assuming a small change in the per-task bound value x—see Theorem 3.1).

To show that the node merging approach works in the presence of HAC accesses, we consider the scheduling of 60-node graphs on $m_\Omega = 6$ CPUs with $\Theta = 20$ and $\Pi = 40$ in Figure 5.16.

Observation 5.8. As with the experiments in Section 4.4, BestPair shows the best performance, while ElemntaryPair and LongestPath have similar performance. All heuristics significantly decrease response-time bounds.

The small differences among the heuristic results is explained by the small total system utilization, under which many nodes can be merged together, and the positions of HAC nodes. The position of HAC nodes (which have the highest expected utilization due to locking-based inflation) in the graph structure are of importance because merging high-utilization nodes is more likely to violate schedulability requirements.

5.7 HAC Accesses as Scheduling Entities

In both Sections 5.2 and 5.4, we consider HAC accesses as synchronization objects (*i.e.,* CPU-centric scheduling). However, it is possible to also view these accesses as schedulable entities as well and apply separate analysis to the CPU and HAC parts of tasks. We explain the complexities related to the accesses as schedulable entities approach in this section. Generally, HAC accesses can be viewed as schedulable entities independent of (via task splitting) or combined with (via self-suspending analysis) CPU workloads.

Task splitting. The first approach is to split each task into CPU-only and HAC-only tasks as done in (Yang et al., 2018). An example of such splitting can be found in Figure 5.17. In the context of our work, task splitting requires us to introduce offsets for all new nodes (Step 2 of Figure 5.1), increases the lengths of some graph paths and, by (4.2), may significantly increase response-time bounds (see also Example 4.3 and Section 4.3). The node merging proposed in Section 4.3 may reduce the increase in bounds, but each HAC access in the node of the longest path increases its length by at least one node.

Moreover, this approach may require heavy internal modification of the tasks. For example, typical HAC accesses require some CPU-HAC data transfer before and after HAC computations (*e.g.,* kernel launch and stream sync for NVIDIA GPUs). An offset introduced by splitting may postpone data transfer from a HAC; this may affect other accesses (*e.g.,* if HAC memory is limited and the returned result is large).

If task splitting is used, we can use the analysis in Section 5.4.1 for factored-out HAC accesses in a reservation almost without modification. The only change required is to set the CPU workloads of all considered tasks to zero.

Note that the node splitting proposed in (Yang et al., 2018) resulted in analytically improved schedulability mostly because of the increased parallelization level: Yang *et al*. used the npc-sporadic task model ($P = m$)

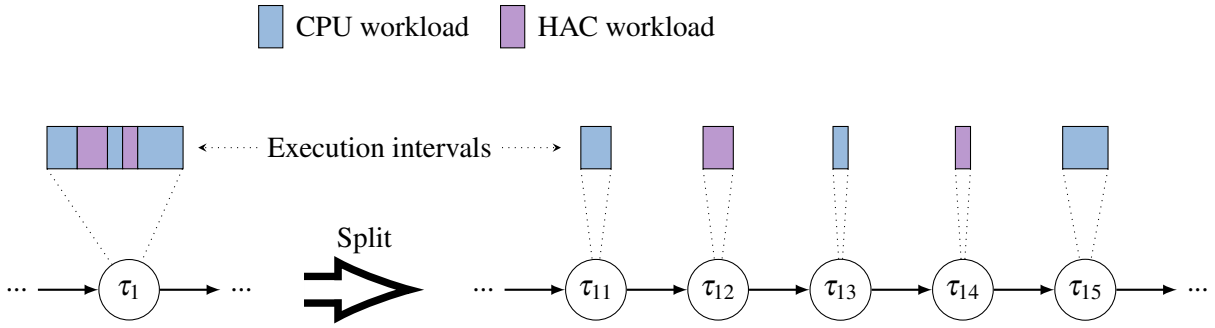


Figure 5.17: Example of a task split.

and compared against (Yang et al., 2015), which used the sporadic task model ($P = 1$, inherited from (Elliott et al., 2014)).

Self-suspension analysis. The second approach is to consider tasks with HAC accesses as self-suspending CPU tasks in CPU analysis, and self-suspending tasks in HAC analysis. However, this creates some surprising analysis difficulties, as illustrated next.

Example 5.4. Consider an rp-sporadic task depicted in Figure 5.18 in a system with one CPU and one HAC. By the considered approach, we must consider a given task from both a CPU perspective—in which case time accessing a HAC is suspension time away from CPU execution—and from a HAC perspective—in which case time executing on a CPU is suspension time away from HAC execution. Determining such suspension times requires determining HAC and CPU response times, respectively. Thus, we have a circularity: in order to determine CPU and HAC response times, we need to know CPU and HAC response times!

Another option is to compute response-time bounds iteratively: compute CPU response-time bound R first (assuming WCETs of HAC accesses as a bound on self-suspension times), then compute a HAC response-time bound using R as a self-suspension time, then compute a new CPU response-time bound R' (using the updated HAC bound), and so on. The first three stages of the computation are illustrated in Figure 5.18. ◊

We think that the reason for these difficulties lies in the nature of self-suspending tasks. For example, many papers on self-suspending tasks (Liu and Anderson, 2012) or (Chen et al., 2019, Chapter 4.1.1) have to consider at least some suspension time as execution time. Thus, the response-time of tasks on HACs affects the self-suspension time of tasks on CPUs and leads to $R' > R$. This process continues with $R'' > R'$ and so on. Unfortunately, the series $R, R', R'', \ldots$ of bounds we tried to consider appeared to be divergent.

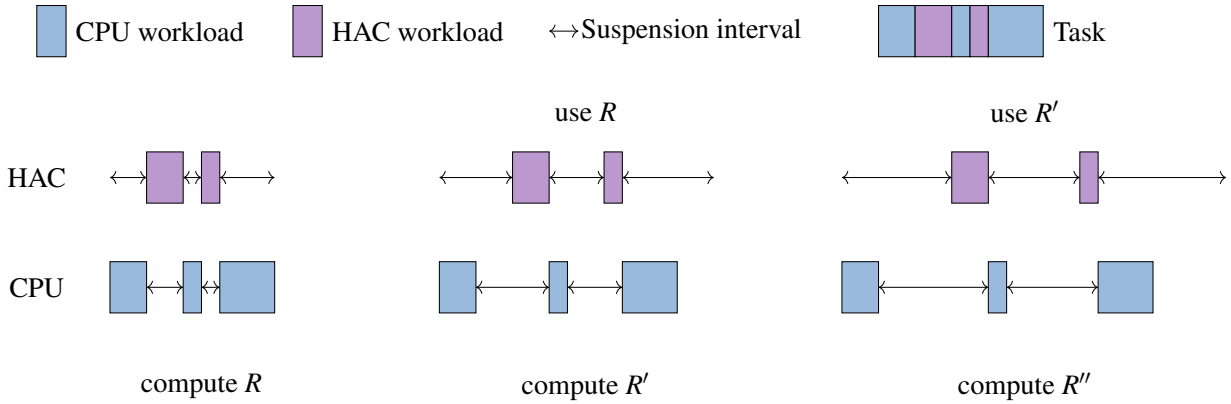


Figure 5.18: A schedule of the task from Example 5.4.

5.8 Conclusion

In this chapter, we have presented a time-partitioned scheduling approach for graph task sets deployed on multicore+accelerator platforms. We used a multiprocessor locking protocol to ensure temporal isolation for system components. We proposed an analytical transformation procedure to use existing response-time analysis in the context of partial supply. We proposed a way to improve system schedulability and demonstrated its efficacy via experiments. We also discussed the existence of other scheduling approaches.

Acknowledgements. The work presented in this chapter is the result of a collaboration among Sergey Voronov, Tanya Amert, and Stephen Tang. Voronov and Amert provided the forbidden zone blocking analysis in Section 5.2.3. Voronov provided the system transformation that abstracts the reservation supply model in Section 5.3, and the analysis for the factored-out tasks in Section 5.4. Some experiments presented in Section 5.6 were designed collaboratively by Voronov and Tang.

The scheduling approach presented in this chapter was tested on real hardware (these experiments are not presented in this dissertation); these experimental results can be found in (Amert, 2021, Section 5.4). The implementation was designed and implemented by Tanya Amert, Zelin (Peter) Tong, and Joshua Bakita.

CHAPTER 6: CONCLUSION

Three overlapping issues are crucial for the development of certification standards for safety-critical systems with heavy CV applications on board: real-time scheduling analysis of graph-based workloads, hardware accelerator usage, and component isolation. In this dissertation, we addressed these three issues to bring the future of certification closer.

First, we have introduced a task model that can enables response-time analysis for a broad range of graph-based workloads. Second, we incorporated accelerator usage into our model, and proposed a graph modification method for a response-time bound reduction in graph-based workloads. We experimentally demonstrated that this method achieves a significant bound reduction. Finally, we proposed a general scheduling approach for a single component to ensure temporal isolation in the presence of non-preemptive accelerator accesses; the approach employs a modular design that simplifies changing of its steps with other existing methods (*e.g.,* changing in-place analysis). We also provided our own response-time analysis for a single system component.

In the rest of the this chapter, we summarize the results presented in this dissertation (Section 6.1), mention other related work of the author that was not included in this dissertation (Section 6.2), and detail directions for future work (Section 6.3).

6.1 Summary of Results

In Chapter 1, we presented the following thesis

> *Time-isolation guarantees can be provided for multi-component systems with graph-based applications in the presence of non-preemptive accelerator accesses; various scheduling choices can significantly improve the schedulability of such systems.*

We now summarize the contributions presented in this dissertation in support of this thesis.

The rp-sporadic task model. Existing real-time task models may not be suitable for some graph-based workloads: applications with specific dependencies among nodes (*e.g.,* dependencies requiring that nodes

be merged to avoid cycles) and applications with low response-time bound requirements. The traditional task models are either too strict (the sporadic task model) to support merging away cycles or too lenient (the npc-sporadic task model) to describe applications' needs.

In Section 2.3.2, we introduced the rp-sporadic task model, which generalizes the sporadic and npc-sporadic task models, allowing for precise control over the tradeoff between schedulability and algorithmic requirements. In Chapter 3, we provided a response-time analysis for a system of rp-sporadic tasks that contain non-preemptive sections (to support future HAC accesses).

Graph response-time bound reduction through node merging. In Chapter 4, we explained the existing SRT scheduling approach for graph-based workloads. Unfortunately, this approach inevitably results in large response-time bounds for large graphs. We proposed a way to reduce these bounds through a graph structure modification: merging nodes of the graph into supernodes.

We conducted an experimental evaluation of the node merging approach and observed significant reductions in the response-time bounds (up to 5–10$\times$ for graphs with a large number of nodes). We demonstrated bound reductions in all cases where nodes had high potential for merging without breaking task set schedulability. Note that careful tuning of the rp-sporadic task model parameters may enable higher reductions.

A response-time bound computation framework for a single component. In Chapter 5, we presented a framework that enables in-component response-time bound computation of rp-sporadic tasks with accelerator accesses using existing techniques that would otherwise be incompatible with the reservation model (*e.g.,* a response-time analysis assuming the identical multiprocessor model). The framework's design is motivated by easing the future usage of novel analyses / approaches for each step. We used a series of abstraction steps to abstract the initial graph, in-task accelerator accesses, temporal isolation requirements of the component, and the partial supply of the computing resources to enable the usage of response-time bound analysis assuming an identical multiprocessor platform.

Response-time bound reduction through factoring out some accelerator accesses. In Section 5.4, we proposed a way to enable schedulablity and reduce response-time bounds for task sets with specific accelerator access patterns (*e.g.,* long accesses or a large number of accesses). We factored out some tasks with accelerator accesses onto a specific CPU to reduce the locking-based WCET inflation for the rest of the task set. We conducted experiments on factoring out such tasks and observed a significant improvement in schedulability.

6.2 Other Publications

In this section, we summarize other research contributions, not presented in this dissertation, of which the author has been a part. We start with two publications briefly mentioned before.

Implementation of the proposed scheduling approach.[1] We considered the practical application of the scheduling approach presented in Chapter 5. We presented TimeWall, a time-partitioning framework for multiprocessor+accelerator platforms ensuring temporal isolation, which can help enable component-wise certification (when applied alongside existing methods for alleviating spatial interference). We considered HAC-abstraction methods described in Section 5.2 and demonstrated isolation properties provided by TimeWall on a real platform via a case study of a computer-vision perception application.

Parallelization level vs response-time bound vs. accuracy trade-off.[2] In this dissertation, we consider graph-based task systems, whose nodes are represented by rp-sporadic tasks (the model described in Section 2.3.2). We considered the choice of parallelization levels P for the system to be given by the system designer based on the application requirements. For example, in the context of CV applications, the acceptable algorithm accuracy may be considered as a requirement.

We evaluated various choices of P for a CV tracking application. Larger P may negatively affect tracking accuracy but improve response-time bounds of the system or even ensure the schedulability of a graph that would otherwise be unschedulable (from the real-time scheduling point of view). We found that allowing $P > 1$ did not significantly affect the CV tracking application's accuracy with some runtime tweaks (modify the application to consume the most recent task data available, which may be older than the application was designed for), but did greatly reduce analytical response-time bounds. In practice, it is observed that fresh data is often available, even for tasks with $P > 1$.

[1] This work appeared in the following paper:

Amert, T., Tong, Z., Voronov, S., Bakita, J., Smith, F. D., and Anderson, J. H. (2021a). Timewall: Enabling time partitioning for real-time multicore+accelerator platforms. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 455–468.

[2] This work appeared in the following paper:

Amert, T., Yang, M., Voronov, S., Nandi, S., Vu, T., Anderson, J. H., and Smith, F. D. (2021b). The price of schedulability in cyclic workloads: The history-vs.-response-time-vs.-accuracy trade-off. *Journal of Systems Architecture, Article 102292*, 120.

6.2.1 Response-Time Analysis

In this subsection, we describe response-time analysis research that falls outside the scope of this dissertation. In the rest of the subsection, we assume SRT scheduling (see the definition in Section 2.2.2).

G-FP scheduling of npc-sporadic tasks on an identical multiprocessor.[3] Global fixed-priority schedulers (G-FP) are often preferable to dynamic-priority ones because they entail less overhead, are easier to implement, and enable certain tasks to be favored over others. However, under G-FP with the standard sporadic task model, response times of low-priority tasks may be unbounded, even if the total task system utilization is low. We proved that such unbounded response times can be circumvented by usage of the npc-sporadic task model. We presented a response-time bound for npc-sporadic task systems that requires only that the total utilization does not exceed the overall processing capacity. In other words, we showed that G-FP is SRT-optimal under the npc-sporadic task model.

Optimal semi-partitioned scheduling on an identical multiprocessor with affinities.[4] Modern operating systems allow task migrations to be restricted by specifying per-task processor affinity masks (a mask specifies the set of processor CPUs upon which a task can be scheduled). We proposed an SRT-optimal semi-partitioned (only certain tasks are allowed to migrate) scheduler called AM-Red (affinity mask reduction). AM-Red is the first (non-clairvoyant) optimal scheduler for sporadic task sets with arbitrary affinity masks. It reduces the affinities of the task set without compromising feasibility offline allowing at most $m-1$ migrating (*i.e.,* with affinity of more than a single CPU) tasks on m CPUs. With a tunable frame size of F, AM-Red is SRT-optimal for any F (task set tardiness is bounded by F) and HRT-optimal for some F (if F divides all task periods). We showed that the total number of migrations is $O(m)$ per frame, and in some cases (acyclic or hierarchical affinity graphs) AM-Red's offline time complexity is optimal.

[3]This work appeared in the following papers:

Voronov, S., Anderson, J. H., and Yang, K. (2018). Tardiness bounds for fixed-priority global scheduling without intra-task precedence constraints. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 8–18.

Voronov, S., Anderson, J. H., and Yang, K. (2021a). Tardiness bounds for fixed-priority global scheduling without intra-task precedence constraints. *Real-Time Systems*, 57(1):4–54.

[4]This work appeared in the following paper:

Voronov, S. and Anderson, J. H. (2018). An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 408–420.

First polynomial G-EDF resp.-time bounds on identical with affinities and uniform multiprocessors.[5] Prior work has shown that G-EDF is SRT-optimal for sporadic task systems on uniform multiprocessor (each CPU has its own fixed speed) establishing exponentially large bounds (Yang and Anderson, 2017). However, identical multiprocessors with affinity masks have received little attention. Such a platform is particularly compelling as noted by Peter Zijlstra in a list of important open problems in his keynote talk at ECRTS 2017 (in the context of Linux's G-EDF implementation). We generalized the existing proof strategy for the uniform platform yielding the first polynomial G-EDF response-time bounds for the uniform multiprocessor and the first such bounds of any kind for identical multiprocessors with affinities.

G-EDF scheduling on unrelated multiprocessors.[6] Identical multiprocessors with affinities and uniform multiprocessors can be generalized as unrelated multiprocessors that define a unique speed for each task-CPU pair. As the default definition of G-EDF on platforms different from identical multiprocessors is vague, we proposed a G-EDF variant that generalizes both G-EDF variants considered in (Tang et al., 2019). We proposed an approximation of the proposed unrelated G-EDF variant and proved that it is at least nearly SRT-optimal, deriving response-time bounds that scale inversely with the slack in the system (*i.e.,* bounds go to infinity as a task system approaches infeasibility).

6.3 Future Work

There are multiple directions for future work based on the contributions of this dissertation.

Precise self-supension analysis of rp-sporadic tasks. Note that under the OMLP locking policy, a job actually suspends on making a HAC access request; suspension is beneficial for the actual execution of the system (as compared to the FIFO spinlock, which busy-waits) but results in worse analytical bounds due to inflation of WCETs. Thus, a self-suspention analysis of rp-sporadic tasks may improve the component's schedulability and the per-task response-time bounds for some task sets if the OMLP (or another suspension-

[5]This work appeared in the following paper:

Tang, S., Voronov, S., and Anderson, J. H. (2019). GEDF tardiness: Open problems involving uniform multiprocessors and affinity masks resolved. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 13:1–13:21.

[6]This work appeared in the following paper:

Tang, S., Voronov, S., and Anderson, J. H. (2021). Extending EDF for soft real-time scheduling on unrelated multiprocessors. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 253–265.

based lock) is used. Note that the self-suspension analysis-related complexities discussed in Section 5.7 arise specifically from the CPU-HAC circularity and not the suspensions themselves.

Flexible component reservation model. In this dissertation, we considered a strictly periodic component reservation. While this type of reservation suits our needs with bounded response times, other types of reservations may work better with other task models. For example, systems using task models with probabilistic parameters (*e.g.,* WCETs are defined through probabilistic density functions) may benefit from an alternative reservation model.

Automated system decomposition into components. The response-time analysis presented in Section 5 assumes component parameters are given. However, this analysis serves only as a first (but necessary) step for the general approach of certifiable components. Recall that the initial non-decomposed system has a continuous supply of computing resources. While the workloads within a given component are likely to be grouped together on a per-application basis (*e.g.,* path planning or pedestrian tracking in a self-driving car), and are thus fixed, the allocation of system resources to workloads (*i.e.,* the reservation parameters) is a tunable parameter in the design space of the system that should be computed by some optimization procedure (*e.g.,* optimizing for schedulability). Despite the fact that the problem "Can we pack reservations together?" appears to be NP-hard (Baker et al., 1980) even if all reservations share the same period Π, the total number of reservations should be relatively small. Thus, a brute force or a polynomial approximation method can be used to define reservation parameters under some simplifying assumptions.

The grand optimization problem. The previous problem is stated as optimizing decomposition of the system into components to make the system schedulable. However, this problem is only one facet of the problem of holistically optimizing for schedulability and reducing response-time bounds. Several methods can be applied simultaneously towards optimizing this objective. First, some tasks can be factored out to one or several HACs to increase the schedulability of the system. Second, some graph nodes can be merged to reduce response-time bounds. Third, different priority points can be used (changing from G-EDF to a G-EDF-like scheduler). These priority points can be optimized to reduce response-time bounds. Fourth, the exact configuration of reservations that leads to the schedulable task system with the lowest response-time bound can be optimized in the presence of the mentioned methods. All these methods must be applied simultaneously because the efficacy of one method depends on the results of the other methods. A grand

optimization problem can be defined to optimize this system. While it is unlikely to have a nice closed-from solution, it can be solved directly for small systems and heuristically for larger systems.

BIBLIOGRAPHY

Abeni, L., Balsini, A., and Cucinotta, T. (2019). Container-based real-time scheduling in the Linux kernel. *ACM SIGBED Review*, 16(3):33–38.

Ahmed, S. and Anderson, J. H. (2021). Tight tardiness bounds for pseudo-harmonic tasks under global-EDF-like schedulers. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*, pages 11:1–11:24.

Ahmed, S. and Anderson, J. H. (2022). Exact response-time bounds of periodic DAG tasks under server-based global scheduling. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 447–459.

Ahmed Bhuiyan, A., Yang, K., Arefin, S., Saifullah, A., Guan, N., and Guo, Z. (2019). Mixed-criticality multicore scheduling of real-time Gang task systems. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 469–480.

Akesson, B., Nasri, M., Nelissen, G., Altmeyer, S., and Davis, R. I. (2020). An empirical survey-based study into industry practice in real-time systems. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*.

Akesson, B., Nasri, M., Nelissen, G., Altmeyer, S., and Davis, R. I. (2022). A comprehensive survey of industry practice in real-time systems. *Real-Time Systems*, 58(3):358–398.

Amert, T. (2021). *Enabling real-time certification of autonomous driving applications*. PhD thesis, University of North Carolina at Chapel Hill.

Amert, T., Otterness, N., Yang, M., Anderson, J. H., and Smith, F. D. (2017). GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 104–115.

Amert, T., Tong, Z., Voronov, S., Bakita, J., Smith, F. D., and Anderson, J. H. (2021a). Timewall: Enabling time partitioning for real-time multicore+accelerator platforms. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 455–468.

Amert, T., Voronov, S., and Anderson, J. H. (2019). OpenVX and real-time certification: The troublesome history. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 312–325.

Amert, T., Yang, M., Voronov, S., Nandi, S., Vu, T., Anderson, J. H., and Smith, F. D. (2021b). The price of schedulability in cyclic workloads: The history-vs.-response-time-vs.-accuracy trade-off. *Journal of Systems Architecture, Article 102292*, 120.

Aronsson, P. and Fritzson, P. (2003). Task merging and replication using graph rewriting. In *Proceedings of the 10th International Workshop on Compilers for Parallel Computers*.

Audsley, N. C., Burns, A., Davis, R. I., Tindell, K. W., and Wellings, A. J. (1995). Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198.

Baker, B. S., Coffman, Jr, E. G., and Rivest, R. L. (1980). Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855.

Baker, T. P. and Baruah, S. K. (2009). An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems. *Real-Time Systems*, 43(1):3–24.

Bakita, J. and Anderson, J. H. (2022). Enabling GPU memory oversubscription via transparent paging to an NVMe SSD. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 370–382.

Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509–512.

Baruah, S. (2014). Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systemss*, pages 97–105.

Baruah, S. (2015). The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proceedings of the 19th Design, Automation and Test in Europe Conference and Exhibition*, pages 1323–1328.

Baruah, S. (2020). Scheduling DAGs when processor assignments are specified. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, pages 111–116.

Baruah, S., Bonifaci, V., and Marchetti-Spaccamela, A. (2015). The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 222–231.

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., and Wiese, A. (2012). A generalized parallel task model for recurrent real-time processes. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium*, pages 63–72.

Baruah, S. K., Cohen, N. K., Plaxton, C. G., and Varvel, D. A. (1993). Proportionate progress: A notion of fairness in resource allocation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 345–354.

Basaran, C. and Kang, K.-D. (2012). Supporting preemptive task executions and memory copies in GPGPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 287–296.

Bhuiyan, A., Guo, Z., Saifullah, A., Guan, N., and Xiong, H. (2018). Energy-efficient real-time scheduling of DAG tasks. *ACM Transactions on Embedded Computing Systems*, 17(5):1–25.

Bhuiyan, A., Liu, D., Khan, A., Saifullah, A., Guan, N., and Guo, Z. (2020). Energy-efficient parallel real-time scheduling on clustered multi-core. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2097–2111.

Bi, R., He, Q., Sun, J., Sun, Z., Guo, Z., Guan, N., and Tan, G. (2022). Response time analysis for prioritized DAG task with mutually exclusive vertices. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 460–473.

Bini, E., Bertogna, M., and Baruah, S. (2009). Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 437–446.

Bini, E. and Buttazzo, G. C. (2005). Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154.

Biondi, A., Balsini, A., Pagani, M., Rossi, E., Marinoni, M., and Buttazzo, G. (2016). A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, pages 1–12.

Blazewicz, J., Drabowski, M., and Weglarz, J. (1986). Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers*, 35(05):389–393.

Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. M. (2020). YOLOv4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*.

Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S., and Wiese, A. (2013). Feasibility analysis in the sporadic DAG task model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 225–233.

Brandenburg, B. B. and Anderson, J. H. (2010). Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60.

Brandenburg, B. B. and Anderson, J. H. (2013). The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342.

Brandenburg, B. B. and Gül, M. (2016). Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, pages 99–110.

Burmyakov, A., Bini, E., and Tovar, E. (2014). Compositional multiprocessor scheduling: the GMPR interface. *Real-Time Systems*, 50(3):342–376.

Buttazzo, G., Bini, E., and Wu, Y. (2011). Partitioning real-time applications over multicore reservations. *IEEE Transactions on Industrial Informatics*, 7(2):302–315.

Capodieci, N., Cavicchioli, R., Bertogna, M., and Paramakuru, A. (2018). Deadline-based scheduling for GPU with preemption support. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 119–130.

Casini, D., Blaß, T., Lütkebohle, I., and Brandenburg, B. B. (2019). Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*.

Chakraborty, S., Künzli, S., and Thiele, L. (2003). A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the 7th Design, Automation and Test in Europe Conference*, volume 3.

Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., and McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.

Chang, S., Bi, R., Sun, J., Liu, W., Yu, Q., Deng, Q., and Gu, Z. (2022a). Toward minimum WCRT bound for DAG tasks under prioritized list scheduling algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):3874–3885.

Chang, S., Sun, J., Hao, Z., Deng, Q., and Guan, N. (2022b). Computing exact WCRT for typed DAG tasks on heterogeneous multi-core processors. *Journal of Systems Architecture, Article 102385*, 124.

Chang, S., Zhao, X., Liu, Z., and Deng, Q. (2020). Real-time scheduling and analysis of parallel tasks on heterogeneous multi-cores. *Journal of Systems Architecture, Article 101704*, 105.

Chen, G., Zhao, Y., Shen, X., and Zhou, H. (2017). EffiSha: A software framework for enabling effficient preemptive scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16.

Chen, J.-J. and Liu, C. (2014). Fixed-relative-deadline scheduling of hard real-time tasks with self-suspensions. In *Proceedings of the 35th IEEE Real-Time Systems Symposium*, pages 149–160.

Chen, J.-J., Nelissen, G., Huang, W.-H., Yang, M., Brandenburg, B., Bletsas, K., Liu, C., Richard, P., Ridouard, F., Audsley, N., et al. (2019). Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207.

Collette, S., Cucu, L., and Goossens, J. (2008). Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187.

Dai, Z., Liu, H., Le, Q. V., and Tan, M. (2021). CoAtNet: Marrying convolution and attention for all data sizes. *Advances in Neural Information Processing Systems*, 34:3965–3977.

Danne, K. and Platzner, M. (2005). Periodic real-time scheduling for FPGA computers. In *Proceedings of the 3rd International Workshop on Intelligent Solutions in Embedded Systems*, pages 117–127.

Derafshi, D., Norollah, A., Khosroanjam, M., and Beitollahi, H. (2019). HRHS: A high-performance real-time hardware scheduler. *IEEE Transactions on Parallel and Distributed Systems*, 31(4):897–908.

Devi, U. C. and Anderson, J. H. (2005). Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341.

Devi, U. C. and Anderson, J. H. (2008). Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189.

Dinh, S., Gill, C., and Agrawal, K. (2020). Efficient deterministic federated scheduling for parallel real-time tasks. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10.

Ditty, M., Karandikar, A., and Reed, D. (2018). NVIDIA's Xavier SoC. In *Hot Chips: A Symposium on High Performance Chips*.

Dong, Z., Yang, K., Fisher, N., and Liu, C. (2021). Tardiness bounds for sporadic Gang tasks under preemptive global EDF scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 32(12):2867–2879.

Ebaid, A., Ammar, R., Rajasekaran, S., and Fergany, T. (2010). Task clustering & scheduling with duplication using recursive critical path approach (RCPA). In *Proceedings of the 10th IEEE International Symposium on Signal Processing and Information Technology*, pages 34–41.

Elliott, G. A. (2015). *Real-time scheduling for GPUs with applications in advanced automotive systems*. PhD thesis, University of North Carolina at Chapel Hill.

Elliott, G. A. and Anderson, J. H. (2014). Exploring the multitude of real-time multi-GPU configurations. In *Proceedings of the 35th IEEE Real-Time Systems Symposium*, pages 260–271.

Elliott, G. A., Kim, N., Erickson, J. P., Liu, C., and Anderson, J. H. (2014). Minimizing response times of automotive dataflows on multicore. In *Proceedings of the 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10.

Elliott, G. A., Ward, B., and Anderson, J. H. (2013). GPUSync: a framework for real-time GPU management. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 33–44.

Erdős, P. and Rényi, A. (1959). On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297.

Erickson, J. P. and Anderson, J. H. (2011). Response time bounds for G-EDF without intra-task precedence constraints. In *Proceedings of the 15th International Conference On Principles Of Distributed Systems*, pages 128–142.

Erickson, J. P. and Anderson, J. H. (2013). Reducing tardiness under global scheduling by splitting jobs. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 14–24.

Erickson, J. P., Guan, N., and Baruah, S. (2010). Tardiness bounds for global EDF with deadlines different from periods. In *Proceedings of the 14th International Conference On Principles Of Distributed Systems*, pages 286–301.

Faragardi, H. R., Lisper, B., Sandström, K., and Nolte, T. (2014). An efficient scheduling of AUTOSAR runnables to minimize communication cost in multi-core systems. In *Proceedings of the 7th International Symposium on Telecommunications*, pages 41–48.

Feitelson, D. G. and Rudolph, L. (1992). Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318.

Ferry, D., Li, J., Mahadevan, M., Agrawal, K., Gill, C., and Lu, C. (2013). A real-time scheduling service for parallel tasks. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 261–272.

Fonseca, J., Nelissen, G., and Nélis, V. (2019). Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems*, 55(2):387–432.

Fonseca, J. C., Nélis, V., Raravi, G., and Pinho, L. M. (2015). A multi-DAG model for real-time parallel applications with conditional execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1925–1932.

Fontanelli, D., Greco, L., and Palopoli, L. (2013). Soft real-time scheduling for embedded control systems. *Automatica*, 49(8):2330–2338.

Forget, J., Boniol, F., Grolleau, E., Lesens, D., and Pagetti, C. (2010). Scheduling dependent periodic tasks without synchronization mechanisms. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 301–310.

Garey, M. R. and Johnson, D. S. (1975). Complexity results for multiprocessor scheduling under resource constraints. *SIAM journal on Computing*, 4(4):397–411.

Giannopoulou, G., Huang, P., Ahmed, R., Bartolini, D. B., and Thiele, L. (2017). Isolation scheduling on multicores: model and scheduling approaches. *Real-Time Systems*, 53:614–667.

Golyanik, V., Nasri, M., and Stricker, D. (2017). Towards scheduling hard real-time image processing tasks on a single GPU. In *Proceedings of the 24th IEEE International Conference on Image Processing*, pages 4382–4386.

Griffin, D., Bate, I., and Davis, R. I. (2020). Generating utilization vectors for the systematic evaluation of schedulability tests. In *Proceedings of the 41th IEEE Real-Time Systems Symposium*, pages 76–88.

Guan, F., Qiao, J., and Han, Y. (2020). DAG-fluid: A real-time scheduling algorithm for DAGs. *IEEE Transactions on Computers*, 70(3):471–482.

Hamdaoui, M. and Ramanathan, P. (1995). A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451.

Han, M., Guan, N., Sun, J., He, Q., Deng, Q., and Liu, W. (2019). Response time bounds for typed DAG parallel tasks on heterogeneous multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2567–2581.

Han, M., Zhang, H., Chen, R., and Chen, H. (2022). Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558.

Han, M., Zhang, T., Lin, Y., and Deng, Q. (2021). Federated scheduling for typed DAG tasks scheduling analysis on heterogeneous multi-cores. *Journal of Systems Architecture, Article 101870*, 112.

Hartmann, C. and Margull, U. (2019). GPUart - an application-based limited preemptive GPU real-time scheduler for embedded systems. *Journal of Systems Architecture*, 97:304–319.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the 32nd IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778.

He, Q., Guan, N., Guo, Z., et al. (2019). Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295.

He, Q., Guan, N., Lv, M., Jiang, X., and Chang, W. (2022). Bounding the response time of DAG tasks using long paths. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 474–486.

He, Q., Lv, M., and Guan, N. (2021). Response time bounds for DAG tasks with arbitrary intra-task priority assignment. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*.

He, Q., Sun, J., Guan, N., Lv, M., and Sun, Z. (2023). Real-time scheduling of conditional DAG tasks with intra-task priority assignment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Hesse, C. N. (2021). Analysis and comparison of performance and power consumption of neural networks on CPU, GPU, TPU and FPGA. Master's thesis, University of Hildesheim.

Holman, P. and Anderson, J. H. (2005). Adapting pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564.

Holman, P. and Anderson, J. H. (2006). Locking under Pfair scheduling. *ACM Transactions on Computer Systems*, 24(2):140–174.

Hu, B., Cao, Z., and Zhou, M. (2021). Energy-minimized scheduling of real-time parallel workflows on heterogeneous distributed computing systems. *IEEE Transactions on Services Computing*, 15(5):2766–2779.

Jaffe, J. M. (1980). Bounds on the scheduling of typed task systems. *SIAM Journal on Computing*, 9(3):541–551.

Jiang, X., Guan, N., Liang, H., Tang, Y., Qiao, L., and Wang, Y. (2021). Virtually-federated scheduling of parallel real-time tasks. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 482–494.

Jiang, X., Guan, N., Liu, D., and Liu, W. (2019a). Analyzing G-EDF scheduling for parallel real-time tasks with arbitrary deadlines. In *Proceedings of the 23rd Design, Automation and Test in Europe Conference and Exhibition*, pages 1537–1542.

Jiang, X., Guan, N., Long, X., Tang, Y., and He, Q. (2020). Real-time scheduling of parallel tasks with tight deadlines. *Journal of Systems Architecture, Article 101742*, 108.

Jiang, X., Guan, N., Long, X., and Yi, W. (2017). Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 80–91.

Jiang, X., Guan, N., Yang, M., Wang, Y., Tang, Y., and Yi, W. (2022a). Real-time scheduling of parallel task graphs with critical sections across different vertices. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4117–4133.

Jiang, X., Liang, H., Guan, N., Tang, Y., Qiao, L., and Wang, Y. (2022b). Scheduling parallel real-time tasks on virtual processors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):33–47.

Jiang, X., Long, X., Guan, N., and Wan, H. (2016). On the decomposition-based global EDF scheduling of parallel real-time tasks. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, pages 237–246.

Jiang, X., Long, X., Yang, T., and Deng, Q. (2018). On the soft real-time scheduling of parallel tasks on multiprocessors. In *In Proceedings of the 15th National Conference on Embedded System Technology*, pages 65–77.

Jiang, X., Sun, J., Tang, Y., and Guan, N. (2019b). Utilization-tensity bound for real-time DAG tasks under global EDF scheduling. *IEEE Transactions on Computers*, 69(1):39–50.

Júnior, J. A. S., Lima, G., Bletsas, K., and Kato, S. (2013). Multiprocessor real-time scheduling with a few migrating tasks. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 170–181.

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562.

Kang, W., Lee, K., Lee, J., Shin, I., and Chwa, H. S. (2021). Lalarand: Flexible layer-by-layer CPU/GPU scheduling for real-time DNN tasks. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 329–341.

Kato, S. and Ishikawa, Y. (2009). Gang EDF scheduling of parallel task systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 459–468.

Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., and Rajkumar, R. (2011a). RGEM: A responsive GPGPU execution model for runtime engines. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 57–66.

Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. (2011b). TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 22th USENIX Annual Technical Conference*, pages 17–30.

Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y., and Azumi, T. (2018). Autoware on board: Enabling autonomous vehicles with embedded systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 287–296.

Kohútka, L. (2022). A new FPGA-based architecture of task scheduler with support of periodic real-time tasks. In *Proceedings of the 29th International Conference on Mixed Design of Integrated Circuits and System (MIXDES)*, pages 77–82.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA. Curran Associates Inc.

Labetoulle, J. (1974). Some theorems on real-time scheduling.

Lakshmanan, K., Kato, S., and Rajkumar, R. (2010). Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 259–268.

Lee, H. and Al Faruque, M. A. (2016). Run-time scheduling framework for event-driven applications on a GPU-based embedded system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1956–1967.

Lee, H., Kim, H., Kim, C., Han, H., and Seo, E. (2020). Idempotence-based preemptive GPU kernel scheduling for embedded systems. *IEEE Transactions on Computers*, 70(3):332–346.

Lee, H., Roh, J., and Seo, E. (2018). A GPU kernel transactionization scheme for preemptive priority scheduling. In *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 202–213.

Lee, S., Guan, N., and Lee, J. (2022). Design and timing guarantee for non-preemptive Gang scheduling. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 132–144.

Lehoczky, J. P. (1990). Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 201–209.

Leontyev, H. and Anderson, J. H. (2009). A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems*, 43:60–92.

Leontyev, H. and Anderson, J. H. (2010). Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1):26–71.

Leontyev, H., Chakraborty, S., and Anderson, J. H. (2011). Multiprocessor extensions to real-time calculus. *Real-Time Systems*, 47:562–617.

Leung, J. Y.-T. and Whitehead, J. (1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250.

Li, J., Agrawal, K., Lu, C., and Gill, C. (2013). Analysis of global EDF for parallel tasks. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems,*, pages 3–13.

Li, J., Chen, J. J., Agrawal, K., Lu, C., Gill, C., and Saifullah, A. (2014). Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 85–96.

Li, J., Ferry, D., Ahuja, S., Agrawal, K., Gill, C., and Lu, C. (2017). Mixed-criticality federated scheduling for parallel real-time tasks. *Real-Time Systems*, 53(5):760–811.

Lin, C.-C., Shi, J., Ueter, N., Günzel, M., Reineke, J., and Chen, J.-J. (2022). Type-aware federated scheduling for typed DAG tasks on heterogeneous multicore platforms. *IEEE Transactions on Computers*, 72(5):1286–1300.

Liu, C. and Anderson, J. H. (2010). Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 3–13.

Liu, C. and Anderson, J. H. (2011). Supporting graph-based real-time applications in distributed systems. In *Proceedings of the IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 143–152.

Liu, C. and Anderson, J. H. (2012). An $O(m)$ analysis technique for supporting real-time self-suspending task systems. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium*, pages 373–382.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61.

Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., and Buttazzo, G. C. (2015). Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems*, pages 211–221.

Mok, A. K., Feng, X., and Chen, D. (2001). Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 75–84.

Mok, A. K.-L. (1983). *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology.

Nasri, M., Nelissen, G., and Brandenburg, B. B. (2019). Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 21–41.

Nelissen, G., Marcè i Igual, J., and Nasri, M. (2022). Response-time analysis for non-preemptive periodic moldable Gang tasks. In *Proceedings of the 34th Euromicro Conference on Real-Time Systems*.

Nemati, F., Behnam, M., and Nolte, T. (2011). Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 251–261.

Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., Liew, Y. T., Srivatsan, K., Moss, D., Subhaschandra, S., et al. (2017). Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–14.

Olmedo, I. S. n., Capodieci, N., Martinez, J. L., Marongiu, A., and Bertogna, M. (2020). Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–225.

Osborne, S. H., Bakita, J., Chen, J., Yandrofski, T., and Anderson, J. H. (2022). Minimizing DAG utilization by exploiting SMT. In *Proceedings of the 28th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 267–280.

Otterness, N. and Anderson, J. H. (2020). AMD GPUs as an alternative to NVIDIA for supporting real-time workloads. In *Proceedings of the 32nd Euromicro Conference on Real-Time Systems*.

Otterness, N. and Anderson, J. H. (2021). Exploring AMD GPU scheduling details by experimenting with "worst practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, pages 24–34.

Otterness, N., Miller, V., Yang, M., Anderson, J., Smith, F. D., and Wang, S. (2016). Gpu sharing for image processing in embedded real-time systems. In *Proceedings of the 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*.

Ousterhout, J. K. et al. (1982). Scheduling techniques for concurrent systems. In *Proceedings of the International Conference on Distributed Computing Systems*, volume 82, pages 22–30.

Park, J. J. K., Park, Y., and Mahlke, S. (2015). Chimera: Collaborative preemption for multitasking on a shared GPU. *ACM SIGARCH Computer Architecture News*, 43(1):593–606.

Parri, A., Biondi, A., and Marinoni, M. (2015). Response time analysis for G-EDF and G-DM scheduling of sporadic DAG-tasks with arbitrary deadline. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 205–214.

Pathan, R., Voudouris, P., and Stenström, P. (2017). Scheduling parallel real-time recurrent tasks on multicore platforms. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):915–928.

Peccerillo, B., Mannino, M., Mondelli, A., and Bartolini, S. (2022). A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture, Article 102561*.

Prisaznuk, P. J. (2008). ARINC 653 role in integrated modular avionics (IMA). In *Proceedings of the 27th IEEE/AIAA Digital Avionics Systems Conference*, pages 1–E.

Qamhieh, M., Fauberteau, F., George, L., and Midonnet, S. (2013). Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 287–296.

Qamhieh, M., George, L., and Midonnet, S. (2014). A stretching algorithm for parallel real-time DAG tasks on multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 13–22.

Qiu, Q., Liu, S., and Wu, Q. (2006). Task merging for dynamic power management of cyclic applications in real-time multiprocessor systems. In *Proceedings of the 24th International Conference on Computer Design*, pages 397–404.

Ramezani, R. (2021). Dynamic scheduling of task graphs in multi-FPGA systems using critical path. *The Journal of Supercomputing*, 77:597–618.

Ren, S., He, L., Li, J., Chen, C., Gu, Z., and Chen, Z. (2018). Scheduling DAG applications for time sharing systems. In *Proceedings of the 21st International Conference on Algorithms and Architectures for Parallel Processing*, pages 272–286.

Rong, P. and Pedram, M. (2008). Energy-aware task scheduling and dynamic voltage scaling in a real-time system. *Journal of Low Power Electronics*, 4(1):1–10.

SAE (2018). Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. *SAE Tech. Paper J3016 201806*.

Saha, S., Zhai, X., Ehsan, S., Majeed, S., and McDonald-Maier, K. (2021). RASA: Reliability-aware scheduling approach for FPGA-based resilient embedded systems in extreme environments. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 52(6):3885–3899.

Saidi, S. E., Pernet, N., and Sorel, Y. (2017). Automatic parallelization of multi-rate fmi-based co-simulation on multi-core. In *TMS/DEVS 2017-Symposium on Theory of Modeling and Simulation*.

Saifullah, A., Ferry, D., Lu, C., and Gill, C. (2012). Real-time scheduling of parallel tasks under a general DAG model. *IEEE Transactions on Parallel and Distributed Systems*.

Saifullah, A., Li, J., Agrawal, K., Lu, C., and Gill, C. (2013). Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435.

Saito, Y., Sato, F., Azumi, T., Kato, S., and Nishio, N. (2018). ROSCH: real-time scheduling framework for ROS. In *Proceedings of the 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 52–58.

Serlin, O. (1972). Scheduling of time critical processes. In *Proceedings of Spring Joint Computer Conference*, pages 925–932.

Serrano, M. A., Melani, A., Bertogna, M., and Quiñones, E. (2016). Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In *Proceedings of the 20th Design, Automation & Test in Europe Conference & Exhibition*, pages 1066–1071.

Serrano, M. A. and Quinones, E. (2018). Response-time analysis of DAG tasks supporting heterogeneous computing. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6.

Seshadri, K., Akin, B., Laudon, J., Narayanaswami, R., and Yazdanbakhsh, A. (2022). An evaluation of edge TPU accelerators for convolutional neural networks. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 79–91.

Shin, I., Easwaran, A., and Lee, I. (2008). Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 181–190.

Soliman, M. R. and Pellizzoni, R. (2019). PREM-based optimal task segmentation under fixed priority scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 4:1–4:23.

Springer, T. and Zhao, P. (2021). Hierarchical scheduling for real-time periodic tasks in symmetric multiprocessing. In *Computer Science & Information Technology Proceedings*, volume 11.

Srinivasan, A. and Anderson, J. H. (2006). Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 6(72):1094–1117.

Sun, J., Guan, N., Guo, Z., Xue, Y., He, J., and Tan, G. (2021). Calculating worst-case response time bounds for OpenMP programs with loop structures. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 123–135.

Sun, J., Guan, N., Sun, J., and Chi, Y. (2019). Calculating response-time bounds for OpenMP task systems with conditional branches. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 169–181.

Sun, J., Li, F., Guan, N., Zhu, W., Xiang, M., Guo, Z., and Yi, W. (2020). On computing exact WCRT for DAG tasks. In *Proceedings of the 57th ACM/IEEE Design Automation Conference*, pages 1–6.

Tămaş-Selicean, D. and Pop, P. (2015). Design optimization of mixed-criticality real-time embedded systems. *ACM Transactions on Embedded Computing Systems*, 14(3):1–29.

Tang, S., Voronov, S., and Anderson, J. H. (2019). GEDF tardiness: Open problems involving uniform multiprocessors and affinity masks resolved. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, pages 13:1–13:21.

Tang, S., Voronov, S., and Anderson, J. H. (2021). Extending EDF for soft real-time scheduling on unrelated multiprocessors. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 253–265.

The Khronos Group (2023). OpenVX: Portable, power efficient vision processing. Online, https://www.khronos.org/openvx/.

Tong, G. and Liu, C. (2015). Supporting soft real-time sporadic task systems on uniform heterogeneous multiprocessors with no utilization loss. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2740–2752.

Ueter, N., Von Der Brüggen, G., Chen, J.-J., Li, J., and Agrawal, K. (2018). Reservation-based federated scheduling for parallel real-time tasks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 482–494.

Ullman, J. D. (1975). NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393.

Verner, U., Mendelson, A., and Schuster, A. (2014). Scheduling periodic real-time communication in multi-GPU systems. In *Proceedings of the 23rd International Conference on Computer Communication and Networks*, pages 1–8.

Verucchi, M., Theile, M., Caccamo, M., and Bertogna, M. (2020). Latency-aware generation of single-rate DAGs from multi-rate task sets. In *Proceedings of the 32nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 226–238.

Voronov, S. and Anderson, J. H. (2018). An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 408–420.

Voronov, S., Anderson, J. H., and Yang, K. (2018). Tardiness bounds for fixed-priority global scheduling without intra-task precedence constraints. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 8–18.

Voronov, S., Anderson, J. H., and Yang, K. (2021a). Tardiness bounds for fixed-priority global scheduling without intra-task precedence constraints. *Real-Time Systems*, 57(1):4–54.

Voronov, S., Tang, S., Amert, T., and Anderson, J. H. (2021b). AI meets real-time: Addressing real-world complexities in graph response-time analysis. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 82–96.

Wang, C. and Luo, Z. (2022). A review of the optimal design of neural networks based on FPGA. *Applied Sciences*, (21).

Wang, K., Jiang, X., Guan, N., Liu, D., Liu, W., and Deng, Q. (2019). Real-time scheduling of DAG tasks with arbitrary deadlines. *ACM Transactions on Design Automation of Electronic Systems*, 24(6):1–22.

Ward, B. C., Erickson, J. P., and Anderson, J. H. (2013). A linear model for setting priority points in soft real-time systems. In *Proceedings of Real-Time Systems: The Past, the Present, and the Future — A Conference Organized in Celebration of Alan Burns' Sixtieth Birthday*, pages 192–205.

Wu, B., Liu, X., Zhou, X., and Jiang, C. (2017). FLEP: Enabling flexible and efficient preemption on GPUs. *ACM SIGPLAN Notices*, 52(4):483–496.

Wu, Y., Gao, Z., and Dai, G. (2014). Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations. *Journal of Systems Architecture*, 60(3):247–257.

Xia, Y., Prasanna, V. K., and Li, J. H. (2010). Hierarchical scheduling of DAG structured computations on manycore processors with dynamic thread grouping. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 154–174.

Xiang, Y. and Kim, H. (2019). Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *Proceedings of the 40th IEEE Real-Time Systems Symposium*, pages 392–405.

Xie, G., Xiao, X., Li, R., and Li, K. (2017). Schedule length minimization of parallel applications with energy consumption constraints using heuristics on heterogeneous distributed systems. *Concurrency and Computation: Practice and Experience*, 29(16).

Xu, J., Li, K., and Chen, Y. (2022). Real-time task scheduling for FPGA-based multicore systems with communication delay. *Microprocessors and Microsystems*, 90:104468.

Xu, Y., Wang, R., Li, T., Song, M., Gao, L., Luan, Z., and Qian, D. (2016). Scheduling tasks with mixed timing constraints in GPU-powered real-time systems. In *Proceedings of the 30th International Conference on Supercomputing*, pages 1–13.

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

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

Yang, K., Elliott, G. A., and Anderson, J. H. (2015). Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the 23th International Conference on Real-Time Networks and Systems*, pages 77–86.

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

Yang, M. (2018). Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*.

Yang, M., Amert, T., Yang, K., Otterness, N., Anderson, J. H., Smith, F. D., and Wang, S. (2018). Making OpenVX really "real time". In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 80–93.

Yang, M., Lei, H., Liao, Y., and Rabbe, F. (2013). PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi-GPU sharing under partitioned scheduling. In *Proceedings of the 11th IEEE International Conference on Dependable, Autonomic and Secure Computing*, pages 207–214.

Yang, T., Deng, Q., and Sun, L. (2019). Building real-time parallel task systems on multi-cores: A hierarchical scheduling approach. *Journal of Systems Architecture*, 92:1–11.

Yao, Y., Liu, S., Wu, S., Wang, J., Ni, J., Yang, G., and Zhang, Y. (2021). WAMP^{2}S: Workload-aware performance model based pseudo-preemptive real-time scheduling for the airborne embedded system. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2767–2780.

Yu, J., Wang, Z., Vasudevan, V., Yeung, L., Seyedhosseini, M., and Wu, Y. (2022). CoCa: Contrastive captioners are image-text foundation models. *Transactions on Machine Learning Research*, 2022.

Zhao, S., Dai, X., and Bate, I. (2022). DAG scheduling and analysis on multi-core systems by modelling parallelism and dependency. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4019–4038.

Zhao, S., Dai, X., Bate, I., Burns, A., and Chang, W. (2020). DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*.

Zhou, H., Tong, G., and Liu, C. (2015). GPES: A preemptive execution system for GPGPU computing. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97.

Zhou, N., Qi, D., Wang, X., and Zheng, Z. (2017). A static task scheduling algorithm for heterogeneous systems based on merging tasks and critical tasks. *Journal of Computational Methods in Sciences and Engineering*, 17(4):715–732.

Zhu, Z., Zhang, J., Zhao, J., Cao, J., Zhao, D., Jia, G., and Meng, Q. (2019). A hardware and software task-scheduling framework based on CPU+FPGA heterogeneous architecture in edge computing. *IEEE Access*, 7:148975–148988.

Zou, A., Li, J., Gill, C. D., and Zhang, X. (2023). RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1450–1465.