Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Combination of Time-Triggered and Event-Triggered Scheduling with Dedicated Resources and Precedences

Bc. Lukáš Halaška

Supervisor: Mgr. Marek Vlk, Ph.D.
Supervisor–specialist: prof. Dr. Ing. Zdeněk Hanzálek
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Halaška Lukáš**	Personal ID number:	**474730**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

Combination of Time-Triggered and Event-Triggered Scheduling with Dedicated Resources and Precedences

Master’s thesis title in Czech:

Kombinace time-triggered a event-triggered rozvrhování s dedikovanými zdroji a s precedencemi

Guidelines:

Cyber-physical systems and communication networks usually incorporate both time-triggered (TT) and event-triggered (ET) manners of communication. While the TT paradigm is necessary for messages of the highest criticality and allows more accessible system certification, ET scheduling brings better system properties. The aim of this work is to propose an algorithm that combines both TT and ET scheduling so as to keep the advantages of both approaches. The student will investigate existing work on unary-resource scheduling of ET messages [1] and for the combination of ET and TT messages [2] and will propose an algorithm for scheduling dedicated tasks on multiple resources and with precedences. The idea is to extend the concept of the schedule abstraction graph [1,3] or the fixation graph [2]. The work will experimentally evaluate the developed algorithms on at least 1000 randomly generated instances, measuring the speed, the schedulability ratio, and pessimism (ratio of false negative results).

Bibliography / sources:

[1] Nasri, M, and Brandenburg, B.B. "An exact and sustainable analysis of non-preemptive scheduling." 2017 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2017.
[2] Jaros, M. "Combination of Time-Triggered and Event-Triggered Scheduling." 2022 Diploma Thesis.
[3] Nasri, M., Nelissen, G., & Brandenburg, B.B. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. Euromicro Conference on Real-Time Systems, 2018.

Name and workplace of master’s thesis supervisor:

Mgr. Marek Vlk, Ph.D. Optimization CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **31.01.2023** Deadline for master's thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Mgr. Marek Vlk, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

First, I would like to thank my supervisor Mgr. Marek Vlk, Ph.D. for his guidance, advice, and a great amount of support throughout the entire master's project.

I would also like to express my gratitude to prof. Dr. Ing. Zdeněk Hanzálek for providing me with the possibility to work on such an interesting topic and for his insightful comments.

Finally, I would like to thank my family, my friends, and my girlfriend for their love and mental support throughout the entire master's studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 25. May 2023

Abstract

The event-triggered (ET) systems provide flexible and responsive behavior by dynamically scheduling tasks based on events in the system, while the time-triggered (TT) systems provide robustness and traceability by scheduling tasks at design time. This thesis aims to combine the advantages of both real-time system paradigms by integrating ET schedulability analysis and TT schedule synthesis. We propose algorithms for the combination of ET schedulability analysis and TT schedule synthesis, considering non-preemptive tasks with precedence constraints and a dedicated multiprocessor platform. The proposed algorithms are based on and extend the Schedule Abstraction Graph generation algorithm, proposed by Nasri and Brandenburg [RTSS 2017, pp. 12–23]. For the combination of TT and ET tasks, we propose two scalable algorithms. The experimental results demonstrate the amount of pessimism, the schedulability ratio, and the scalability of the proposed approaches.

Keywords: online and offline scheduling, non-preemptive tasks, dedicated resources, precedence constraints, schedulability analysis, schedule abstraction graph

Supervisor: Mgr. Marek Vlk, Ph.D.
České vysoké učení technické v Praze,
Český institut informatiky, robotiky a kybernetiky,
IID: Průmyslová informatika,
OPT: Optimalizace,
Jugoslávských partyzánů 1580,
160 00 Praha 6

Abstrakt

Event-triggered (ET) systémy poskytují flexibilní a responsivní chování tím, že dynamicky rozvrhují úlohy na základě událostí v systému, zatímco time-triggered (TT) systémy poskytují robustnost a sledovatelnost rozvrhováním úloh v době návrhu. Cílem této práce je spojit výhody obou paradigmat systémů reálného času prostřednictvím integrace ET analýzy rozvrhovatelnosti a TT syntézy rozvrhu. Tato práce navrhuje algoritmy pro kombinaci ET analýzy rozvrhovatelnosti a TT syntézy rozvrhu pro nepreemptivní úlohy s relacemi následnosti a dedikované zdroje. Navržené algoritmy jsou založeny na algoritmu, který zároveň rozšiřují, pro generování rozvrhovacího grafu (Schedule Abstraction Graph), navrženém autory Nasri a Brandenburg [RTSS 2017, s. 12–23]. Pro kombinaci TT a ET úloh jsou navrženy dva škálovatelné algoritmy. Experimentální výsledky demonstrují míru pesimismu, rozvrhovatelnost a škálovatelnost navrhovaných přístupů.

Klíčová slova: online a offline rozvrhování, nepreemptivní úlohy, dedikované zdroje, relace následnosti, analýza rozvrhovatelnosti, rozvrhovací graf

Překlad názvu: Kombinace time-triggered a event-triggered rozvrhování s dedikovanými zdroji a s precedencemi

Contents

1 Introduction 1
1.1 Related work 1
1.2 This thesis 2

2 Formal description of the problem 3
2.1 Multiprocessor platform 3
2.2 Event-triggered tasks and jobs 3
2.3 Scheduling of ET jobs and scheduling policy 5
2.4 Execution scenario and schedulability 6
2.5 Time-triggered tasks and jobs 6
2.6 Scheduling of TT jobs 8
2.7 Combining ET and TT tasks 8

3 ET solutions 9
3.1 Brute force approach to schedulability analysis 9
3.2 Worst-case approach to schedulability analysis 12
3.3 Schedule Abstraction Graph approach to schedulability analysis 12
3.3.1 Schedule Abstraction Graph 13
3.3.2 SAG generation 13
3.3.3 Expansion phase 14
3.3.4 Merge phase 14
3.3.5 Parameters of vertices and edges 15
3.3.6 Job eligibility 15
3.3.7 Pseudocode 16
3.3.8 Discussion and example 21

4 ET solution evaluation 27
4.1 Model of the multiprocessor platform 27
4.2 Instance generation 28
4.2.1 Generated datasets 29
4.3 Evaluation of pessimism in the SAG approach and evaluation of the worst-case approach impact 30
4.4 Evaluation of schedulability and runtime of the SAG approach 31

5 ET+TT solutions 35
5.1 Brute force algorithm 35
5.2 Fixation Graph Generation algorithm 38
5.2.1 Fixation graph structure 38
5.2.2 Description of the FGG 39
5.2.3 Fixation phase 39
5.2.4 Pseudocode 40
5.2.5 Discussion 44

6 ET+TT solution evaluation 45
6.1 Instance generation 45
6.1.1 Generated datasets 46
6.2 Evaluation of pessimism in the FGG algorithm and the FGG-EB algorithm 46
6.3 Evaluation of schedulability and runtime of the FGG algorithm and the FGG-EB algorithm 47

7 Conclusion 53

Bibliography 55

Figures

3.1 The example of the generated SAG. The inside of each vertex v is divided into two columns. The left column contains all applicable jobs in the vertex and their respective earliest and latest release time in the vertex, which is denoted as $E_{i,j,k} : \left(v.r^{min}_{i,j,k}, v.r^{max}_{i,j,k}\right)$. The right column contains the finish time intervals of all processors in the vertex. The finish time interval of a processor π is denoted as $\pi : [v.EFT_\pi, v.LFT_\pi]$. Each level of the SAG is labeled with a set of vertices that constitute the corresponding level. Each edge is labeled with a scheduled job. 25

4.1 Example of the network model. The edges correspond to the processors. Therefore, they are labeled with a unique index $\pi \in \{1, \dots, 8\}$. 28
4.2 Runtimes of the SAG approach to schedulability analysis for datasets $\mathcal{D}^a_1, \dots, \mathcal{D}^a_9$ relative to the number of ET tasks. 32
4.3 Runtimes of the SAG approach to schedulability analysis for datasets $\mathcal{D}^a_1, \dots, \mathcal{D}^a_9$ relative to the number of ET jobs. 33

6.1 Runtimes of the FGG algorithm for datasets $\mathcal{D}^b_1, \dots, \mathcal{D}^b_9$ relative to the number of TT tasks. 48
6.2 Runtimes of the FGG-EB algorithm for datasets $\mathcal{D}^b_1, \dots, \mathcal{D}^b_9$ relative to the number of TT tasks. 49
6.3 Runtimes of the FGG algorithm for datasets $\mathcal{D}^b_1, \dots, \mathcal{D}^b_9$ relative to the number of TT jobs. 50
6.4 Runtimes of the FGG-EB algorithm for datasets $\mathcal{D}^b_1, \dots, \mathcal{D}^b_9$ relative to the number of TT jobs. 51

Tables

3.1 Example of ET task parameters for calculation of $N^{\mathcal{E}}$, $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4\}$. 11
3.2 Parameters of ET tasks used in the SAG generation example, $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2\}$. 22
3.3 Parameters of ET jobs used in the SAG generation example, $\mathcal{E} = \{\{\{E_{1,1,1}, E_{1,1,2}\}\}, \{\{E_{2,1,1}, E_{2,1,2}\}\}\}$. 22

4.1 Values of parameters A^r, A^d, A^j, and A^c used for generation of datasets $\mathcal{D}^a_1, \dots, \mathcal{D}^a_9$. 30
4.2 Results of the schedulability analysis of $\mathcal{D}^a_0$ using the brute force (BF) approach, the SAG approach, and the worst-case (WC) approach. 31
4.3 Schedulability of instances from datasets $\mathcal{D}^a_1, \dots, \mathcal{D}^a_9$. Each cell corresponding to a dataset contains the total number of schedulable instances, the total number of unschedulable instances, and the total number of instances that reached the time limit in this order. 32

5.1 Example of TT task parameters for calculation of $N^{\mathcal{T}}$, $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4\}$. 37

6.1 Values of parameters A^r, A^d, A^j, and A^c used for generation of datasets $\mathcal{D}^b_1, \dots, \mathcal{D}^b_9$. 46
6.2 Resulting schedulability of the instances from $\mathcal{D}^b_0$ for the brute force (BF) algorithm, the FGG algorithm, and the FGG-EB algorithm. 47

6.3 Schedulability of instances from datasets $\mathcal{D}_1^b, \ldots, \mathcal{D}_9^b$. Each cell corresponding to a dataset contains two rows. The first row contains the results of the FGG algorithm and the second row contains the results of the FGG-EB algorithm. Each row in a cell corresponding to a dataset consists of the total number of schedulable instances, the total number of unschedulable instances, the total number of instances that reached the time limit, and the total number of instances that reached the memory limit in this order. 49

Chapter 1
Introduction

Event-triggered (ET) systems and time-triggered (TT) systems are the two main paradigms of real-time systems. In the ET systems, tasks, which are released as a consequence of an occurrence of an event in the system, are scheduled online (dynamically) based on a scheduling policy in the online scheduler. This allows for more flexible and responsive system behavior as tasks are dispatched as needed rather than being tied to fixed time slots. However, the ET systems lack traceability since the runtime executions may vary. To impose guarantees on the completion times of the tasks, the schedulability analysis of the ET system, which may be computationally expensive, is conducted in advance before the system is run in real-time. In contrast, in the TT systems, tasks are executed at predetermined start times. Therefore, the tasks in the TT system are scheduled offline (at design time). This allows for the robustness and dependability of the TT systems. Moreover, the deterministic nature of the TT systems provides better traceability than in the case of the ET systems. Nevertheless, the TT systems lack flexibility since the fixed schedule for the tasks is computed in advance, which is computationally expensive. Consequently, many real-time systems combine both ET and TT paradigms to keep their advantages.

The main aim of this thesis is to provide the integration of both ET schedulability analysis and TT schedule synthesis on multiple dedicated resources to keep the advantages of both.

1.1 Related work

The combination of ET and TT scheduling has been addressed in [1, 2]. Other works focus on the usage of the combination of ET and TT scheduling in specific domains such as ethernet-based networks [3, 4], time-sensitive networks (TSNs) [5, 6, 7, 8], distributed automotive networks [9, 10, 11], or the FTT-CAN protocol [12, 13].

In many current works, no prior information about the release times and the execution times of the ET tasks is considered. Moreover, many of these works address only preemptive scheduling. Non-preemptive scheduling, i.e., scheduling of tasks that must finish their execution without any interruption once their execution has started, usually poses a more challenging problem to tackle. A solution for the combination of non-preemptive ET and TT tasks on a single unary resource was proposed in [14, 15], where prior information about the release times and the execution times of the ET tasks is assumed. Moreover, [14, 15] improves and corrects the scalable schedulability analysis of non-preemptive ET tasks proposed in [16]. The schedulability analysis from [16] introduced a novel framework of Schedule Abstraction Graphs (SAGs). Further scalability improvements of the schedulability analysis from [16] were proposed in [17]. Other works [18, 19] propose a scalable schedulability analysis for multiple resources (multiprocessor platforms) using the SAG framework. In [19], tasks are considered to be restricted by precedence constraints, which are modeled with Directed Acyclic Graphs (DAGs). Nevertheless, both of these works consider parallel resources, however, the dedication of tasks to processors is not considered.

1.2 This thesis

In this thesis, we tackle a scheduling problem with two types of tasks, namely ET tasks and TT tasks. For the ET tasks, we assume fixed priority, release jitter, and execution time variation. Moreover, we assume a lower bound and an upper bound for both the release times and the execution times of the ET tasks. This information is assumed to be known a priori. For the TT tasks, we assume fixed release and execution times.

All tasks are periodic and non-preemptive. Furthermore, all tasks have deadlines. Moreover, both types of tasks are restricted by precedence constraints. In this work, we consider only one specific type of precedence constraint, namely the precedence chains. All tasks are executed on dedicated resources that form a multiprocessor platform. Each processor can be occupied by the execution of at most one task at any time. The goal is either to find start times for the TT tasks such that they meet their deadline and the ET tasks are guaranteed to meet their deadlines as well in any execution scenario or to decide that no such schedule for the TT tasks exists. The start times for the TT tasks are computed offline. The ET tasks are analyzed offline as well and then scheduled online during the runtime. The schedulability analysis simulates the work of an online scheduler. The online scheduler is assumed to schedule the ET tasks based on a work-conserving policy, i.e., a scheduling policy that does not allow any idle time when there is an ET task that can be executed. To summarize, we do a schedule synthesis for the TT tasks and a schedulability analysis for the ET tasks such that no collisions and no deadline misses occur during the online scheduling.

The contribution of this thesis is a formal description of the scheduling problem that combines the scheduling of ET and TT tasks restricted by precedence constraints on dedicated resources, algorithms for the ET schedulability analysis, algorithms for the TT schedule synthesis ensuring schedulability of all tasks, and the empirical evaluation of all proposed algorithms. The source codes of the implemented algorithms and the instances used for the empirical evaluation are publicly available on GitHub[1].

The rest of this thesis is structured as follows. Chapter 2 provides a formal description of the problem. Chapter 3 focuses on the ET schedulability analysis. The proposed approaches to schedulability analysis are evaluated in Chapter 4. Analogously, Chapter 5 focuses on our solutions for the combination of ET and TT tasks. The proposed algorithms for the combination of ET and TT tasks are evaluated in Chapter 6. The conclusion of this thesis is situated in Chapter 7.

[1] `https://github.com/halasluk/ETTT_dedicated_multicore`

Chapter 2
Formal description of the problem

This chapter provides a formal description of the ET tasks and their scheduling. Moreover, the TT tasks, their scheduling, and their combination with ET tasks are described. All parameters described in this chapter can have only integer values. All proposed algorithms in later chapters work only with integer values. Moreover, most of the notation and terminology in this chapter is consistent with [14, 15, 16].

2.1 Multiprocessor platform

The multiprocessor platform is considered to consist of m processors. Each processor in our multiprocessor platform is assigned a unique index $\pi \in \{1, \dots, m\}$. For simplicity, processors are called by their indexes.

2.2 Event-triggered tasks and jobs

The set of ET tasks is defined as $\mathcal{E} = \{\mathcal{E}_1, \dots, \mathcal{E}_n\}$, where n is the total number of ET tasks. Each ET task $\mathcal{E}_i \in \mathcal{E}$ is defined by its period τ_i^{ET}, deadline d_i^{ET}, earliest release time r_i^{min}, latest release time r_i^{max}, shortest execution time c_i^{min}, longest execution time c_i^{max}, priority p_i, and sequence of processors $\sigma_i = (\sigma_{i,1}, \dots, \sigma_{i,l_i})$, where l_i is the length of this sequence, which corresponds to a chain of processors that $\mathcal{E}_i$ is assigned to. The values of ET task parameters are assumed to satisfy the following constraints:

$$r_i^{max} + l_i \cdot c_i^{max} \leq d_i^{ET} \leq \tau_i^{ET},$$

$$0 \leq r_i^{min} \leq r_i^{max},$$

$$1 \leq c_i^{min} \leq c_i^{max},$$

$$p_i \in \mathbb{N}_0.$$

Each ET task $\mathcal{E}_i$ consists of its periodic occurrences (task's occurrences). Therefore, we define each ET task as $\mathcal{E}_i = \{\mathcal{E}_{i,1}, \dots, \mathcal{E}_{i,h_i}\}$, where h_i is the total number of occurrences of task $\mathcal{E}_i$. We compute h_i for each ET task from set $\mathcal{E}$ as

$$h_i = \frac{\eta}{\tau_i^{ET}},$$

where η is a hyperperiod, which is computed as

$$\eta = \mathrm{LCM}\left(\tau_1^{ET}, \ldots, \tau_n^{ET}\right),$$

where the function LCM computes the least common multiple of its arguments. The hyperperiod η gives us the time window in which we are conducting the schedulability analysis. Thanks to the assumption that $d_i^{ET} \leq \tau_i^{ET}$, we do not need to analyze a larger time window than η.

Furthermore, we define each occurrence of each ET task as a set of ET jobs, which are dedicated to processors given by σ_i. Therefore, we define $\mathcal{E}_{i,j} = \{E_{i,j,1}, \ldots, E_{i,j,l_i}\}$, where job $E_{i,j,k} \in \mathcal{E}_{i,j}$ is dedicated to be executed on processor $\sigma_{i,k}$, which corresponds to k-th element of the sequence of processors σ_i. Therefore, we can see that the dedication of an ET job to a processor does not depend on the occurrence of the corresponding ET task. In other words, $E_{i,j,k}$ is dedicated to processor $\sigma_{i,k}$, $\forall j \in \{1, \ldots, h_i\}$. Each ET job $E_{i,j,k}$ also has deadline $d_{i,j,k}^{ET}$, shortest execution time $c_{i,j,k}^{min}$, longest execution time $c_{i,j,k}^{max}$, and priority $p_{i,j,k}$. For $k = 1$, each ET job $E_{i,j,1}$ is also defined by its earliest release time $r_{i,j,1}^{min}$ and latest release time $r_{i,j,1}^{max}$. The values of ET job parameters are assumed to satisfy the following constraints. $\forall i \in \{1, \ldots, n\}$, $\forall j \in \{1, \ldots, h_i\}$, $\forall k \in \{1, \ldots, l_i\}$,

$$d_{i,j,k}^{ET} = d_i^{ET} + (j-1) \cdot \tau_i^{ET} - (l_i - k) \cdot c_i^{max},$$

$$c_{i,j,k}^{min} = c_i^{min},$$

$$c_{i,j,k}^{max} = c_i^{max},$$

$$p_{i,j,k} = p_i,$$

and, $\forall i \in \{1, \ldots, n\}$, $\forall j \in \{1, \ldots, h_i\}$,

$$r_{i,j,1}^{min} = r_i^{min} + (j-1) \cdot \tau_i^{ET},$$

$$r_{i,j,1}^{max} = r_i^{max} + (j-1) \cdot \tau_i^{ET}.$$

The constraints show that each ET job inherits execution times and priority from the corresponding ET task. Release times of ET jobs $E_{i,j,1}$ are shifted according to the occurrence of the corresponding ET task as well as the deadline of each ET job $E_{i,j,k}$ which is in addition tightened by $(l_i - k) \cdot c_i^{max}$. This additional shift of deadline will be discussed shortly. To sum up, we can express $\mathcal{E}$ as a set of ET jobs in the following way:

$$\mathcal{E} = \left\{\left\{\{E_{1,1,1}, \ldots, E_{1,1,l_1}\}, \ldots, \{\ldots, E_{1,h_1,l_1}\}\right\}, \ldots, \left\{\ldots, \{\ldots, E_{n,h_n,l_n}\}\right\}\right\}.$$

Another important property of such defined ET jobs is that some ET jobs precede other ones. When

$$E_{i,j,k_x} \prec E_{i,j,k_y},$$

we say that E_{i,j,k_x} precedes E_{i,j,k_y}. This means that E_{i,j,k_x} needs to finish its execution before E_{i,j,k_y} is released. An ET job that has some predecessors is released when all its predecessors finish their execution. The precedence constraint is considered to be present only among ET jobs of the same occurrence of a given ET task. Thanks to our definition of ET jobs, it always holds that

$$k_x < k_y \Leftrightarrow E_{i,j,k_x} \prec E_{i,j,k_y}, \quad \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, h_i\}.$$

Moreover, we can observe that ET jobs of the same occurrence of a given ET task form a precedence chain, which is a special case of precedence constraint. Therefore, we consider only precedence chains consisting of ET jobs as precedence constraints restricting ET jobs in this work. The precedence chain can be expressed as

$$E_{i,j,1} \prec \cdots \prec E_{i,j,l_i}, \quad \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, h_i\}.$$

The fact that ET jobs of the same occurrence of a given ET task form a precedence chain causes the position of an ET job $E_{i,j,k}$ in the precedence chain to affect its deadline $d^{ET}_{i,j,k}$. That is why $d^{ET}_{i,j,k}$ is tightened by $(l_i - k) \cdot c^{max}_i$. Note that meeting deadlines $d^{ET}_{i,j,k}$, $\forall k \in \{1, \ldots, l_i\}$, ensures that the deadline of the j-th occurrence of a given ET task $\mathcal{E}_i$ is always met.

2.3 Scheduling of ET jobs and scheduling policy

Let us now define a few important terms regarding the scheduling of ET jobs. A priori unknown release time $r^{ET}_{i,j,k} \in \left[r^{min}_{i,j,k}, r^{max}_{i,j,k}\right]$ of each ET job $E_{i,j,k}$ is revealed during runtime of the ET system. The fact that $r^{ET}_{i,j,k}$ is from a closed interval is referred to as release jitter. Once the value of $r^{ET}_{i,j,k}$ is known, we say that $E_{i,j,k}$ is released at time $r^{ET}_{i,j,k}$. ET job $E_{i,j,k}$ can start its execution at time t_e only if it is released and if the processor dedicated to this job is not occupied by executing another ET job. If $E_{i,j,k}$ starts its execution at time t_e, then it occupies its dedicated processor during interval $\left[t_e, t_e + c^{ET}_{i,j,k}\right)$, where $c^{ET}_{i,j,k} \in \left[c^{min}_{i,j,k}, c^{max}_{i,j,k}\right]$ is a priori unknown execution time. The fact that $c^{ET}_{i,j,k}$ is from a closed interval is referred to as execution time variation. ET job $E_{i,j,k}$ is finished when it finishes its execution, otherwise it is termed unfinished. If $E_{i,j,k}$ finishes its execution at time $t_e + c^{ET}_{i,j,k} > d^{ET}_{i,j,k}$, then a deadline miss occurs.

Another important aspect of scheduling ET jobs that we have to consider is the scheduling policy. First, we define E^R to be a set of released ET jobs. Furthermore, E_π is defined as a set of all ET jobs dedicated to processor π, i.e., a set of jobs $E_{i,j,k}$ for which $\sigma_{i,k} = \pi$, $\forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, h_i\}, \forall k \in \{1, \ldots, l_i\}$. In this work, scheduling policy is defined as a function $\mathcal{P}\left(t, \pi, E^R \cap E_\pi\right)$, which takes a given time t, a given processor π, and the intersection of given sets E^R and E_π as its inputs and returns a job $E_{i,j,k}$ that should be scheduled next. Such job $E_{i,j,k}$ must be released by the time t and the processor π must not be occupied by executing another job. The scheduling policy is invoked whenever a job finishes its execution or when a new job is released. We can observe from the definition of the scheduling policy that it is invoked for each processor independently. Therefore, the scheduling policy can be invoked more than once at time t if multiple jobs dedicated to different processors are released or finish their execution at the same time. It is possible for the scheduling policy not to return any job. This happens when $E^R \cap E_\pi = \emptyset$ or when a newly released job at time t is dedicated to a processor which is occupied by executing another job at time t.

In this work, there is only one type of scheduling policy considered and utilized. It is the fixed priority – earliest deadline first (FP-EDF) policy. As mentioned earlier, the FP-EDF policy is called for each processor separately, i.e., the FP-EDF policy always picks a job from $E^R \cap E_\pi$. The FP-EDF policy picks a job that should be scheduled next based on the following decision rules. It chooses a job with the highest priority, i.e., a job that has the lowest value of p. If multiple jobs have the same highest priority, then it picks a job with the lowest deadline d^{ET}. If there are multiple jobs with the same highest priority and the same lowest deadline, then it picks a job $E_{i,j,k}$ with the lowest index i, which is an additional rule that ensures deterministic behavior of the FP-EDF policy.

2.4 Execution scenario and schedulability

For a set of ET jobs $\mathcal{E}$, we define an execution scenario as $\gamma = \{R, C\}$. Here, R corresponds to a set of release times $r^{ET}_{i,j,1}$ of ET jobs $E_{i,j,1}$, $\forall i \in \{1,\dots,n\}, \forall j \in \{1,\dots,h_i\}$, and C corresponds to a set of execution times $c^{ET}_{i,j,k}$ of ET jobs $E_{i,j,k}$, $\forall i \in \{1,\dots,n\}, \forall j \in \{1,\dots,h_i\}, \forall k \in \{1,\dots,l_i\}$. The set of release times is defined as $R = \{R_1,\dots,R_n\}$, where $R_i = \left\{r^{ET}_{i,1,1},\dots,r^{ET}_{i,h_i,1}\right\}$, where $r^{ET}_{i,j,1} \in \left[r^{min}_{i,j,1}, r^{max}_{i,j,1}\right]$. The set of execution times is defined as $C = \{C_1,\dots,C_n\}$, where $C_i = \{C_{i,1},\dots,C_{i,h_i}\}$, where $C_{i,j} = \left\{c^{ET}_{i,j,1},\dots,c^{ET}_{i,j,l_i}\right\}$, where $c^{ET}_{i,j,k} \in \left[c^{min}_{i,j,k}, c^{max}_{i,j,k}\right]$. In other words, an execution scenario γ specifies one possible combination of release times and execution times of ET jobs in $\mathcal{E}$. It should be noted that $r^{ET}_{i,j,k}$ of $E_{i,j,k}$, $\forall i \in \{1,\dots,n\}, \forall j \in \{1,\dots,h_i\}, \forall k \in \{2,\dots,l_i\}$ cannot be part of such definition of execution scenario as they depend on the time when their predecessors in the precedence chain finish their execution, which depends on the scheduling policy.

We say that a set of ET tasks $\mathcal{E}$ is schedulable under a policy $\mathcal{P}$ if there exists no execution scenario γ which results in a deadline miss using $\mathcal{P}$. In other words, each ET job in $\mathcal{E}$ must finish its execution before its respective deadline in each possible execution scenario for $\mathcal{E}$ to be schedulable under policy $\mathcal{P}$.

Since one of the aims of this work is to develop a scalable schedulability analysis of a set of ET tasks, we will rigorously define three different types of schedulability analysis. The definitions of these types of schedulability analysis are taken from [20].

A schedulability analysis is termed sufficient if all instances of a given scheduling problem that are deemed schedulable according to the analysis are indeed schedulable. A schedulability analysis is termed necessary if all instances of a given scheduling problem that are deemed unschedulable according to the analysis are indeed unschedulable. A schedulability analysis that is both sufficient and necessary is referred to as exact.

2.5 Time-triggered tasks and jobs

The set of TT tasks is defined as $\mathcal{T} = \{\mathcal{T}_1,\dots,\mathcal{T}_{n'}\}$, where n' is the total number of TT tasks. Each TT task $\mathcal{T}_i \in \mathcal{T}$ is defined by its period τ^{TT}_i, deadline d^{TT}_i, release time r^{TT}_i, execution time c^{TT}_i, and sequence of processors $\sigma'_i = \left(\sigma'_{i,1},\dots,\sigma'_{i,l'_i}\right)$, where l'_i is the length of this sequence, which corresponds to a chain of processors that $\mathcal{T}_i$ is assigned to. The values of TT task parameters are assumed to satisfy the following constraints:

$$r^{TT}_i + l'_i \cdot c^{TT}_i \le d^{TT}_i \le \tau^{TT}_i,$$

$$0 \le r^{TT}_i,$$

$$1 \le c^{TT}_i.$$

Each TT task $\mathcal{T}_i$ consists of its periodic occurrences (task's occurrences). Therefore, we define each TT task as $\mathcal{T}_i = \left\{\mathcal{T}_{i,1}, \ldots, \mathcal{T}_{i,h'_i}\right\}$, where h'_i is the total number of occurrences of task $\mathcal{T}_i$. We compute h'_i for each TT task from set $\mathcal{T}$ as

$$h'_i = \frac{\eta}{\tau_i^{TT}},$$

where η is a redefined hyperperiod, which is computed as

$$\eta = \mathrm{LCM}\left(\tau_1^{ET}, \ldots, \tau_n^{ET}, \tau_1^{TT}, \ldots, \tau_{n'}^{TT}\right).$$

The redefined hyperperiod η provides us with a sufficient observation time window thanks to the assumption that the deadlines of all tasks do not exceed their respective periods.

Each occurrence of each TT task consists of a set of TT jobs, which are dedicated to processors given by σ'_i. Therefore, we define $\mathcal{T}_{i,j} = \left\{T_{i,j,1}, \ldots, T_{i,j,l'_i}\right\}$, where job $T_{i,j,k} \in \mathcal{T}_{i,j}$ is dedicated to be executed on processor $\sigma'_{i,k}$, which corresponds to k-th element of the sequence of processors σ'_i. Note that the dedication of a TT job to a processor does not depend on the occurrence of the corresponding TT task. Therefore, $T_{i,j,k}$ is dedicated to processor $\sigma'_{i,k}$, $\forall j \in \{1, \ldots, h'_i\}$. Each TT job $T_{i,j,k}$ has deadline $d_{i,j,k}^{TT}$ and execution time $c_{i,j,k}^{TT}$. For $k = 1$, each TT job $T_{i,j,1}$ is also defined by its release time $r_{i,j,1}^{TT}$. The values of TT job parameters are assumed to satisfy the following constraints. $\forall i \in \{1, \ldots, n'\}$, $\forall j \in \{1, \ldots, h'_i\}$, $\forall k \in \{1, \ldots, l'_i\}$,

$$d_{i,j,k}^{TT} = d_i^{TT} + (j-1) \cdot \tau_i^{TT} - (l'_i - k) \cdot c_i^{TT},$$

$$c_{i,j,k}^{TT} = c_i^{TT},$$

and, $\forall i \in \{1, \ldots, n'\}$, $\forall j \in \{1, \ldots, h'_i\}$,

$$r_{i,j,1}^{TT} = r_i^{TT} + (j-1) \cdot \tau_i^{TT}.$$

The constraints show that each TT job inherits its execution time from the corresponding TT task. Release time of jobs $T_{i,j,1}$ is shifted according to the occurrence of the corresponding TT task as well as the deadline of each TT job $T_{i,j,k}$ which is in addition tightened by $(l'_i - k) \cdot c_i^{TT}$. This additional shift of deadline will be discussed shortly. To sum up, we can express $\mathcal{T}$ as a set of TT jobs in the following way:

$$\mathcal{T} = \left\{\left\{\left\{T_{1,1,1}, \ldots, T_{1,1,l'_1}\right\}, \ldots, \left\{\ldots, T_{1,h'_1,l'_1}\right\}\right\}, \ldots, \left\{\ldots, \left\{\ldots, T_{n',h'_{n'},l'_{n'}}\right\}\right\}\right\}.$$

The definition of precedence constraints restricting TT jobs is similar to that of ET jobs. When

$$T_{i,j,k_x} \prec T_{i,j,k_y},$$

we say that T_{i,j,k_x} precedes T_{i,j,k_y}. This means that T_{i,j,k_x} must finish its execution before T_{i,j,k_y} is released. If a TT job has some predecessors, then it is released when all its predecessors finish their execution. The precedence constraints are considered only for TT jobs of the same occurrence of a given TT task. Thanks to our definition of TT jobs, it always holds that

$$k_x < k_y \Leftrightarrow T_{i,j,k_x} \prec T_{i,j,k_y}, \quad \forall i \in \{1, \ldots, n'\}, \forall j \in \{1, \ldots, h'_i\}.$$

Similar to ET jobs, we can observe that TT jobs of the same occurrence of a given TT task form a precedence chain. Therefore, we consider only precedence chains consisting of TT jobs as precedence constraints restricting TT jobs in this work. The precedence chain can be expressed as

$$T_{i,j,1} \prec \cdots \prec T_{i,j,l'_i}, \quad \forall i \in \{1,\dots,n'\}, \forall j \in \{1,\dots,h'_i\}.$$

The fact that TT jobs of the same occurrence of a given TT task form a precedence chain causes the position of a TT job $T_{i,j,k}$ in the precedence chain to affect its deadline $d^{TT}_{i,j,k}$. That is why $d^{TT}_{i,j,k}$ is tightened by $(l'_i - k)\cdot c^{TT}_i$. Note that meeting deadlines $d^{TT}_{i,j,k}$, $\forall k \in \{1,\dots,l'_i\}$, ensures that the deadline of the j-th occurrence of a given TT task $\mathcal{T}_i$ is met.

2.6 Scheduling of TT jobs

The main difference between the scheduling of ET jobs and TT jobs is that TT jobs have to start their execution at predetermined time points. Therefore, the main goal of scheduling TT jobs is to find a set of start times which is defined as

$$\mathcal{S} = \left\{ s_{1,1,1},\dots,s_{1,1,l'_1}, s_{1,2,1},\dots\dots, s_{1,h'_1,l'_1}, s_{2,1,1},\dots\dots\dots, s_{n',h'_{n'},l'_{n'}} \right\},$$

where $s_{i,j,k}$ is a start time of TT job $T_{i,j,k}$, $\forall i \in \{1,\dots,n'\}$, $\forall j \in \{1,\dots,h'_i\}$, $\forall k \in \{1,\dots,l'_i\}$. The scheduling of TT jobs in the dedicated multiprocessor setting is done incrementally due to the precedence constraints. The release time of each TT job $T_{i,j,k+1}$, for $k < l'_i$, is revealed after assigning the start time to each of its predecessors in the precedence chain. Therefore, it holds that $r^{TT}_{i,j,k+1} = s_{i,j,k} + c^{TT}_{i,j,k}$, for $k < l'_i$, $\forall i \in \{1,\dots,n'\}$, $\forall j \in \{1,\dots,h'_i\}$. Moreover, we define T_π to be a set of all TT jobs dedicated to processor π, i.e., set of jobs $T_{i,j,k}$ for which $\sigma'_{i,k} = \pi$, $\forall i \in \{1,\dots,n'\}, \forall j \in \{1,\dots,h'_i\}, \forall k \in \{1,\dots,l'_i\}$.

The set of start times $\mathcal{S}$ is feasible if all deadlines are met, i.e., $s_{i,j,k} \in \left[r^{TT}_{i,j,k}, d^{TT}_{i,j,k} - c^{TT}_{i,j,k}\right]$, $\forall i \in \{1,\dots,n'\}, \forall j \in \{1,\dots,h'_i\}, \forall k \in \{1,\dots,l'_i\}$, and if the executions of jobs dedicated to the same processor do not overlap, i.e., $\left[s_{i,j,k}, s_{i,j,k} + c^{TT}_{i,j,k}\right) \cap \left[s_{i',j',k'}, s_{i',j',k'} + c^{TT}_{i',j',k'}\right) = \emptyset$, for all pairs of different TT jobs $T_{i,j,k}, T_{i',j',k'} \in T_\pi$, $\forall \pi \in \{1,\dots,m\}$.

2.7 Combining ET and TT tasks

When combining ET and TT tasks, all TT jobs must start their execution at predetermined start times. This must be considered when conducting the schedulability analysis for the ET jobs. Since all jobs are considered to be non-preemptive, the execution of an ET job cannot be interrupted by the online scheduler to allow a TT job to start its execution at its predetermined start time. Therefore, there must not be an ongoing execution of an ET job on a processor π at the time when some TT job dedicated to processor π is scheduled to start its execution. The situation when some TT job cannot start its execution at its predetermined start time due to an ongoing execution of an ET job is referred to as a collision.

Given a set of ET tasks $\mathcal{E}$, a set of TT tasks $\mathcal{T}$, a number of processors m, and a scheduling policy $\mathcal{P}$, the set of start times $\mathcal{S}$ is referred to as valid if $\mathcal{S}$ is feasible and if $\mathcal{E}$ is schedulable under policy $\mathcal{P}$ (i.e., no execution scenario results in a deadline miss) and if each TT job $T_{i,j,k}$ always starts its execution at its predetermined start time $s_{i,j,k}$ (i.e., no collisions occur). The main goal of combining ET and TT tasks is to find a set of start times that is valid. The problem of finding a valid set of start times may be infeasible, e.g., when $\mathcal{E}$ is not schedulable under policy $\mathcal{P}$.

Chapter 3

ET solutions

This chapter describes and thoroughly explains the algorithms used for the schedulability analysis of a set of ET tasks and their implementation. The baseline brute force approach to schedulability analysis is described in Section 3.1. Section 3.2 describes the worst-case approach to schedulability analysis. The novel approach to schedulability analysis based on the SAG framework is described in Section 3.3.

3.1 Brute force approach to schedulability analysis

The first approach to schedulability analysis, which is described in this section, is the brute force schedulability analysis. The main idea of this algorithm is quite simple as it simulates the work of an online scheduler for every possible execution scenario.

The pseudocode of the brute force schedulability analysis can be seen in Algorithm 1. The pseudocode consists of three functions. The brute force schedulability analysis is launched by calling the function BRUTE_FORCE_ANALYSIS. A set of ET tasks $\mathcal{E}$ as well as the number of processors m are passed to this function and it assigns release and execution times to the corresponding jobs for each execution scenario. After each such assignment, the function SIMULATE_EXECUTION_SCENARIO is called to determine whether the given execution scenario results in a deadline miss or not. If there is no such execution scenario that results in a deadline miss, then the instance is schedulable.

Furthermore, the decision of which job should be scheduled next is made in the function SCHEDULING_POLICY that requires finish times of all processors FT, a set of applicable jobs E^A, and time t as its inputs. We define applicable jobs E^A as set that contains ET jobs $E_{i,j,k}$ that satisfy the following conditions:

$$E_{i,j,k} \text{ is unfinished } \wedge \Big((j = 1 \wedge k = 1) \vee \\ (j > 1 \wedge k = 1 \wedge E_{i,j-1,l_i} \text{ is finished}) \vee (k > 1 \wedge E_{i,j,k-1} \text{ is finished}) \Big).$$

In other words, the set of applicable jobs always contains at most one ET job of each ET task. It is always the first unfinished job of the corresponding task. If there is no finished job of the corresponding task, then it is the job of the task's first occurrence that has no predecessors. If all jobs of an occurrence of the corresponding task are finished, then it is the job of the task's next occurrence that has no predecessors. If there are some but not all finished jobs within one occurrence of the corresponding task, then it is the job within the same occurrence whose predecessors are all finished. We can also observe that it holds that

$\left|E^A\right| \leq n$. Setting the jobs to be either unfinished or finished is taken care of by the function SIMULATE_EXECUTION_SCENARIO on lines 14 and 28, respectively.

Furthermore, we define finish times of all processors as $FT = \{FT_1, \ldots, FT_m\}$, where FT_π is a time until which processor π is occupied.

Lines 17-21 in the function SIMULATE_EXECUTION_SCENARIO set the finish times of all processors in such a way that enables the function SCHEDULING_POLICY to return some job $E_{i,j,k}$ each time it is called. This change to the finish times of all processors on line 21 can be done since no unfinished job can start its execution before time t. Moreover, line 27 in the function SIMULATE_EXECUTION_SCENARIO shows that successors in the precedence chain have their release times revealed based on when their direct predecessor finished its execution.

Algorithm 1 Brute force approach to schedulability analysis

```
    Input: set of ET tasks $\mathcal{E}$, number of processors $m$
    Output: true if the set of ET tasks $\mathcal{E}$ is schedulable under policy $\mathcal{P}$, false otherwise

 1: function BRUTE_FORCE_ANALYSIS($\mathcal{E}, m$)
 2:     for each unique execution scenario $\gamma = \{R, C\}$ do
 3:         for each job $E_{i,j,1} \in \mathcal{E}$ do
 4:             set $r^{ET}_{i,j,1}$ according to $\gamma$
 5:         for each job $E_{i,j,k} \in \mathcal{E}$ do
 6:             set $c^{ET}_{i,j,k}$ according to $\gamma$
 7:         if ¬SIMULATE_EXECUTION_SCENARIO($\mathcal{E}, m$) then
 8:             return false
 9:     return true                                  ▷ No execution scenario resulted in a deadline miss
10: function SIMULATE_EXECUTION_SCENARIO($\mathcal{E}, m$)
11:     for $\pi = 1$ to $m$ do
12:         $FT_\pi \leftarrow 0$
13:     $FT \leftarrow \{FT_1, \ldots, FT_m\}$
14:     set all jobs from $\mathcal{E}$ to unfinished
15:     $E^A \leftarrow$ applicable jobs from $\mathcal{E}$
16:     while $E^A \neq \emptyset$ do
17:         $t \leftarrow \infty$
18:         for each $E_{i',j',k'} \in E^A$ do
19:             $t \leftarrow \min(t, \max(r^{ET}_{i',j',k'}, FT_{\sigma_{i',k'}}))$
20:         for each $FT_\pi \in FT$ do
21:             $FT_\pi \leftarrow \max(FT_\pi, t)$
22:         $E_{i,j,k} \leftarrow$ SCHEDULING_POLICY($FT, E^A, t$)
23:         if $FT_{\sigma_{i,k}} + c^{ET}_{i,j,k} > d^{ET}_{i,j,k}$ then
24:             return false                                                   ▷ Deadline miss
25:         $FT_{\sigma_{i,k}} \leftarrow FT_{\sigma_{i,k}} + c^{ET}_{i,j,k}$
26:         if $E_{i,j,k}$ has a direct successor $E_{i,j,k+1}$ in the precedence chain then
27:             $r^{ET}_{i,j,k+1} \leftarrow FT_{\sigma_{i,k}}$
28:         set $E_{i,j,k}$ to finished
29:         $E^A \leftarrow$ applicable jobs from $\mathcal{E}$
30:     return true                       ▷ Each job finished its execution with no deadline miss
31: function SCHEDULING_POLICY($FT, E^A, t$)
32:     $E^R \leftarrow$ subset of $E^A$ with only released jobs       ▷ Job $E_{i,j,k} \in E^R \subseteq E^A$ iff $r^{ET}_{i,j,k} \leq t$
33:     $\pi \leftarrow$ arbitrary processor st. $FT_\pi = t \wedge E^R \cap E_\pi \neq \emptyset$     ▷ $E_\pi$ is a set of all ET jobs dedicated to $\pi$
34:     $E_{i,j,k} \leftarrow \mathcal{P}\left(t, \pi, E^R \cap E_\pi\right)$
35:     return $E_{i,j,k}$                                     ▷ Job picked by the scheduling policy
```

The implemented function SCHEDULING_POLICY works with unfinished released jobs E^R at time t. E^R is a subset of applicable jobs E^A. Note that time t, which is obtained on line 19 in the function SIMULATE_EXECUTION_SCENARIO and then used as an input of the function SCHEDULING_POLICY, guarantees that E^R is not an empty set. Moreover, there is always at least one processor that satisfies the conditions on line 33 thanks to the change to finish times of all processors on line 21 in the function SIMULATE_EXECUTION_SCENARIO. Since the scheduling policy $\mathcal{P}$ is defined to be invoked for each processor independently, we can choose an arbitrary processor satisfying the conditions on line 33. The policy $\mathcal{P}$ then picks some ET job $E_{i,j,k}$ based on its decision rules.

One of the main advantages of this algorithm is that it provides exact schedulability analysis since it simulates all possible execution scenarios and it does not add any execution scenarios that are not possible to occur for a given set of ET tasks. Therefore, it is used as a baseline approach in empirical verification. Another advantage of this algorithm is its simple implementation. Moreover, the brute force schedulability analysis is modular concerning decision rules of the scheduling policy $\mathcal{P}$. Therefore, the FP-EDF policy, which is considered in this work, can be replaced by any other work-conserving policy while preserving the correctness of the brute force schedulability analysis.

The only disadvantage of the brute force schedulability analysis is that it suffers severely from combinatorial explosion. The number of execution scenarios, which the algorithm simulates, can be enormous. For a given set of ET tasks $\mathcal{E}$, the number of execution scenarios $N^{\mathcal{E}}$ can be expressed using the parameters of ET jobs in a closed form as

$$N^{\mathcal{E}} = \prod_{i=1}^{n} \prod_{j=1}^{h_i} \left(\left(r_{i,j,1}^{max} + 1 - r_{i,j,1}^{min} \right) \cdot \prod_{k=1}^{l_i} \left(c_{i,j,k}^{max} + 1 - c_{i,j,k}^{min} \right) \right),$$

or it can be equivalently expressed using the parameters of ET tasks as

$$N^{\mathcal{E}} = \prod_{i=1}^{n} \left(\left(r_i^{max} + 1 - r_i^{min} \right) \cdot \left(c_i^{max} + 1 - c_i^{min} \right)^{l_i} \right)^{h_i}.$$

Let us consider the following example given by Table 3.1 to show how enormous can $N^{\mathcal{E}}$ be even for a relatively small instance. Only 4 ET tasks are considered in this example.

ET task	r^{min}	r^{max}	c^{min}	c^{max}	h	l
$\mathcal{E}_1$	0	9	1	10	1	5
$\mathcal{E}_2$	0	9	1	10	2	4
$\mathcal{E}_3$	0	9	1	10	4	3
$\mathcal{E}_4$	0	9	1	10	8	2

Table 3.1: Example of ET task parameters for calculation of $N^{\mathcal{E}}$, $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4\}$.

For $\mathcal{E}$ given by Table 3.1, we can calculate $N^{\mathcal{E}}$ as

$$N^{\mathcal{E}} = \left(10 \cdot 10^5\right)^1 \cdot \left(10 \cdot 10^4\right)^2 \cdot \left(10 \cdot 10^3\right)^4 \cdot \left(10 \cdot 10^2\right)^8 = 10^{56}.$$

This means that there are in total 10^{56} possible execution scenarios to be simulated by the brute force schedulability analysis in this example.

3.2 Worst-case approach to schedulability analysis

The worst-case approach simulates the work of an online scheduler for one specific execution scenario. In this specific execution scenario $\gamma = \{R, C\}$, the set of release times R is considered to consist of the latest release times and the set of execution times C is considered to consist of the longest execution times. If a deadline miss occurs during the simulation of this execution scenario, then we know that the entire instance is unschedulable. This is the reason why we call this approach "worst-case".

The pseudocode of the worst-case approach can be seen in Algorithm 2. It simply initializes the execution scenario by setting release times to the respective latest release times and by setting execution times to the respective longest execution times. The execution scenario is then simulated using the function SIMULATE_EXECUTION_SCENARIO from Algorithm 1. If a deadline miss occurs during the simulation, then the instance is unschedulable. Otherwise, the algorithm deems the instance schedulable.

However, note that the term worst-case is a mere name. This approach provides only a necessary schedulability analysis. This means that this approach can deem some instances schedulable even though they are in fact unschedulable. In other words, we cannot be sure that such a specifically chosen execution scenario leads to the true worst-case execution. In fact, considering ET jobs to be non-preemptive causes scheduling anomalies to occur. Therefore, we cannot say in advance which execution scenario would lead to the true worst-case execution without simulating all of them.

Algorithm 2 Worst-case approach to schedulability analysis

```
Input: set of ET tasks $\mathcal{E}$, number of processors $m$
Output: true if the worst-case execution scenario for $\mathcal{E}$ does not result in a deadline miss, false otherwise

1: function WORST_CASE_ANALYSIS($\mathcal{E}, m$)
2:     for each job $E_{i,j,1} \in \mathcal{E}$ do
3:         $r^{ET}_{i,j,1} \leftarrow r^{max}_{i,j,1}$
4:     for each job $E_{i,j,k} \in \mathcal{E}$ do
5:         $c^{ET}_{i,j,k} \leftarrow c^{max}_{i,j,k}$
6:     return SIMULATE_EXECUTION_SCENARIO($\mathcal{E}, m$)
```

The main advantage of this approach is that it is computationally inexpensive. Therefore, it is used as a potential speedup in the SAG approach, which can be seen later in Section 3.3.

3.3 Schedule Abstraction Graph approach to schedulability analysis

Another approach to schedulability analysis for tasks on dedicated resources with precedences is based on the so-called Schedule Abstraction Graph (SAG) framework, which was introduced in 2017 by Mitra Nasri and Björn B. Brandenburg in [16]. This framework provided an exact and scalable ET schedulability analysis for non-preemptive tasks on a single processor. The SAG in their work encompasses all possible execution scenarios and merges similar scenarios, thus making the algorithm scalable.

This section describes what SAG is and how it is utilized in the dedicated multiprocessor setting. Furthermore, a high-level description of the SAG generation procedure, which is inspired by [14, 15], will be provided. A low-level description of the SAG generation with pseudocode will be provided afterward. Finally, an example demonstrating the SAG generation procedure step-by-step is given. Also, note that this section makes use of OOP notation to express the parameters of objects.

3.3.1 Schedule Abstraction Graph

The SAG can be intuitively described as a directed acyclic graph. Each vertex of the SAG represents a state that contains a set of finished jobs and an occupation state of each processor represented by a time range that corresponds to times when the processor may finish the execution of a job. Each edge of the SAG characterizes the execution of an ET job. The time ranges are utilized due to the presence of release jitter and execution time variation. The SAG is generated to find out whether a set of ET tasks $\mathcal{E}$ is schedulable under policy $\mathcal{P}$.

The SAG starts with a root vertex with no finished jobs and the time ranges of all processors initialized to $[0, 0]$. The SAG is then gradually built from the root vertex. It picks a job that should be scheduled next based on rules that will be explained later. After a job is picked, a new edge and a new vertex are added to the SAG. The new edge connects the vertex where the job was picked to the new vertex. Each vertex can have multiple edges emanating from the vertex, i.e., multiple jobs can be picked in each vertex. After the branching of the root vertex is done, we end up with another layer of vertices. Therefore, it is useful to see the SAG as a directed level-structured graph. We repeat the same expansion procedure for each layer of vertices. Moreover, vertices of the same layer can be merged before their expansion under conditions described later.

Formally, we define the SAG as a directed acyclic graph $\mathcal{G}^{SAG} = \left(\mathcal{V}^{SAG}, \mathcal{E}^{SAG}\right)$, where each vertex has a label that consists of the earliest finish time EFT and latest finish time LFT of each processor. State of occupation of processor π is therefore defined by finish time interval $[EFT_\pi, LFT_\pi]$. This interval represents times when the processor may finish the execution of the last scheduled job and thus may become unoccupied. Finish time intervals of all processors are defined as a set $FTI = \{[EFT_1, LFT_1], \dots, [EFT_m, LFT_m]\}$. Therefore, for each vertex $v \in \mathcal{V}^{SAG}$, we denote finish time intervals of all processors as $v.FTI = \{[v.EFT_1, v.LFT_1], \dots, [v.EFT_m, v.LFT_m]\}$. Furthermore, each edge $e \in \mathcal{E}^{SAG}$ corresponds to a single job labeled with $e.J$. There can be multiple edges in the SAG with the same label.

Since the SAG is a directed level-structured graph, $\mathcal{V}^{SAG}$ can be divided into disjoint sets of vertices based on their distance from the root vertex v_r. Distance of vertex v from v_r is the length of any directed path from v_r to v. We define V_ι as a set of all vertices from $\mathcal{V}^{SAG}$ with distance from v_r equal to ι. Moreover, $V_0 = \{v_r\}$.

3.3.2 SAG generation

The generation algorithm begins with initializing the root vertex v_r. It sets $v_r.FTI$ to $\{[0, 0], \dots, [0, 0]\}$ and $V_0 = \{v_r\}$. Then it keeps alternating between two different phases. The first one is called expansion phase and the second one is called merge phase. In the expansion phase, all vertices of V_ι are expanded and a new layer of vertices $V_{\iota+1}^{ex}$ is generated. The merge phase is then performed upon vertices from $V_{\iota+1}^{ex}$. The merge phase tries to merge some vertices from $V_{\iota+1}^{ex}$ with each other. After the merge phase is done, we end up with the layer of merged vertices $V_{\iota+1}$. The algorithm keeps repeating the expansion phase and the merge phase until a layer $V_{|\mathcal{E}|}$ is generated, where $|\mathcal{E}|$ is the total number of ET jobs in $\mathcal{E}$. The procedure of SAG generation can also terminate earlier when a deadline miss occurs. If a deadline miss occurs during SAG generation, then $\mathcal{E}$ is not schedulable under policy $\mathcal{P}$. If the entire SAG is generated without encountering a deadline miss, then $\mathcal{E}$ is schedulable under policy $\mathcal{P}$.

3.3.3 Expansion phase

Each vertex $v \in V_\iota$ is characterized by finish time intervals of all processors $v.FTI$ and by applicable jobs in this vertex, which is denoted as $v.E^A$. The applicable jobs are acquired based on the position of the vertex in the SAG. Jobs that are the labels of edges along any directed path from root vertex v_r to v constitute a set of finished jobs $v.E^F$. The set of applicable jobs in vertex v is then acquired based on $v.E^F$. With knowledge of $v.FTI$ and $v.E^A$ some jobs from $v.E^A$ are picked to be executed next. This adds new edges and vertices to the SAG. After each vertex of V_ι is expanded, a set of newly generated vertices $V^{ex}_{\iota+1}$ is processed in the merge phase.

3.3.4 Merge phase

The merge phase takes a set of newly generated vertices $V^{ex}_{\iota+1}$, which is the result of the last expansion phase, and tries to merge some vertices from this set with each other. Note that the earliest and the latest release time of each ET job that has some predecessors in the precedence chain are not part of the problem definition as their values depend on the time when the predecessors finish their execution. Consequently, each such ET job may have a different earliest or latest release time in two different vertices in the SAG. Therefore, for each vertex $v \in \mathcal{V}^{SAG}$ and for each ET job $E_{i,j,k} \in v.E^A$, we define $v.r^{min}_{i,j,k}$ and $v.r^{max}_{i,j,k}$ to be the earliest and the latest release time of $E_{i,j,k}$ in vertex v, respectively. Two vertices $v, w \in V^{ex}_{\iota+1}$ can be merged if each of the following three conditions holds.

- $v.E^F = w.E^F$.
- $[v.EFT_\pi, v.LFT_\pi] \cap [w.EFT_\pi, w.LFT_\pi] \neq \emptyset, \forall \pi \in \{1, \dots, m\}$.
- $\forall E_{i,j,k} \in v.E^A \cap w.E^A$ st. $k > 1 \land \left(v.r^{min}_{i,j,k} \neq w.r^{min}_{i,j,k} \lor v.r^{max}_{i,j,k} \neq w.r^{max}_{i,j,k}\right)$ the following condition must be satisfied. Considering $EFT^{max} \leftarrow \max\left(v.EFT_{\sigma_{i,k}}, w.EFT_{\sigma_{i,k}}\right)$, if $v.r^{max}_{i,j,k} = w.r^{max}_{i,j,k}$, then $v.r^{min}_{i,j,k} \leq EFT^{max} \land w.r^{min}_{i,j,k} \leq EFT^{max}$ must hold, otherwise $v.r^{max}_{i,j,k} \leq EFT^{max} \land w.r^{max}_{i,j,k} \leq EFT^{max}$ must hold.

If these conditions are met, then v and w are merged. This means that all edges incident to w are redirected to be incident to v, both earliest and latest release times of some jobs in $v.E^A$ are updated to their respective modified versions, which is thoroughly described in pseudocode of the merge phase, and the finish time intervals of all processors in vertex v are updated as $[v.EFT_\pi, v.LFT_\pi] \leftarrow [v.EFT_\pi, v.LFT_\pi] \cup [w.EFT_\pi, w.LFT_\pi]$, $\forall \pi \in \{1, \dots, m\}$. Finally, vertex w gets deleted from the SAG. The merge phase terminates when there are no pairs of vertices left in $V^{ex}_{\iota+1}$ that can be merged. The result of the merge phase is a set of vertices $V_{\iota+1}$. Afterward, the expansion phase is called upon this set and the process repeats.

3.3.5 Parameters of vertices and edges

Regarding the parameters of vertices, each vertex $v \in \mathcal{V}^{SAG}$ has the finish time intervals of all processors $v.FTI$ and the set of finished jobs $v.E^F$ from which we can derive the set of applicable jobs $v.E^A$. These parameters are already defined. Moreover, each vertex v has a set of incoming edges denoted as $v.in$ and a set of outgoing edges denoted as $v.out$. The parameters of each edge $e \in \mathcal{E}^{SAG}$ are its job label $e.J$, source vertex $e.s$, and destination vertex $e.d$.

3.3.6 Job eligibility

This part explains the main rules based on which jobs are picked during the expansion phase in some vertex $v \in \mathcal{V}^{SAG}$. Job eligibility provides us time widows when a job may start its execution. First, we define a few terms regarding the release state of an ET job and the occupation state of a processor. Given a set of applicable jobs $v.E^A$, we say that an ET job $E_{i,j,k} \in v.E^A$ is possibly released at time t when $v.r^{min}_{i,j,k} \leq t < v.r^{max}_{i,j,k}$, and that $E_{i,j,k}$ is certainly released at time t when $v.r^{max}_{i,j,k} \leq t$. Furthermore, given finish time intervals of all processors $v.FTI$, we say that a processor π is certainly occupied at time t when $t < v.EFT_\pi$, that it is possibly occupied at time t when $v.EFT_\pi \leq t < v.LFT_\pi$, and it is unoccupied at time t when $v.LFT_\pi \leq t$.

Given a set of applicable jobs $v.E^A$, a processor π, and jobs $E_{i,j,k}, E_{i',j',k'} \in v.E^A \cap E_\pi$, we say that job $E_{i,j,k}$ has higher policy priority than job $E_{i',j',k'}$ if the scheduling policy $\mathcal{P}\left(\infty, \pi, \{E_{i,j,k}, E_{i',j',k'}\}\right)$ returns $E_{i,j,k}$. That is, the scheduling policy would pick $E_{i,j,k}$ given that both $E_{i,j,k}$ and $E_{i',j',k'}$ are certainly released. In this work, the FP-EDF scheduling policy is considered as policy $\mathcal{P}$. However, without loss of generality, any other work-conserving policy can be used as policy $\mathcal{P}$ instead.

For a given set of applicable jobs $v.E^A$, a processor π, and time t, an ET job E^t_{ce} is certainly-eligible at time t if E^t_{ce} is certainly released at time t, if processor π to which E^t_{ce} is dedicated is not certainly occupied at time t, and if there is no other certainly released applicable job at time t that is dedicated to processor π with higher policy priority than E^t_{ce}. Formally, $E^t_{ce} \in v.E^A \cap E_\pi$ is certainly-eligible at time t if

$$\begin{aligned} &E^t_{ce}.r^{max} \leq t \land v.EFT_\pi \leq t \land \nexists E_{i,j,k} \in v.E^A \cap E_\pi \text{ st.} \\ &\quad E_{i,j,k} \neq E^t_{ce} \land v.r^{max}_{i,j,k} \leq t \land E_{i,j,k} = \mathcal{P}\left(\infty, \pi, \left\{E_{i,j,k}, E^t_{ce}\right\}\right). \end{aligned}$$

We can observe that for each processor there may be at most one certainly-eligible job at time t. We say that E^t_{ce} dedicated to processor π does not exist when there is no certainly-eligible job at time t dedicated to processor π.

Furthermore, for a given set of applicable jobs $v.E^A$, a processor π, and time t, an ET job $E_{i,j,k}$ is possibly-eligible at time t if $E_{i,j,k}$ is possibly released at time t, if processor π to which $E_{i,j,k}$ is dedicated is not certainly occupied at time t, and if either E^t_{ce} dedicated to the same processor as $E_{i,j,k}$ does not exist or if $E_{i,j,k}$ has higher policy priority than E^t_{ce}. Note that there can be multiple possibly-eligible jobs at time t dedicated to the same processor. Formally, assuming dedication of E^t_{ce} to processor π, we define the set of possibly-eligible jobs at time t dedicated to processor π as

$$\Big\{E_{i,j,k} \,\Big|\, E_{i,j,k} \in v.E^A \cap E_\pi \wedge v.r^{min}_{i,j,k} \leq t < v.r^{max}_{i,j,k} \wedge v.EFT_\pi \leq t \wedge \\ \Big(E^t_{ce} \text{ does not exist} \vee E_{i,j,k} = \mathcal{P}\left(\infty, \pi, \left\{E_{i,j,k}, E^t_{ce}\right\}\right)\Big)\Big\}.$$

3.3.7 Pseudocode

Expansion phase

The pseudocode of the expansion phase can be seen in Algorithm 3. The expansion phase begins with a set of vertices V_ι and it gradually expands each vertex $v \in V_\iota$ by calling the function GET_NEXT_NODES that returns a set of vertices V^v that are the result of the expansion of vertex v. The pseudocode of the function GET_NEXT_NODES provides a high-level view of how a vertex gets expanded. If the vertex has no applicable jobs left, then we know that we reached the last layer of the SAG. Otherwise, a time window $[t_{min}, t_{max}]$ is acquired on lines 14-15. Here, t_{min} corresponds to the minimal time when some job from $v.E^A$ is either possibly or certainly-eligible. Furthermore, t_{max} corresponds to the minimal time when some job from $v.E^A$ is certainly-eligible and it is dedicated to a processor that is unoccupied at that time. Considering a larger time window would result in considering not possible execution scenarios. The earliest start time (EST) and the latest start time (LST) on lines 19-20 give us the boundary values of times when the corresponding job may start its execution.

The function EXPAND_VERTEX generates a new vertex and a new edge in the SAG. In the new vertex, the finish time interval of the processor to which the scheduled job $E_{i,j,k}$ is dedicated is modified accordingly as can be seen on lines 26-27. Moreover, if $E_{i,j,k}$ has a direct successor in the precedence chain, then the earliest and the latest release time of such direct successor are assigned the earliest and the latest time when $E_{i,j,k}$ may finish its execution, respectively.

Detailed version of GET_NEXT_NODES

Algorithm 4 provides a low-level explanation and implementation of the function GET_NEXT_NODES from Algorithm 3. The main idea of Algorithm 4 is that we are searching for time windows when jobs are either possibly or certainly-eligible. Since we consider only work-conserving policies, namely the FP-EDF policy, there is potentially only one such time window for each job in each vertex of the SAG. In other words, if a job ceases to be either certainly-eligible or possibly-eligible without consequently becoming certainly-eligible in some vertex v in the SAG, then the job can never become possibly or certainly-eligible again in this vertex. Recall that the time window when a job is either possibly or certainly-eligible corresponds to times when the job may start its execution. Moreover, this procedure is done for each processor separately.

Variables CE_π and PE_π, which are initialized on lines 5-6 in Algorithm 4, are utilized to store a certainly-eligible job and a set of possibly-eligible jobs dedicated to processor π, respectively. In this algorithm, the *null* pointer is used to express that something does not exist, e.g., that E^t_{ce} does not exist. The algorithm then iterates over all times $t \in [t_{min}, t_{max} + 1]$,

Algorithm 3 Expansion phase

```
Input: set of vertices $V_\iota$
Output: set of vertices $V^{ex}_{\iota+1}$

function EXPANSION_PHASE($V_\iota$)
    $V^{ex}_{\iota+1} \leftarrow \emptyset$
    for each $v \in V_\iota$ do
        $V^v \leftarrow$ GET_NEXT_NODES($v$)
        $V^{ex}_{\iota+1} \leftarrow V^{ex}_{\iota+1} \cup V^v$
    return $V^{ex}_{\iota+1}$
function GET_NEXT_NODES($v$)
    if $v.E^A = \emptyset$ then
        return $\emptyset$                                        ▷ All jobs are finished
    $V^v \leftarrow \emptyset$
    $t_{min} \leftarrow \infty$
    $t_{max} \leftarrow \infty$
    for each $E_{i,j,k} \in v.E^A$ do
        $t_{min} \leftarrow \min(t_{min},\ \max(v.r^{min}_{i,j,k}, v.EFT_{\sigma_{i,k}}))$
        $t_{max} \leftarrow \min(t_{max},\ \max(v.r^{max}_{i,j,k}, v.LFT_{\sigma_{i,k}}))$
    for each $E_{i,j,k} \in v.E^A$ do
        if $\max(v.r^{min}_{i,j,k}, v.EFT_{\sigma_{i,k}}) > t_{max}$ then
            continue                ▷ The job cannot be scheduled within the given time window
        $[EST, LST] \leftarrow$ all times when $E_{i,j,k}$ is possibly or certainly-eligible converted to closed interval
        $[EST, LST] \leftarrow [EST, LST] \cap [t_{min}, t_{max}]$
        $w \leftarrow$ EXPAND_VERTEX($v, E_{i,j,k}, EST, LST$)
        $V^v \leftarrow V^v \cup \{w\}$
    return $V^v$
function EXPAND_VERTEX($v, E_{i,j,k}, EST, LST$)
    $w \leftarrow$ new vertex with $w.FTI \leftarrow v.FTI, w.E^F \leftarrow v.E^F \cup \{E_{i,j,k}\}, w.out \leftarrow \emptyset$
    $w.EFT_{\sigma_{i,k}} \leftarrow EST + c^{min}_{i,j,k}$
    $w.LFT_{\sigma_{i,k}} \leftarrow LST + c^{max}_{i,j,k}$
    if $E_{i,j,k}$ has a direct successor $E_{i,j,k+1}$ in the precedence chain then
        $w.r^{min}_{i,j,k+1} \leftarrow w.EFT_{\sigma_{i,k}}$
        $w.r^{max}_{i,j,k+1} \leftarrow w.LFT_{\sigma_{i,k}}$
    $e \leftarrow$ new edge with $e.J \leftarrow E_{i,j,k}, e.s \leftarrow v, e.d \leftarrow w$
    $w.in \leftarrow \{e\}$
    $v.out \leftarrow v.out \cup \{e\}$
    $w.E^A \leftarrow$ applicable jobs based on $w.E^F$
    return $w$
```

where t_{min} and t_{max} are obtained in the same way as in Algorithm 3, and, for each time t, the algorithm iterates over all processors.

On line 18, a certainly-eligible job at time t dedicated to processor π may not exist. In that case, E^t_{ce} is set to $null$. If E^t_{ce} exists and if it is different from the job stored in CE_π, then we know that we obtained a new certainly-eligible job dedicated to processor π. In that case, if there is some job stored in CE_π, then the vertex v gets expanded since the job stored in CE_π is no longer certainly-eligible at time t. This expansion can be seen on line 21. Moreover, if E^t_{ce} was already stored in the set of possibly-eligible jobs dedicated to processor π, then it gets deleted from this set since it became certainly-eligible. E^t_{ce} is stored in CE_π afterwards. Furthermore, if some job from PE_π is no longer possibly-eligible at time t, then the vertex v gets expanded, which can be seen on line 28, and the job gets deleted from PE_π. Finally, we add jobs dedicated to processor π that became possibly-eligible at time t to PE_π.

Algorithm 4 Detailed version of GET_NEXT_NODES

```
    Input: vertex $v$
    Output: set of vertices $V^v$

 1: function GET_NEXT_NODES($v$)
 2:     if $v.E^A = \emptyset$ then
 3:         return $\emptyset$
 4:     for $\pi = 1$ to $m$ do
 5:         $CE_\pi \leftarrow null$
 6:         $PE_\pi \leftarrow \emptyset$
 7:     $V^v \leftarrow \emptyset$
 8:     $t_{min} \leftarrow \infty$
 9:     $t_{max} \leftarrow \infty$
10:     for each $E_{i,j,k} \in v.E^A$ do
11:         $t_{min} \leftarrow \min(t_{min},\ \max(v.r^{min}_{i,j,k}, v.EFT_{\sigma_{i,k}}))$
12:         $t_{max} \leftarrow \min(t_{max},\ \max(v.r^{max}_{i,j,k}, v.LFT_{\sigma_{i,k}}))$
13:     for $t = t_{min}$ to $t_{max} + 1$ do
14:         for $\pi = 1$ to $m$ do
15:             if $t = t_{max} + 1$ then
16:                 $E^t_{ce} \leftarrow$ dummy job st. $E^t_{ce} = \mathcal{P}\left(\infty, \pi, \{E_{i,j,k}, E^t_{ce}\}\right)$ for any job $E_{i,j,k} \in v.E^A \cap E_\pi$
17:             else
18:                 $E^t_{ce} \leftarrow$ certainly-eligible job at time $t$ from $v.E^A \cap E_\pi$ or $null$
19:             if $E^t_{ce} \neq null \wedge E^t_{ce} \neq CE_\pi$ then
20:                 if $CE_\pi \neq null$ then
21:                     $w \leftarrow$ EXPAND_VERTEX($v, CE_\pi, \max(CE_\pi.r^{min}, v.EFT_\pi), t-1$)
22:                     $V^v \leftarrow V^v \cup \{w\}$
23:                 if $E^t_{ce} \in PE_\pi$ then
24:                     $PE_\pi \leftarrow PE_\pi \setminus \{E^t_{ce}\}$
25:                 $CE_\pi \leftarrow E^t_{ce}$
26:                 for each $E_{i,j,k} \in PE_\pi$ do
27:                     if $E_{i,j,k}$ is not possibly-eligible at time $t$ then
28:                         $w \leftarrow$ EXPAND_VERTEX($v, E_{i,j,k}, \max(v.r^{min}_{i,j,k}, v.EFT_\pi), t-1$)
29:                         $V^v \leftarrow V^v \cup \{w\}$
30:                         $PE_\pi \leftarrow PE_\pi \setminus \{E_{i,j,k}\}$
31:             for each $E_{i,j,k} \in v.E^A \cap E_\pi$ do
32:                 if $E_{i,j,k}$ is possibly-eligible at time $t \wedge E_{i,j,k} \notin PE_\pi$ then
33:                     $PE_\pi \leftarrow PE_\pi \cup \{E_{i,j,k}\}$
34:     return $V^v$
```

The last iteration of the outermost for-loop in Algorithm 4, i.e., when $t = t_{max} + 1$, ensures that the vertex v gets expanded according to the remaining possibly and certainly-eligible jobs across all processors. This is done by creating a dummy job E^t_{ce} on line 16 that has higher policy priority than any other job. Considering the FP-EDF policy, this can be achieved by setting $E^t_{ce}.p = -1$.

Merge phase

The pseudocode of the merge phase can be seen in Algorithm 5. The merge phase keeps merging vertices until there is no pair of vertices left that satisfies the three conditions that allow a pair of vertices to be merged.

The first condition $v.E^F = w.E^F$ is explicitly stated on line 2. The second condition is expressed by the function FTI_INTERSECT which checks whether the corresponding finish time intervals of vertices v and w intersect. The third condition is described by functions MERGEABLE_JOBS and RELEASE_TIMES_CONDITION. The main idea of this condition is that a job that has some predecessors in the precedence chain can have different earliest or latest release times in two different vertices in the SAG. Therefore, if we want to merge a pair of vertices v, w, then the earliest and latest release times of the job in vertices v and w must have one of two properties that are described in the function RELEASE_TIMES_CONDITION. The first property, which can be seen on lines 27-28, is that if the latest release times are the same, then the earliest release times must be less than or equal to EFT^{max} that was precomputed on line 22. The second property, which can be seen on lines 30-31, is that if the latest release times are different, then the latest release times must be less than or equal to EFT^{max}, which, if satisfied, also guarantees the earliest release times to be less than or equal to EFT^{max}. To put it another way, these two properties guarantee that at each time $t \geq EFT^{max}$ a job is either possibly released in both vertices v, w or it is certainly released in both vertices v, w. Note that merging two vertices when there is a job that is simultaneously possibly released in one of the vertices and certainly released in the other vertex at some time $t \geq EFT^{max}$ would lead to considering an impossible execution scenario.

If the merging conditions are satisfied, then the algorithm merges vertices v and w into one vertex v, and the parameters of v are updated. First, all jobs that have different earliest or latest release times in vertices v and w get their respective earliest and latest release time updated, which can be seen on lines 5-6. The line 5 corresponds to acquiring the minimum time across both vertices v, w when a job may become possibly-eligible. The line 6 corresponds to acquiring the minimum time across both vertices v, w when a job may become certainly-eligible. Afterward, finish time intervals of all processors in vertex v are updated as stated on lines 7-9. Furthermore, all edges incident to w are redirected to be incident to v. Finally, the vertex w gets removed from the SAG.

Complete SAG generation

Algorithm 6 provides the pseudocode of the complete SAG generation procedure. As can be seen on line 2, the procedure starts with conducting the worst-case approach to schedulability analysis, which is described in Algorithm 2 in Section 3.2. This can potentially significantly reduce the runtime of the SAG procedure since we know that if an instance is deemed

Algorithm 5 Merge phase

```
Input: set of vertices $V^{ex}_{\iota+1}$
Output: none, changes are done locally

 1: function MERGE_PHASE($V^{ex}_{\iota+1}$)
 2:     while $\exists v, w \in V^{ex}_{\iota+1}$ st. $v.E^F = w.E^F \wedge$ FTI_INTERSECT$(v, w) \wedge$ MERGEABLE_JOBS$(v, w)$ do
 3:         for each job $E_{i,j,k} \in v.E^A \cap w.E^A$ do
 4:             if $k > 1 \wedge (v.r^{min}_{i,j,k} \neq w.r^{min}_{i,j,k} \vee v.r^{max}_{i,j,k} \neq w.r^{max}_{i,j,k})$ then
 5:                 $v.r^{min}_{i,j,k} \leftarrow \min(\max(v.r^{min}_{i,j,k}, v.EFT_{\sigma_{i,k}}), \max(w.r^{min}_{i,j,k}, w.EFT_{\sigma_{i,k}}))$
 6:                 $v.r^{max}_{i,j,k} \leftarrow \min(\max(v.r^{max}_{i,j,k}, v.EFT_{\sigma_{i,k}}), \max(w.r^{max}_{i,j,k}, w.EFT_{\sigma_{i,k}}))$
 7:         for $\pi = 1$ to $m$ do
 8:             $v.EFT_\pi \leftarrow \min(v.EFT_\pi, w.EFT_\pi)$
 9:             $v.LFT_\pi \leftarrow \max(v.LFT_\pi, w.LFT_\pi)$
10:         for each edge $e \in w.in$ do
11:             $e.d \leftarrow v$
12:             $v.in \leftarrow v.in \cup \{e\}$
13:         remove $w$
14: function FTI_INTERSECT$(v, w)$
15:     for $\pi = 1$ to $m$ do
16:         if $[v.EFT_\pi, v.LFT_\pi] \cap [w.EFT_\pi, w.LFT_\pi] = \emptyset$ then
17:             return false
18:     return true
19: function MERGEABLE_JOBS$(v, w)$
20:     for each job $E_{i,j,k} \in v.E^A \cap w.E^A$ do                  ▷ $v.E^A = w.E^A$
21:         if $k > 1 \wedge (v.r^{min}_{i,j,k} \neq w.r^{min}_{i,j,k} \vee v.r^{max}_{i,j,k} \neq w.r^{max}_{i,j,k})$ then
22:             $EFT^{max} \leftarrow \max(v.EFT_{\sigma_{i,k}}, w.EFT_{\sigma_{i,k}})$
23:             if ¬RELEASE_TIMES_CONDITION$(EFT^{max}, v.r^{min}_{i,j,k}, v.r^{max}_{i,j,k}, w.r^{min}_{i,j,k}, w.r^{max}_{i,j,k})$ then
24:                 return false
25:     return true
26: function RELEASE_TIMES_CONDITION$(EFT^{max}, v.r^{min}_{i,j,k}, v.r^{max}_{i,j,k}, w.r^{min}_{i,j,k}, w.r^{max}_{i,j,k})$
27:     if $v.r^{max}_{i,j,k} = w.r^{max}_{i,j,k}$ then
28:         if $v.r^{min}_{i,j,k} \leq EFT^{max} \wedge w.r^{min}_{i,j,k} \leq EFT^{max}$ then
29:             return true
30:     else
31:         if $v.r^{max}_{i,j,k} \leq EFT^{max} \wedge w.r^{max}_{i,j,k} \leq EFT^{max}$ then
32:             return true
33:     return false
```

unschedulable by the worst-case approach, then the instance is indeed unschedulable, and, therefore, the generation of the SAG for such instance is needless.

If an instance is not deemed unschedulable by the worst-case approach, then the SAG generation starts with initializing a root vertex v_r. The initialization of v_r can be seen on lines 4-9. The first layer of vertices V_0 consists of the root vertex. The algorithm then keeps alternating between the expansion phase and the merge phase until there are no unfinished jobs left or until a deadline miss occurs. Each expansion phase generates a new set of vertices $V^{ex}_{\iota+1}$. This set is then processed by the merge phase resulting in a set of vertices $V_{\iota+1}$ which is then used as the input of the next expansion phase. Note that even though not stated explicitly, both the expansion phase and the merge phase have access to the inputs of Algorithm 6, i.e., to $\mathcal{E}$ and m.

If a deadline miss occurs after the expansion phase at any point of the SAG generation, then the algorithm concludes that the instance is unschedulable. If no deadline miss occurs during the entire SAG generation procedure, then the algorithm concludes that the instance is schedulable.

Algorithm 6 Complete SAG generation

Input: set of ET tasks $\mathcal{E}$, number of processors m
Output: true if the set of ET tasks $\mathcal{E}$ is schedulable under policy $\mathcal{P}$, false otherwise

```
function GENERATE_SAG($\mathcal{E}, m$)
    if ¬WORST_CASE_ANALYSIS($\mathcal{E}, m$) then
        return false    ▷ Deadline miss occurred while conducting worst-case schedulability analysis
    $v_r \leftarrow$ new vertex with $v_r.E^F \leftarrow \emptyset, v_r.in \leftarrow \emptyset, v_r.out \leftarrow \emptyset$
    $v_r.E^A \leftarrow$ applicable jobs based on $v_r.E^F$
    for $\pi = 1$ to $m$ do
        $v_r.EFT_\pi \leftarrow 0$
        $v_r.LFT_\pi \leftarrow 0$
    $v_r.FTI \leftarrow \{[v_r.EFT_1, v_r.LFT_1], \dots, [v_r.EFT_m, v_r.LFT_m]\}$
    $V_0 \leftarrow \{v_r\}$
    for $\iota = 0$ to $|\mathcal{E}| - 1$ do    ▷ $|\mathcal{E}|$ is the total number of ET jobs in $\mathcal{E}$
        $V^{ex}_{\iota+1} \leftarrow$ EXPANSION_PHASE($V_\iota$)
        for each $v \in V^{ex}_{\iota+1}$ do
            $E_{i,j,k} \leftarrow e.J$ where $v.in = \{e\}$    ▷ $e$ is the only edge in $v.in$
            if $v.LFT_{\sigma_{i,k}} > d^{ET}_{i,j,k}$ then
                return false    ▷ Deadline miss
        MERGE_PHASE($V^{ex}_{\iota+1}$)
        $V_{\iota+1} \leftarrow V^{ex}_{\iota+1}$
    return true    ▷ No deadline miss occurred during SAG generation
```

3.3.8 Discussion and example

The proposed SAG approach to schedulability analysis is unfortunately not exact. However, it provides us a sufficient schedulability analysis, i.e., it may deem some instances unschedulable even though they are in fact schedulable. Therefore, this approach is safe since it never deems an instance schedulable when the instance is in fact unschedulable, which would be a critical flaw.

The cause of the inexactness stems from the following fact. When a new vertex w is generated by expanding a vertex v with a job $E_{i,j,k}$ that has a direct successor $E_{i,j,k+1}$ in the precedence chain, then the earliest and the latest release time of $E_{i,j,k+1}$ in the vertex w are considered to be equal to the earliest and the latest time when $E_{i,j,k}$ may finish its execution, respectively. This allows for scalability of the SAG approach to schedulability analysis, however, it also causes the schedulability analysis to be pessimistic, i.e., the schedulability analysis accounts for some impossible execution scenarios as well.

The following example shows the manifestation of pessimism in the SAG approach to schedulability analysis. Let us consider set of 2 ET tasks $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2\}$, 2 processors, and the FP-EDF policy. The parameters of the ET tasks can be seen in Table 3.2. Given the parameters of the ET tasks, we can express $\mathcal{E}$ as a set of ET jobs. The parameters of the ET jobs can be seen in Table 3.3. $\mathcal{E}$ is in fact schedulable under the FP-EDF policy, which can be checked using the exact brute force approach, however, the SAG approach deems $\mathcal{E}$ unschedulable under the FP-EDF policy. The entire generated SAG can be seen in Figure 3.1. A detailed description of the SAG generation follows.

Since the worst-case approach deems the example instance schedulable, the SAG generation

ET task	τ^{ET}	d^{ET}	r^{min}	r^{max}	c^{min}	c^{max}	p	σ	h	l
$\mathcal{E}_1$	6	6	1	1	1	1	1	$\{1,2\}$	1	2
$\mathcal{E}_2$	6	4	0	0	1	2	2	$\{1,2\}$	1	2

Table 3.2: Parameters of ET tasks used in the SAG generation example, $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2\}$.

ET job	d^{ET}	r^{min}	r^{max}	c^{min}	c^{max}	p	σ
$E_{1,1,1}$	5	1	1	1	1	1	1
$E_{1,1,2}$	6	-	-	1	1	1	2
$E_{2,1,1}$	2	0	0	1	2	2	1
$E_{2,1,2}$	4	-	-	1	2	2	2

Table 3.3: Parameters of ET jobs used in the SAG generation example, $\mathcal{E} = \{\{\{E_{1,1,1}, E_{1,1,2}\}\}, \{\{E_{2,1,1}, E_{2,1,2}\}\}\}$.

procedure starts by initializing the root vertex v_r with no jobs finished. Therefore, the set of applicable jobs in v_r consists of jobs $E_{1,1,1}$ and $E_{2,1,1}$, which are both dedicated to processor 1. Finish time intervals of both processors 1 and 2 are initialized to $[0, 0]$. The expansion phase on $V_0 = \{v_r\}$ considers inspecting only time $t = 0$ since job $E_{2,1,1}$ is certainly released at $t = 0$ and processor 1 is unoccupied at that time. The expansion phase concludes that the FP-EDF policy would pick $E_{2,1,1}$ at $t = 0$ since $E_{1,1,1}$ is not released at that time in any execution scenario. Therefore, a new vertex v_1 is created with finish time interval of processor 1 updated as $[v_1.EFT_1, v_1.LFT_1] = \left[0 + c^{min}_{2,1,1}, 0 + c^{max}_{2,1,1}\right] = [1, 2]$. Moreover, job $E_{2,1,2}$, which is a successor of $E_{2,1,1}$ in the precedence chain, is added to the set of applicable jobs in vertex v_1 with $v_1.r^{min}_{2,1,2} = v_1.EFT_1 = 1$ and $v_1.r^{max}_{2,1,2} = v_1.LFT_1 = 2$. Additionally, a new edge emanating from v_r and incident to v_1 is added to the SAG with the label corresponding to the picked job $E_{2,1,1}$. This concludes the expansion phase on V_0. The merge phase is executed next but it does not change the SAG in any way since the expansion phase resulted in generating only one new vertex.

The algorithm continues with expansion phase on $V_1 = \{v_1\}$. When expanding v_1, all times from a time window $[1, 2]$ have to be inspected. Note that $E_{1,1,1}$ is dedicated to processor 1, whereas $E_{2,1,2}$ is dedicated to processor 2. At $t = 1$, $E_{1,1,1}$ is certainly released and it would be picked by the FP-EDF policy if the processor 1 would not be occupied at that time since there is no other released job dedicated to processor 1. If $E_{2,1,2}$ releases at $t = 1$, then it would also be picked by the FP-EDF policy since processor 2 is unoccupied at that time and there is no other released job dedicated to processor 2. At $t = 2$, the processor 1 becomes unoccupied and $E_{1,1,1}$ is picked by the FP-EDF policy. Simultaneously, $E_{2,1,2}$ is certainly released at $t = 2$, and therefore, $E_{2,1,2}$ is also picked by the FP-EDF policy since it is dedicated to a different processor than $E_{1,1,1}$. To summarize, the FP-EDF policy always picks both $E_{1,1,1}$ and $E_{2,1,2}$ in time window $[1, 2]$. In other words, both $E_{1,1,1}$ and $E_{2,1,2}$ may start their execution as early as $t = 1$ and as late as $t = 2$. This results in creating two new vertices v_2 and v_3 where $[v_2.EFT_1, v_2.LFT_1] = \left[1 + c^{min}_{1,1,1}, 2 + c^{max}_{1,1,1}\right] = [2, 3]$ and $[v_3.EFT_2, v_3.LFT_2] = \left[1 + c^{min}_{2,1,2}, 2 + c^{max}_{2,1,2}\right] = [2, 4]$. Moreover, job $E_{1,1,2}$, which is a successor of $E_{1,1,1}$ in the precedence chain, is added to the set of applicable jobs in vertex v_2 with $v_2.r^{min}_{1,1,2} = v_2.EFT_1 = 2$ and $v_2.r^{max}_{1,1,2} = v_2.LFT_1 = 3$. This concludes the expansion phase.

The merge phase does not merge vertices v_2 and v_3 since $v_2.E^F \neq v_3.E^F$.

The expansion phase on $V_2 = \{v_2, v_3\}$ individually expands both v_2 and v_3. When expanding v_2, all times from a time window $[1, 2]$ have to be inspected since time $t = 2$ is the earliest time when there is a certainly released job dedicated to an unoccupied processor, namely $E_{2,1,2}$. The set of applicable jobs in v_2 consists of jobs $E_{1,1,2}$ and $E_{2,1,2}$, which are both dedicated to processor 2. If $E_{2,1,2}$ releases at $t = 1$, then it would be picked by the FP-EDF policy since there is no other released job dedicated to processor 2. At $t = 2$, there are two different possible execution scenarios. If $E_{1,1,2}$ releases at $t = 2$, then the FP-EDF policy would pick $E_{1,1,2}$ since it has higher priority than $E_{2,1,2}$, which is certainly released at that time. If $E_{1,1,2}$ does not release at $t = 2$, then $E_{2,1,2}$ would be picked by the FP-EDF policy instead. This results in creating two new vertices v_4 and v_5 where $[v_4.EFT_2, v_4.LFT_2] = \left[2 + c^{min}_{1,1,2}, 2 + c^{max}_{1,1,2}\right] = [3, 3]$ and $[v_5.EFT_2, v_5.LFT_2] = \left[1 + c^{min}_{2,1,2}, 2 + c^{max}_{2,1,2}\right] = [2, 4]$. When expanding v_3, all times from a time window $[1, 2]$ have to be inspected since $E_{1,1,1}$ is certainly released at $t = 1$, however, $E_{1,1,1}$ is dedicated to processor 1 which becomes unoccupied at $t = 2$. $E_{1,1,1}$ would be picked by the FP-EDF policy at both times $t = 1$ and $t = 2$ since there is no other released job dedicated to processor 1 at both $t = 1$ and $t = 2$. Therefore, a new vertex w is created where $[w.EFT_1, w.LFT_1] = \left[1 + c^{min}_{1,1,1}, 2 + c^{max}_{1,1,1}\right] = [2, 3]$. Moreover, job $E_{1,1,2}$, which is a successor of $E_{1,1,1}$ in the precedence chain, is added to the set of applicable jobs in vertex w with $w.r^{min}_{1,1,2} = w.EFT_1 = 2$ and $w.r^{max}_{1,1,2} = w.LFT_1 = 3$. This concludes the expansion phase on V_2. The merge phase merges vertices v_5 and w since $v_5.E^F = w.E^F$, $[v_5.EFT_1, v_5.LFT_1] \cap [w.EFT_1, w.LFT_1] \neq \emptyset$, $[v_5.EFT_2, v_5.LFT_2] \cap [w.EFT_2, w.LFT_2] \neq \emptyset$, $v_5.r^{min}_{1,1,2} = w.r^{min}_{1,1,2}$, and $v_5.r^{max}_{1,1,2} = w.r^{max}_{1,1,2}$. The merge is done by redirecting the edge emanating from v_3 so that it becomes incident to v_5. In this case, the parameters of v_5 do not change in any way as a result of the merge since $v_5.r^{min}_{1,1,2} = w.r^{min}_{1,1,2} = 2$, $v_5.r^{max}_{1,1,2} = w.r^{max}_{1,1,2} = 3$, $\min(v_5.EFT_1, w.EFT_1) = \min(2, 2) = 2 = v_5.EFT_1$, $\max(v_5.LFT_1, w.LFT_1) = \max(3, 3) = 3 = v_5.LFT_1$, $\min(v_5.EFT_2, w.EFT_2) = \min(2, 2) = 2 = v_5.EFT_2$, $\max(v_5.LFT_2, w.LFT_2) = \max(4, 4) = 4 = v_5.LFT_2$. The vertex w gets removed from the SAG afterward, which is illustrated by using light gray color for w in Figure 3.1.

The final iteration of the expansion phase starts on $V_3 = \{v_4, v_5\}$. The expansion phase individually expands both v_4 and v_5. There is only one unfinished job for both v_4 and v_5. When expanding v_4, we see that $E_{2,1,2}$ is certainly released at time $t = 2$, however, processor 2, to which $E_{2,1,2}$ is dedicated, is certainly occupied until $t = 3$ when it becomes unoccupied. Therefore, $E_{2,1,2}$ is picked by the FP-EDF policy at $t = 3$. This results in creating a new vertex v_6 where $[v_6.EFT_2, v_6.LFT_2] = \left[3 + c^{min}_{2,1,2}, 3 + c^{max}_{2,1,2}\right] = [4, 5]$. When expanding v_5, all times from a time window $[2, 4]$ have to be inspected since $E_{1,1,2}$, which is dedicated to processor 2, may release as early as $t = 2$, however, processor 2 may be occupied until $t = 4$. If $E_{1,1,2}$ is released at $t = 2$ and if processor 2 is not occupied at that time, then $E_{1,1,2}$ is picked by the FP-EDF policy. The same holds for time $t = 3$. At $t = 4$, processor 2 is unoccupied and $E_{1,1,2}$ is certainly released, and therefore, $E_{1,1,2}$ is picked by the FP-EDF policy. To summarize, the FP-EDF policy always picks $E_{1,1,2}$ in time window $[2, 4]$. This results in creating a new vertex v_7 where $[v_7.EFT_2, v_7.LFT_2] = \left[2 + c^{min}_{1,1,2}, 4 + c^{max}_{1,1,2}\right] = [3, 5]$. This concludes the expansion phase on V_3. At this point, it is found out that a deadline miss occurred when v_6 was created since $v_6.LFT_2 = 5 > 4 = d^{ET}_{2,1,2}$. This is illustrated by using red color for $v_6.LFT_2$ in Figure 3.1. Therefore, the SAG generation procedure terminates with $V^{ex}_4 = \{v_6, v_7\}$, i.e., without carrying out the final iteration of the merge phase. To summarize, the SAG approach to schedulability analysis results in deeming the example instance unschedulable since a deadline miss occurred during the SAG generation.

Such a result is a consequence of considering impossible execution scenarios during the SAG generation. Vertex v_2 in Figure 3.1 states that $E_{1,1,2}$ may release in time interval $[2, 3]$ and that $E_{2,1,2}$ may release in time interval $[1, 2]$. However, it is impossible for both $E_{1,1,2}$ and $E_{2,1,2}$ to release simultaneously at time $t = 2$. This stems from the scheduling of $E_{1,1,1}$ and $E_{2,1,1}$ which are the predecessors of $E_{1,1,2}$ and $E_{2,1,2}$ in the precedence chain, respectively. $E_{2,1,1}$ always starts its execution at $t = 0$. If $E_{2,1,1}$ finishes its execution at $t = 1$, then $E_{2,1,2}$ releases at $t = 1$, $E_{1,1,1}$ starts its execution at $t = 1$, finishes its execution at $t = 2$, and $E_{1,1,2}$ releases at $t = 2$. If $E_{2,1,1}$ finishes its execution at $t = 2$, then $E_{2,1,2}$ releases at $t = 2$, $E_{1,1,1}$ starts its execution at $t = 2$, finishes its execution at $t = 3$, and $E_{1,1,2}$ releases at $t = 3$. These are the only two possibilities of how $E_{1,1,1}$ and $E_{2,1,1}$ can be scheduled using the FP-EDF policy.

To summarize, considering $E_{i,j,k+1}$ to be released within a time interval which is equivalent to the time interval when $E_{i,j,k}$ may finish its execution is the root cause of pessimism in the SAG approach to schedulability analysis. On the other hand, it allows for the scalability of the SAG approach. We have yet to come up with a scalable solution that would provide exact schedulability analysis in the dedicated multiprocessor setting. The amount of pessimism in the SAG approach is evaluated in Section 4.3.

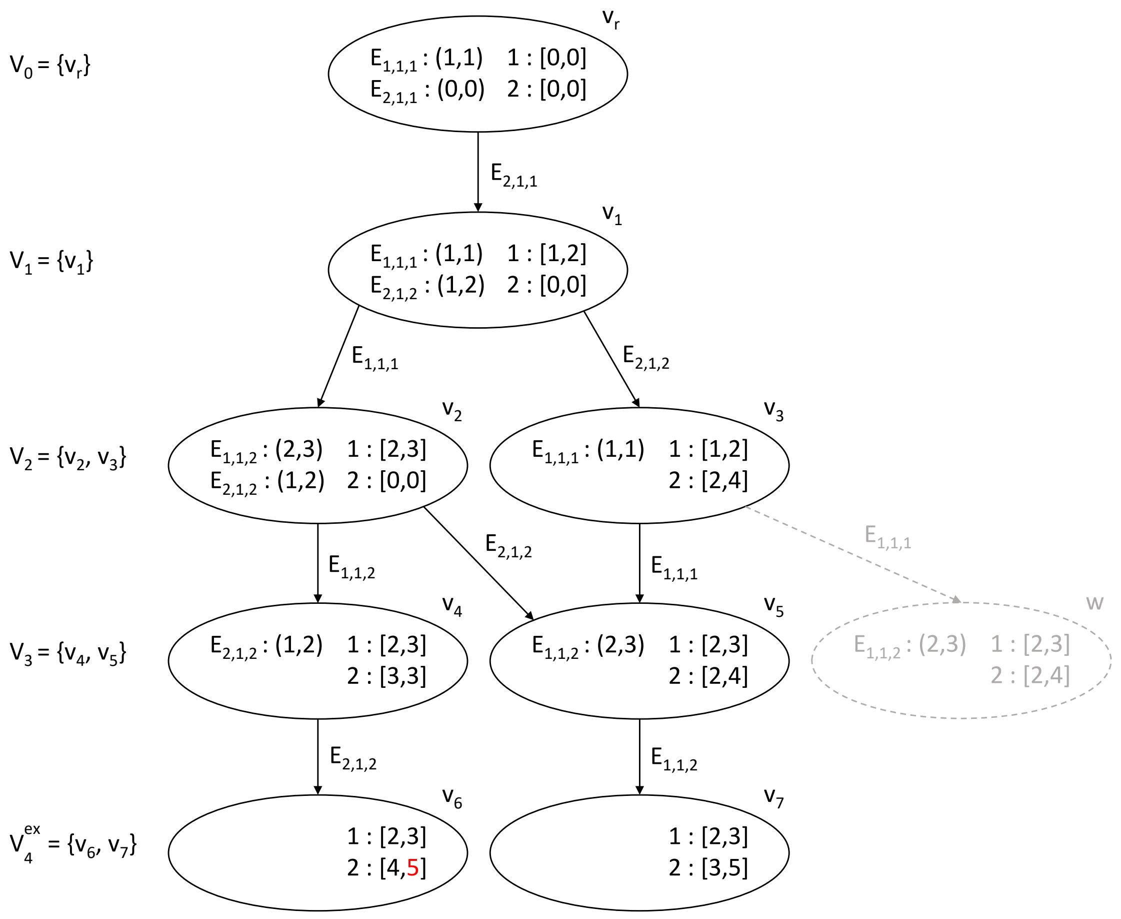
v_r
$V_0 = \{v_r\}$
$E_{1,1,1}$: (1,1) 1 : [0,0]
$E_{2,1,1}$: (0,0) 2 : [0,0]
$E_{2,1,1}$
v_1
$V_1 = \{v_1\}$
$E_{1,1,1}$: (1,1) 1 : [1,2]
$E_{2,1,2}$: (1,2) 2 : [0,0]
$E_{1,1,1}$
$E_{2,1,2}$
v_2
v_3
$V_2 = \{v_2, v_3\}$
$E_{1,1,2}$: (2,3) 1 : [2,3]
$E_{2,1,2}$: (1,2) 2 : [0,0]
$E_{1,1,1}$: (1,1) 1 : [1,2]
2 : [2,4]
$E_{1,1,2}$
$E_{2,1,2}$
$E_{1,1,1}$
$E_{1,1,1}$
v_4
v_5
w
$V_3 = \{v_4, v_5\}$
$E_{2,1,2}$: (1,2) 1 : [2,3]
2 : [3,3]
$E_{1,1,2}$: (2,3) 1 : [2,3]
2 : [2,4]
$E_{1,1,2}$: (2,3) 1 : [2,3]
2 : [2,4]
$E_{2,1,2}$
$E_{1,1,2}$
v_6
v_7
$V_4^{ex} = \{v_6, v_7\}$
1 : [2,3]
2 : [4,5]
1 : [2,3]
2 : [3,5]

Figure 3.1: The example of the generated SAG. The inside of each vertex v is divided into two columns. The left column contains all applicable jobs in the vertex and their respective earliest and latest release time in the vertex, which is denoted as $E_{i,j,k} : \left(v.r_{i,j,k}^{min}, v.r_{i,j,k}^{max}\right)$. The right column contains the finish time intervals of all processors in the vertex. The finish time interval of a processor π is denoted as $\pi : [v.EFT_\pi, v.LFT_\pi]$. Each level of the SAG is labeled with a set of vertices that constitute the corresponding level. Each edge is labeled with a scheduled job.

Chapter 4

ET solution evaluation

This chapter discusses the evaluation of the proposed approaches to schedulability analysis of a set of ET tasks. The evaluation is done empirically, i.e., the results are derived from experiments and measurements conducted on datasets of randomly generated instances, which are publicly available on GitHub[1]. The proposed algorithms from Chapter 3 are implemented in C++. The implementation supports the C++20 standard. The experiments were run on a system with 2x Intel® Xeon® E5-2690 v4 CPU, 14 Cores/CPU; 2.6GHz; 35 MB SmartCache; with 256 GB DDR4, ECC. Each instance was run on a single core and the memory available to each instance was restricted to 8 GB. The time limit was set to 10 minutes.

4.1 Model of the multiprocessor platform

The experiments were conducted for one specific application of our problem. The application is inspired by considering ET scheduling in networks such as Ethernet TSN [5]. The network model is similar to the model described in [21]. The network is modeled as a connected directed graph $\mathcal{G}^{NET} = \left(\mathcal{V}^{NET}, \mathcal{E}^{NET}\right)$. $\mathcal{V}^{NET}$ is a set of nodes in the network and $\mathcal{E}^{NET}$ is a set of links connecting the nodes. We consider a full-duplex network for the experiments. This means that, for two different nodes $v, w \in \mathcal{V}^{NET}$, there is a link $(v, w) \in \mathcal{E}^{NET}$ iff there is a link $(w, v) \in \mathcal{E}^{NET}$. The ET tasks correspond to data streams being transmitted from a node $v \in \mathcal{V}^{NET}$ to a node $w \in \mathcal{V}^{NET}$. The routing is done by finding the shortest path in the network. Each link in our network model can transmit data of at most one data stream at a time. Therefore, each link in the network model corresponds to a processor in our formal definition of the multiprocessor platform.

The network model is utilized to determine σ_i for each ET task $\mathcal{E}_i$. An example of the network model can be seen in Figure 4.1. To demonstrate how σ_i is determined, let us consider the following example for the network model in Figure 4.1. Let us assume that sending data from node v to node w corresponds to ET task $\mathcal{E}_1$. The shortest path from v to w contains links 7, 3, and 5 in this order. Therefore, $\sigma_1 = (7, 3, 5)$.

Note that a tree topology is considered for the network model in this work. This means that there is always exactly one path connecting two different nodes $v, w \in \mathcal{V}^{NET}$ such that the path does not visit any link and any node more than once.

[1]`https://github.com/halasluk/ETTT_dedicated_multicore`

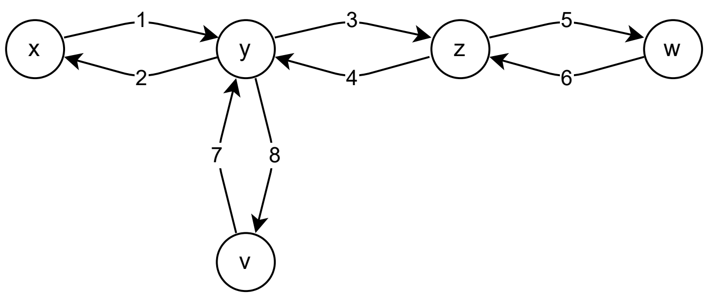

Figure 4.1: Example of the network model. The edges correspond to the processors. Therefore, they are labeled with a unique index $\pi \in \{1, \dots, 8\}$.

4.2 Instance generation

The instances were generated randomly using the following procedure. Note that the randomly generated values are assumed to be generated uniformly at random. The procedure utilizes the following input parameters:

- **Number of nodes** $\left|\mathcal{V}^{NET}\right|$ – Total number of nodes in the network model.
- **Number of ET tasks** n – Total number of ET tasks of the instance.
- **Target hyperperiod** A^{η} – Maximum possible hyperperiod of the instance.
- **Minimum period** A^{τ} – Minimum possible period of a task.
- **Maximum utilization** U – Maximum utilization of each processor of the instance.
- **Maximum release time shift percentage** A^{r} – The maximum amount of shift of release time from the beginning of a period for each task.
- **Maximum deadline shift percentage** A^{d} – The maximum amount of shift of deadline from the end of a period for each task.
- **Maximum release jitter percentage** A^{j} – The maximum amount of release jitter for each task.
- **Maximum execution time variation percentage** A^{c} – The maximum amount of execution time variation for each task.
- **Minimum priority** p_{min} – The highest priority a task can have.
- **Maximum priority** p_{max} – The lowest priority a task can have.

The procedure works as follows. First, the network model with a random tree topology is generated such that it has $\left|\mathcal{V}^{NET}\right|$ nodes. Then, for each of n ET tasks, two different nodes from the generated network model are chosen and the shortest path between them is found. This determines σ_i. Furthermore, we can determine which links in the network model are utilized by at least one task. This provides us with the number of processors m. The period τ_i^{ET} is chosen randomly such that $A^{\tau} \leq \tau_i^{ET}$ and A^{η} is divisible by τ_i^{ET}.

When generating c_i^{max}, the aim is that the utilization of each processor is at most equal to U. Therefore, we define

$$U = \max_{\pi \in \{1,\dots,m\}} U_\pi,$$

where U_π is the utilization of processor π, which is calculated as

$$U_\pi = \sum_{\{\mathcal{E}_i \in \mathcal{E} | \pi \in \sigma_i\}} \frac{c_i^{max}}{\tau_i^{ET}}.$$

The latest release time r_i^{max} is generated randomly such that

$$\frac{r_i^{max}}{\tau_i^{ET}} \leq A^r.$$

Then, the deadline d_i^{ET} is generated randomly such that

$$\frac{\tau_i^{ET} - d_i^{ET}}{\tau_i^{ET}} \leq A^d.$$

Note that the procedure also ensures that $r_i^{max} + l_i \cdot c_i^{max} \leq d_i^{ET} \leq \tau_i^{ET}$ holds. We can observe that A^r and A^d restrict the time window when the jobs of a task's occurrence may execute without missing their deadlines. This time window will be referred to as slack time. The higher A^r and A^d, the more restricted slack time.

Afterward, r_i^{min} is generated randomly such that

$$\frac{r_i^{max} - r_i^{min}}{r_i^{max}} \leq A^j,$$

for $r_i^{max} > 0$. If $r_i^{max} = 0$, then $r_i^{min} = 0$. Similar to r_i^{min}, c_i^{min} is generated randomly such that

$$\frac{c_i^{max} - c_i^{min}}{c_i^{max} - 1} \leq A^c,$$

for $c_i^{max} > 1$. If $c_i^{max} = 1$, then $c_i^{min} = 1$. We can observe that A^j and A^c influence the number of execution scenarios for the instance. The higher A^j and A^c, the higher the number of execution scenarios.

Finally, for a task $\mathcal{E}_i$, the priority p_i is generated randomly such that $p_{min} \leq p_i \leq p_{max}$. Note that the values of all parameters that represent percentages, i.e., U, A^r, A^d, A^j, and A^c, can be set to any real number from a closed interval $[0, 1]$.

4.2.1 Generated datasets

First, one dataset $\mathcal{D}_0^a$ was generated for the experiments in Section 4.3. $\mathcal{D}_0^a$ contains 10000 instances. The generation of the instances slightly differed from what is described in Section 4.2. The reason for this slight change is thoroughly explained in Section 4.3. The instances were generated with the following parameters: $n = 4$, $A^\eta = 12$, $A^\tau = 6$, $U = 0.3$, $A^r = 1$, $A^d = 1$, $p_{min} = 1$, $p_{max} = 4$. Note that $\left|\mathcal{V}^{NET}\right|$ was not provided, therefore, no network model was generated. Instead, we manually set $\sigma_1 = (1)$, $\sigma_2 = (1, 2, 3)$, $\sigma_3 = (2)$, $\sigma_4 = (2, 3)$, and, therefore, $m = 3$. Moreover, A^j and A^c were not provided. Instead, the instances were generated such that $r_i^{max} - r_i^{min} \leq 1$ and $c_i^{max} - c_i^{min} \leq 1$, for each ET task $\mathcal{E}_i$.

Then, we generated 9 datasets for the experiments in Section 4.4. These datasets were generated according to the procedure described in Section 4.2. Each of these datasets contains 1000 instances. For the generation of these 9 datasets, a single network model was generated with $\left|\mathcal{V}^{NET}\right| = 10$. Each of these datasets was then generated with the following parameters: $1 \leq n \leq 20$, $A^\eta = 10^7$, $A^\tau = 10^6$, $U = 0.3$, $p_{min} = 1$, $p_{max} = 5$. Furthermore, each of these datasets was generated with a different combination of values of parameters A^r, A^d, A^j, and A^c. The combination of values of these parameters for each dataset can be seen in Table 4.1, e.g., dataset $\mathcal{D}_6^a$ was generated with $A^r = A^d = A^j = A^c = 0.3$.

	$A^r = A^d = 0.1$	$A^r = A^d = 0.2$	$A^r = A^d = 0.3$
$A^j = A^c = 0$	$\mathcal{D}_1^a$	$\mathcal{D}_2^a$	$\mathcal{D}_3^a$
$A^j = A^c = 0.3$	$\mathcal{D}_4^a$	$\mathcal{D}_5^a$	$\mathcal{D}_6^a$
$A^j = A^c = 0.6$	$\mathcal{D}_7^a$	$\mathcal{D}_8^a$	$\mathcal{D}_9^a$

Table 4.1: Values of parameters A^r, A^d, A^j, and A^c used for generation of datasets $\mathcal{D}_1^a, \dots, \mathcal{D}_9^a$.

Datasets $\mathcal{D}_1^a, \dots, \mathcal{D}_9^a$ contain arguably harder instances than dataset $\mathcal{D}_0^a$ since the instances consist of up to 20 ET tasks and since the target hyperperiod is nearly 10^6 times larger than the target hyperperiod of the instances from $\mathcal{D}_0^a$.

4.3 Evaluation of pessimism in the SAG approach and evaluation of the worst-case approach impact

To evaluate the amount of pessimism in the SAG approach, we needed to generate a dataset of sufficiently small instances such that the exact brute force approach can verify the schedulability of each instance in a reasonable amount of time. We struggled to find a dataset such that there is at least one instance that is schedulable according to the brute force approach but is deemed unschedulable by the SAG approach. This already suggested that the manifestation of pessimism in the SAG approach is quite rare. Ultimately, we generated dataset $\mathcal{D}_0^a$. The results of the schedulability analysis of the instances from $\mathcal{D}_0^a$ can be seen in Table 4.2. We can observe that the false negative rate of the SAG approach is approximately only 0.27 %, i.e., the SAG approach deems an instance unschedulable even though it is in fact schedulable in 0.27 % of the instances. Therefore, the true positive rate of the SAG approach is approximately 99.73 %, i.e., the SAG approach deems an instance schedulable when it is indeed schedulable in 99.73 % of the instances.

Moreover, we evaluate the worst-case approach to schedulability analysis in this section. Recall that the worst-case approach is used as a speedup in the SAG approach. When the worst-case approach deems the instance unschedulable, it is indeed unschedulable. Therefore, we evaluate the true negative rate of the worst-case approach. The results of the worst-case approach to schedulability analysis can be seen in Table 4.2. We can observe that the worst-case approach deems an instance unschedulable when it is indeed unschedulable in approximately 81.54 % of the instances.

	All instances	BF schedulable only	BF unschedulable only
BF schedulable	5866	5866	0
BF unschedulable	4134	0	4134
SAG schedulable	5850	5850	0
SAG unschedulable	4150	16	4134
WC schedulable	6629	5866	763
WC unschedulable	3371	0	3371

Table 4.2: Results of the schedulability analysis of $\mathcal{D}_0^a$ using the brute force (BF) approach, the SAG approach, and the worst-case (WC) approach.

4.4 Evaluation of schedulability and runtime of the SAG approach

The runtime of the SAG approach was measured for datasets $\mathcal{D}_1^a, \dots, \mathcal{D}_9^a$. Figure 4.2 shows the runtimes relative to the number of ET tasks of each instance for each dataset. Figure 4.3 shows the runtimes relative to the number of ET jobs of each instance for each dataset. Note that the layout of the subfigures in both Figure 4.2 and Figure 4.3 corresponds to the layout of datasets $\mathcal{D}_1^a, \dots, \mathcal{D}_9^a$ in Table 4.1 in Subsection 4.2.1, i.e., the runtimes for dataset $\mathcal{D}_1^a$, for example, are shown in the subfigure in the top left corner of Figure 4.2 and Figure 4.3.

We can observe that the instances from datasets with higher release jitter and execution time variation have longer runtimes. This is caused by the fact that the branching factor of the SAG is much higher for these instances. Therefore, the smallest instance from datasets $\mathcal{D}_7^a, \mathcal{D}_8^a, \mathcal{D}_9^a$ that reached the time limit consists of 63 ET jobs, whereas the smallest instance from datasets $\mathcal{D}_4^a, \mathcal{D}_5^a, \mathcal{D}_6^a$ that reached the time limit consists of 117 ET jobs. Recall that the time limit was set to 10 minutes. Note that the runtimes of instances from datasets $\mathcal{D}_1^a, \mathcal{D}_2^a, \mathcal{D}_3^a$ are quite low because these datasets were generated with no release jitter and no execution time variation. Moreover, the low runtimes of many unschedulable instances across all datasets are the consequence of the worst-case approach speedup. The largest instance with non-zero release jitter and execution time variation that was verified to be schedulable by the SAG approach to schedulability analysis consists of 350 ET jobs.

The schedulability of the instances from each dataset is shown in Table 4.3. Note that the layout of Table 4.3 corresponds to the layout of Table 4.1 in Subsection 4.2.1. Each cell in Table 4.3 that corresponds to a dataset contains the total number of schedulable instances, the total number of unschedulable instances, and the total number of instances that reached the time limit in this order. When we inspect Table 4.3 column-wise, we can observe that the datasets with the largest slack time, i.e., datasets $\mathcal{D}_1^a, \mathcal{D}_4^a, \mathcal{D}_7^a$, contain the maximal number of schedulable instances in their respective rows. For datasets with smaller slack time, the number of schedulable instances decreases, whereas the number of unschedulable instances increases. Moreover, the number of schedulable instances also decreases with the increase of release jitter and execution time variation which is caused by the increase in the number of instances that reached the time limit. To summarize, dataset $\mathcal{D}_1^a$ has the highest schedulability ratio of 82.4 %, whereas dataset $\mathcal{D}_9^a$ has the lowest schedulability ratio of approximately 49.9 %.

	$A^r = A^d = 0.1$	$A^r = A^d = 0.2$	$A^r = A^d = 0.3$
$A^j = A^c = 0$	824, 176, 0	781, 219, 0	694, 306, 0
$A^j = A^c = 0.3$	601, 224, 175	560, 276, 164	482, 344, 174
$A^j = A^c = 0.6$	449, 198, 353	399, 265, 336	358, 360, 282

Table 4.3: Schedulability of instances from datasets $\mathcal{D}_1^a, \ldots, \mathcal{D}_9^a$. Each cell corresponding to a dataset contains the total number of schedulable instances, the total number of unschedulable instances, and the total number of instances that reached the time limit in this order.

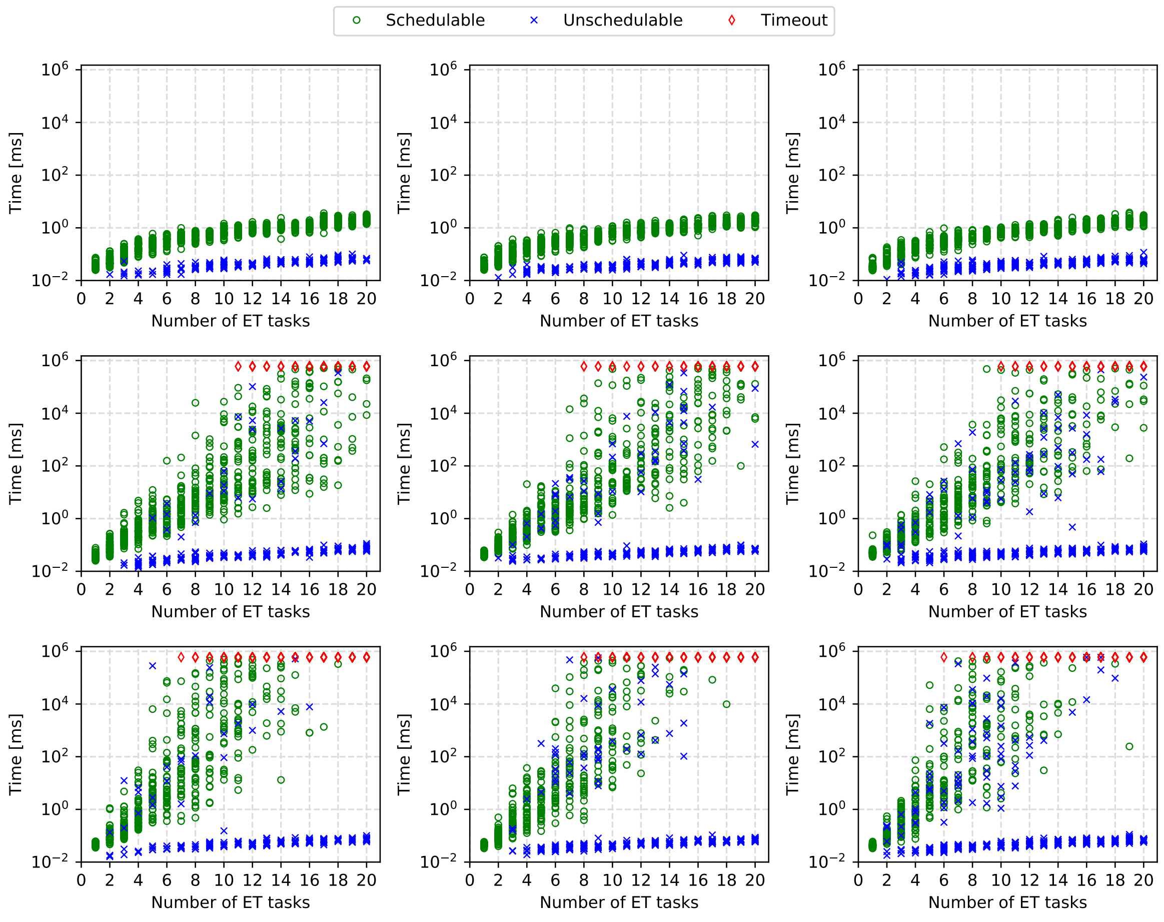

Figure 4.2: Runtimes of the SAG approach to schedulability analysis for datasets $\mathcal{D}_1^a, \ldots, \mathcal{D}_9^a$ relative to the number of ET tasks.

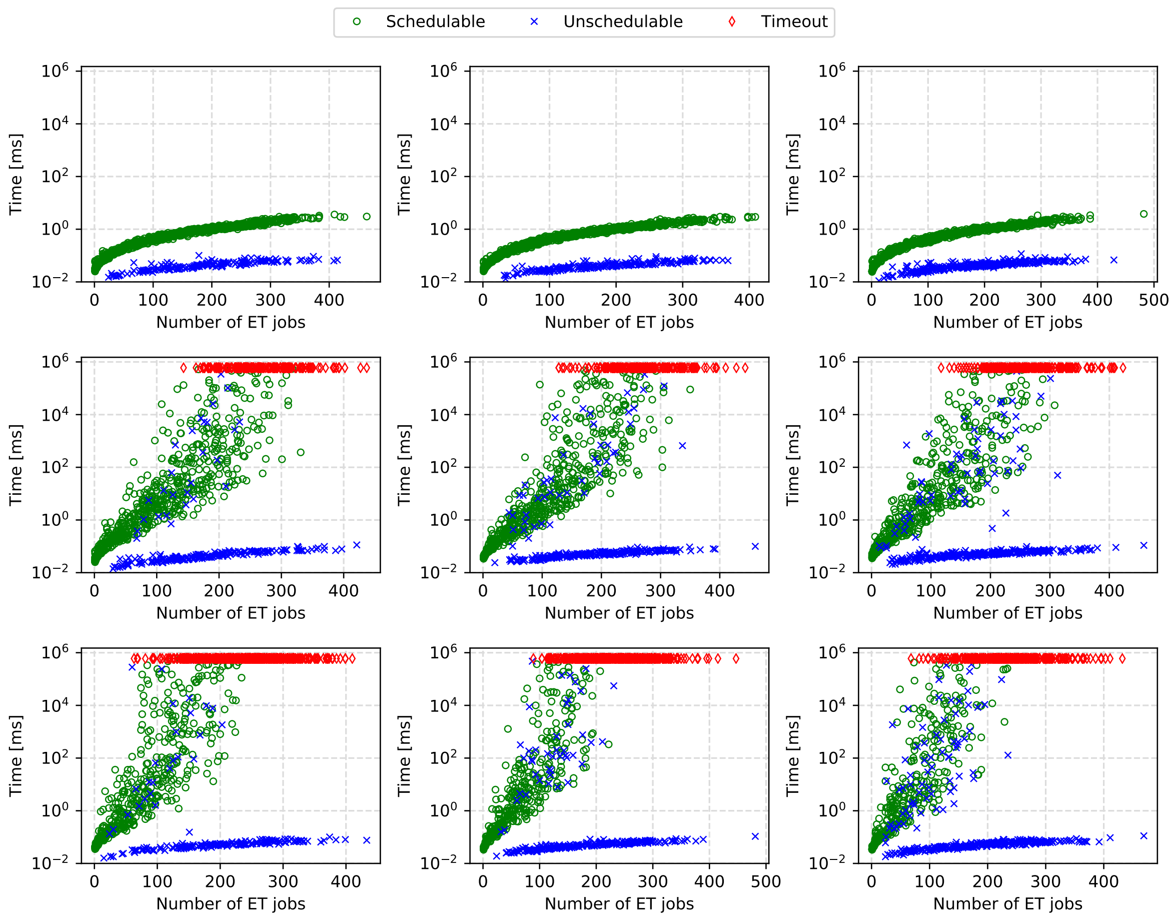

Figure 4.3: Runtimes of the SAG approach to schedulability analysis for datasets $\mathcal{D}_1^a, \ldots, \mathcal{D}_9^a$ relative to the number of ET jobs.

Chapter 5
ET+TT solutions

This chapter describes and thoroughly explains the algorithms used for scheduling the combination of ET and TT tasks in the dedicated multiprocessor setting. Given a set of ET tasks $\mathcal{E}$, a set of TT tasks $\mathcal{T}$, a number of processors m, and a scheduling policy $\mathcal{P}$, the main aim is to find a valid set of start times $\mathcal{S}$ for the TT jobs. As mentioned in Section 2.7, $\mathcal{S}$ is valid if $\mathcal{S}$ is feasible and if no execution scenario results in a deadline miss and if each TT job $T_{i,j,k}$ always starts its execution at its predetermined start time $s_{i,j,k}$ when the ET jobs are scheduled online.

First, a simple but non-scalable brute force algorithm is proposed in Section 5.1. The brute force algorithm is exact, i.e., the algorithm either finds a valid set of start times if it exists or proves that a valid set of start times does not exist. Section 5.2 describes the novel and scalable algorithm that is safe but pessimistic, i.e., if the algorithm finds a solution, then the solution is indeed a valid set of start times, however, the algorithm may not always be able to find a solution even if it exists. Note that this chapter makes use of OOP notation to express the parameters of objects.

5.1 Brute force algorithm

First, we need to describe a method that, for a set of ET tasks $\mathcal{E}$, a set of TT tasks $\mathcal{T}$, a number of processors m, a scheduling policy $\mathcal{P}$, and a set of start times $\mathcal{S}$, checks whether $\mathcal{S}$ is a valid set of start times. This can be done by transforming the TT tasks $\mathcal{T}$ into ET tasks and subsequently launching the brute force schedulability analysis of the combination of the transformed TT tasks and the ET tasks $\mathcal{E}$. Formally, TT job $T_{i,j,k}$ becomes fixed at start time $s_{i,j,k} \in \left[r^{TT}_{i,j,k}, d^{TT}_{i,j,k} - c^{TT}_{i,j,k}\right]$ when it is transformed into an ET job $E^f_{i,j,k}$ such that $E^f_{i,j,k}.r^{min} = E^f_{i,j,k}.r^{max} = s_{i,j,k}$, $E^f_{i,j,k}.c^{min} = E^f_{i,j,k}.c^{max} = c^{TT}_{i,j,k}$, $E^f_{i,j,k}.d^{ET} = s_{i,j,k} + c^{TT}_{i,j,k}$, $E^f_{i,j,k}.p = 0$, and $E^f_{i,j,k}$ is dedicated to processor $\sigma'_{i,k}$.

The set of TT tasks $\mathcal{T}$ becomes fixed when each TT job $T_{i,j,k}$ is fixed. The set of fixed TT tasks is denoted as $\mathcal{E}^f$. We can check whether, for the set of ET tasks $\mathcal{E}$, the set of TT tasks $\mathcal{T}$, the number of processors m, and the scheduling policy $\mathcal{P}$, a set of start times $\mathcal{S}$ is valid by launching the brute force schedulability analysis of the set $\mathcal{E} \cup \mathcal{E}^f$, where $\mathcal{E}^f$ was obtained using $\mathcal{S}$. The set of start times $\mathcal{S}$ is valid if the set $\mathcal{E} \cup \mathcal{E}^f$ is schedulable under policy $\mathcal{P}$, i.e., no execution scenario results in a deadline miss. Note that a constraint $p_{i,j,k} > 0$ is added for each ET job $E_{i,j,k} \in \mathcal{E}$ to make sure that the fixed TT jobs are always prioritized over the ET jobs when the FP-EDF policy is considered as policy $\mathcal{P}$.

The pseudocode of the brute force algorithm can be seen in Algorithm 7. The pseudocode consists of two functions. The brute force algorithm is launched by calling the function RUN_BRUTE_FORCE which simply initiates recursive fixing of the TT jobs. As the name suggests, the function FIX_RECURSIVELY finds all possible start times for each TT job recursively starting with $T_{1,1,1}$. This function also makes use of an auxiliary set $\mathcal{S}^{temp}$ that stores already-found start times. $\mathcal{S}^{temp}$ is initialized to $\emptyset$ within calling the function FIX_RECURSIVELY for the first time since no TT job has its start time assigned yet. Lines 13-18 ensure that the executions of the TT jobs dedicated to the same processor do not overlap. This is done by checking whether the execution of $T_{i,j,k}$ overlaps with the execution of any TT job dedicated to the same processor as $T_{i,j,k}$ that has its start time already assigned when considering a start time $s_{i,j,k}$ for $T_{i,j,k}$. Line 20 describes that a TT job $T_{i,j,k+1}$ is released when its direct predecessor in the precedence chain $T_{i,j,k}$ finishes its execution.

When $i > n'$, we know that a feasible set of start times $\mathcal{S}$ was found. To check whether $\mathcal{S}$ is valid, $\mathcal{E}^f$ is created using $\mathcal{S}$, and then the brute force schedulability analysis is launched on line 7 by calling the function BRUTE_FORCE_ANALYSIS, which is described in Algorithm 1 in Section 3.1, with inputs $\mathcal{E} \cup \mathcal{E}^f$ and m. Note that even though not stated explicitly, the function FIX_RECURSIVELY has access to all inputs of Algorithm 7, i.e., not only to $\mathcal{T}$ but also to both $\mathcal{E}$ and m. Moreover, it should be noted that when launching the function BRUTE_FORCE_ANALYSIS with its first input being $\mathcal{E} \cup \mathcal{E}^f$, then $E^f_{i,j,k}.r^{ET} = E^f_{i,j,k}.r^{min} = E^f_{i,j,k}.r^{max}$ is considered, i.e., $E^f_{i,j,k}.r^{ET}$ is considered to be revealed, $\forall E^f_{i,j,k} \in \mathcal{E}^f$. Therefore, $\forall E^f_{i,j,k} \in \mathcal{E}^f$, lines 26-27 in Algorithm 1 are ignored.

If, for the set $\mathcal{E} \cup \mathcal{E}^f$, no execution scenario results in a deadline miss, then the set of start times $\mathcal{S}$ is valid. In that case, Algorithm 7 terminates and returns $\mathcal{S}$ by propagating it through the recursive calls. Otherwise, $\emptyset$ is returned as can be seen on line 28 and the function FIX_RECURSIVELY continues to find a different feasible set of start times $\mathcal{S}$. If there exists no feasible set of start times $\mathcal{S}$ that is also valid, then Algorithm 7 terminates by returning $\emptyset$.

The main advantage of the brute force algorithm is that it is exact. The disadvantage of this algorithm is that it heavily suffers from combinatorial explosion. The runtime of the brute force algorithm depends on the number of execution scenarios $N^{\mathcal{E}}$, which is defined in Section 3.1, as well as on the number of feasible sets of start times. Given an instance of the combination of ET and TT tasks, we cannot easily compute how many feasible sets of start times there are. However, we can estimate that number with an upper bound. If we only consider that deadlines have to be met for all TT jobs, then the number of sets of start times that satisfy this condition can be expressed using the parameters of TT tasks in a closed form as

$$N^{\mathcal{T}} = \prod_{i=1}^{n'} \binom{d_i^{TT} - r_i^{TT} - l'_i \cdot (c_i^{TT} - 1)}{l'_i}^{h'_i},$$

where $d_i^{TT} - r_i^{TT} - l'_i \cdot \left(c_i^{TT} - 1\right) = d_i^{TT} - r_i^{TT} - l'_i \cdot c_i^{TT} + l'_i \geq l'_i$ holds thanks to the assumption that, for each TT task $\mathcal{T}_i \in \mathcal{T}$, $r_i^{TT} + l'_i \cdot c_i^{TT} \leq d_i^{TT} \Rightarrow d_i^{TT} - r_i^{TT} - l'_i \cdot c_i^{TT} \geq 0$. This means that the brute force schedulability analysis may be launched up to $N^{\mathcal{T}}$ times. Therefore, for the combination of ET and TT tasks, the total number of execution scenarios to be simulated is upper bounded by $N^{\mathcal{E}} \cdot N^{\mathcal{T}}$.

Let us now consider the following example of the combination of ET and TT tasks. First, for a set of ET tasks $\mathcal{E}$, let us consider the same set as in the example in Section 3.1 given

Algorithm 7 Brute force algorithm for ET+TT

```
Input: set of TT tasks $\mathcal{T}$, set of ET tasks $\mathcal{E}$, number of processors $m$
Output: valid set of start times $\mathcal{S}$ if it exists, $\emptyset$ otherwise

function RUN_BRUTE_FORCE($\mathcal{T}, \mathcal{E}, m$)
    return FIX_RECURSIVELY($\mathcal{T}, 1, 1, 1, \emptyset$)        ▷ Initial call to recursive fixation of TT jobs
function FIX_RECURSIVELY($\mathcal{T}, i, j, k, \mathcal{S}^{temp}$)
    if $i > n'$ then                                                ▷ $n'$ is the total number of TT tasks
        $\mathcal{S} \leftarrow \mathcal{S}^{temp}$
        $\mathcal{E}^f \leftarrow$ set of fixed TT tasks $\mathcal{T}$ using $\mathcal{S}$
        if BRUTE_FORCE_ANALYSIS($\mathcal{E} \cup \mathcal{E}^f, m$) then
            return $\mathcal{S}$
    else
        $\pi \leftarrow \sigma'_{i,k}$                  ▷ $\sigma'_{i,k}$ is a processor to which the TT job $T_{i,j,k}$ is dedicated
        for $s_{i,j,k} = r^{TT}_{i,j,k}$ to $(d^{TT}_{i,j,k} - c^{TT}_{i,j,k})$ do
            $overlap \leftarrow$ false
            for each TT job $T_{i',j',k'} \in T_\pi$ st. $s_{i',j',k'} \in \mathcal{S}^{temp}$ do ▷ $T_\pi$ is a set of all TT jobs dedicated to $\pi$
                if $[s_{i,j,k}, s_{i,j,k} + c^{TT}_{i,j,k}) \cap [s_{i',j',k'}, s_{i',j',k'} + c^{TT}_{i',j',k'}) \neq \emptyset$ then
                    $overlap \leftarrow$ true
                    break
            if $overlap$ then
                continue
            if $k + 1 \leq l'_i$ then
                $r^{TT}_{i,j,k+1} \leftarrow s_{i,j,k} + c^{TT}_{i,j,k}$      ▷ Release time of the TT job $T_{i,j,k+1}$ is revealed
                $\mathcal{S} \leftarrow$ FIX_RECURSIVELY($\mathcal{T}, i, j, k+1, \mathcal{S}^{temp} \cup \{s_{i,j,k}\}$)
            else if $j + 1 \leq h'_i$ then
                $\mathcal{S} \leftarrow$ FIX_RECURSIVELY($\mathcal{T}, i, j+1, 1, \mathcal{S}^{temp} \cup \{s_{i,j,k}\}$)
            else
                $\mathcal{S} \leftarrow$ FIX_RECURSIVELY($\mathcal{T}, i+1, 1, 1, \mathcal{S}^{temp} \cup \{s_{i,j,k}\}$)
            if $\mathcal{S} \neq \emptyset$ then
                return $\mathcal{S}$
    return $\emptyset$
```

by Table 3.1. For this $\mathcal{E}$, we computed that $N^{\mathcal{E}} = 10^{56}$. For $\mathcal{T}$, let us consider a set of 4 TT tasks given by Table 5.1.

TT task	d^{TT}	r^{TT}	c^{TT}	h'	l'
$\mathcal{T}_1$	54	27	12	1	2
$\mathcal{T}_2$	38	6	10	2	3
$\mathcal{T}_3$	20	9	4	4	2
$\mathcal{T}_4$	10	2	2	8	3

Table 5.1: Example of TT task parameters for calculation of $N^{\mathcal{T}}$, $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4\}$.

For $\mathcal{T}$ given by Table 5.1, we can calculate $N^{\mathcal{T}}$ as

$$N^{\mathcal{T}} = \binom{5}{2}^1 \cdot \binom{5}{3}^2 \cdot \binom{5}{2}^4 \cdot \binom{5}{3}^8 = 10^{15}.$$

This means that there are up to 10^{15} feasible sets of start times in this example. To summarize, for this particular combination of ET and TT tasks, the total number of execution scenarios to be simulated is upper bounded by $10^{56} \cdot 10^{15} = 10^{71}$.

5.2 Fixation Graph Generation algorithm

This section describes a scalable algorithm that, for the combination of ET and TT tasks in the dedicated multiprocessor setting, finds a valid set of start times for the TT jobs. This algorithm is an extension of the SAG generation procedure described in Section 3.3. It adds a new phase called fixation phase and the resulting graph of the algorithm is called fixation graph. Therefore, this algorithm is called Fixation Graph Generation (FGG). The terminology and the main ideas of the FGG algorithm are inspired by [14, 15].

First, the structure of the fixation graph will be described. Furthermore, a high-level description of the FGG will be provided. A low-level description of the FGG with pseudocode will be provided afterward. Finally, the sources of pessimism in this approach and possible improvements will be discussed.

5.2.1 Fixation graph structure

The structure of the fixation graph is very similar to that of the SAG. The fixation graph can be described as a directed acyclic graph. Similar to the SAG, the fixation graph starts with a root vertex v_r. Vertices and edges of the fixation graph inherit their parameters from vertices and edges of the SAG, respectively. Each edge in the fixation graph corresponds to either a single ET job or a single TT job. Therefore, the vertices of the fixation graph can be expanded by both ET and TT jobs. Expanding a vertex with a TT job will be called "fixing a TT job" to distinguish it from expanding a vertex with an ET job. A TT job is fixed after determining its start time.

So far we have considered one set of finished jobs in vertex v consisting of ET jobs. From now on, $v.E^F$ will be called the set of finished ET jobs in vertex v. Furthermore, in each vertex v of the fixation graph, we will distinguish between the set of finished ET jobs $v.E^F$ and the set of finished TT jobs $v.T^F$. The set $v.E^F$ is therefore redefined to consist only of ET jobs that are the labels of edges along any directed path from the root vertex v_r to v. The set $v.T^F$ is defined to consist only of TT jobs that are the labels of edges along any directed path from the root vertex v_r to v.

Furthermore, from now on, $v.E^A$ will be called the set of applicable ET jobs. In each vertex v of the fixation graph, we will distinguish between the set of applicable ET jobs $v.E^A$ and the set of applicable TT jobs $v.T^A$. The set $v.T^A$ is defined to consist of TT jobs $T_{i,j,k}$ that satisfy the following conditions:

$$T_{i,j,k} \text{ is unfinished } \wedge \Big((j = 1 \wedge k = 1) \vee \\ \big(j > 1 \wedge k = 1 \wedge T_{i,j-1,l'_i} \text{ is finished}\big) \vee (k > 1 \wedge T_{i,j,k-1} \text{ is finished}) \Big).$$

Note that this definition differs from the definition of applicable ET jobs only by substituting TT jobs for ET jobs. Moreover, note that $v.E^A$ and $v.T^A$ are computed from $v.E^F$ and $v.T^F$, respectively.

Similar to the SAG, the fixation graph is a directed level-structured graph, which means that its vertices can be divided into disjoint sets based on their distance from the root vertex v_r. However, the main difference between the SAG and the fixation graph is that the layers of vertices in the fixation graph are generated recursively. From now on, a layer of vertices will be denoted as V.

Moreover, when fixing a TT job, given a layer of vertices V, all vertices $v \in V$ get expanded by the TT job. Hence, we say that the fixation of a TT job is done on the entire layer of vertices V.

5.2.2 Description of the FGG

The FGG algorithm begins with initializing the root vertex v_r. In the root vertex, all ET and TT jobs are unfinished and the finish time interval of each processor is set to $[0, 0]$. The initial layer of vertices, therefore, consists only of v_r. Then, the main loop in the algorithm is launched on the initial layer of vertices. The main loop in the algorithm gradually performs three phases: the fixation phase, the expansion phase, and the merge phase. The fixation phase determines whether there are any applicable TT jobs on layer V that may be fixed by finding suitable start times for them. If there is a TT job that may be fixed on the layer V, then a new layer V^{fix} is generated by fixing that job, and the main loop is recursively relaunched on V^{fix}. If no TT job may be fixed on the layer V, then the expansion phase is performed upon V. This results in generating a new layer of vertices V^{ex}. The merge phase is then performed upon V^{ex}. Note that the expansion phase and the merge phase correspond to those described in the SAG generation. After obtaining the layer of merged vertices, the algorithm continues with the next iteration of the main loop on this layer. If a deadline miss occurs during the FGG, then the algorithm backtracks to the last layer of vertices when a TT job was fixed, i.e., it returns from the last recursive call and removes all vertices and edges that were added to the fixation graph after the last TT job fixation. If there is a different TT job that may be fixed, then the algorithm fixes it. Otherwise, the algorithm resumes the main loop and executes the expansion phase. The algorithm terminates when it generates a layer of vertices with no unfinished ET and TT jobs. In this case, the TT jobs are fixed in such a way that no execution scenario results in a deadline miss. Therefore, we obtain the valid set of start times $\mathcal{S}$ by gathering the start times of the fixed TT jobs. However, if all combinations of the fixed TT jobs considered by the FGG algorithm result in a deadline miss, then the algorithm terminates and returns $\emptyset$ since it was not able to find a valid set of start times $\mathcal{S}$.

The FGG algorithm uses a tabu list $\mathcal{L}$ to speed up the computation by pruning the search space. To put it simply, $\mathcal{L}$ stores all combinations of the fixed TT jobs and their start times that caused a deadline miss to occur during the generation of the fixation graph. Therefore, when fixing a TT job $T_{i,j,k}$, we check whether the combination of $T_{i,j,k}$ and its start time $s_{i,j,k}$ with the already-fixed TT jobs and their start times is stored in $\mathcal{L}$. If the combination is stored in $\mathcal{L}$, then the algorithm does not fix $T_{i,j,k}$ at $s_{i,j,k}$ since we already know that this combination results in a deadline miss.

5.2.3 Fixation phase

The main aim of the fixation phase is to find the start times at which the TT jobs may be fixed. As mentioned before, the fixation of a TT job is done on the entire layer of vertices V. Moreover, we know that when the FGG algorithm backtracks to a layer V, all vertices and edges that were added to the fixation graph after fixing a TT job on layer V are removed. Therefore, we can see that, for any layer V in the fixation graph, all vertices $v \in V$ have the same set of finished TT jobs and, therefore, the same set of applicable TT jobs. Moreover, similar to the ET jobs, the release time of each TT job that has some predecessors in the precedence chain depends on the time when the predecessors finish their execution. Therefore, for each vertex v in the fixation graph and for each TT job $T_{i,j,k}$, we define $v.r^{TT}_{i,j,k}$ to be the release time of $T_{i,j,k}$ in vertex v. However, note that, for any layer V in the fixation graph, an applicable TT job $T_{i,j,k}$ has the same release time in all vertices $v \in V$. To summarize, given a layer of vertices V, it holds that $v.T^F = w.T^F$, $v.T^A = w.T^A$, and $v.r^{TT}_{i,j,k} = w.r^{TT}_{i,j,k}$, for each $T_{i,j,k} \in v.T^A \cap w.T^A$ and for each two different vertices $v, w \in V$.

The fixation phase finds the start times as follows. First, it computes the earliest possible time across all vertices of the layer V when some applicable ET job may start its execution. Then, it computes the earliest start time $s_{i,j,k}$ for each applicable TT job $T_{i,j,k}$ such that $T_{i,j,k}$ is released and $\sigma'_{i,k}$ is unoccupied in all vertices of V. If the computed start time $s_{i,j,k}$ causes the applicable TT job $T_{i,j,k}$ to miss its deadline, then the fixation phase concludes and the FGG algorithm has to backtrack since we know that $T_{i,j,k}$ cannot be fixed without missing its deadline on any other layer of vertices resulting from further expansion of V. If the earliest possible time when some applicable ET job may start its execution is not strictly less than $s_{i,j,k}$, then $T_{i,j,k}$ may be fixed at $s_{i,j,k}$. In other words, this condition ensures that no collision can occur in any execution scenario.

5.2.4 Pseudocode

Fixation phase

The pseudocode of the fixation phase can be seen in Algorithm 8. The fixation phase begins with a set of vertices V. It returns either a set $\mathcal{S}^{fix}$ or the *null* pointer. $\mathcal{S}^{fix}$ contains pairs. Each pair $(T_{i,j,k}, s_{i,j,k}) \in \mathcal{S}^{fix}$ consists of a TT job $T_{i,j,k}$ that may be fixed at start time $s_{i,j,k}$. The set $\mathcal{S}^{fix}$ is initialized to $\emptyset$. Recall that all vertices from V contain the same set of applicable TT jobs, and each applicable TT job has the same release time in all vertices from V. Therefore, when we want to inspect the applicable TT jobs and their release times, we can pick an arbitrary vertex $w \in V$ as can be seen on line 3. If all TT jobs are finished, then there are no TT jobs left to be fixed, and, therefore, $\emptyset$ is returned.

The algorithm continues by computing t^f on lines 6-9 which is the earliest possible time across all vertices of V when some applicable ET job may start its execution. In other words, t^f provides us with an upper bound on the time at which an applicable TT job may be fixed. Fixing an applicable TT job at some time greater than t^f is not allowed since a collision may occur. Then, the earliest start time $s_{i,j,k}$ is computed for each applicable TT job $T_{i,j,k}$. The computation of $s_{i,j,k}$ ensures that there is no ongoing execution of any job at $s_{i,j,k}$ in any execution scenario, i.e., the processor $\sigma'_{i,k}$ is unoccupied at $s_{i,j,k}$ in all execution scenarios. If $s_{i,j,k}$ causes $T_{i,j,k}$ to miss its deadline, then the *null* pointer is returned as shown on line 15 to signal that there is an applicable TT job that cannot be fixed on V without missing its deadline. If $s_{i,j,k}$ is less than or equal to t^f, then the pair $(T_{i,j,k}, s_{i,j,k})$ is added to $\mathcal{S}^{fix}$ as can be seen on line 17. The fixation phase terminates by returning $\mathcal{S}^{fix}$. Note that $\mathcal{S}^{fix}$ may be an empty set if there is no applicable TT job that may be fixed before or at t^f.

Fixation of TT jobs in the fixation graph

The fixation of a TT job $T_{i,j,k}$ at $s_{i,j,k}$ can be seen in Algorithm 9. Note that the fixation is done on the entire layer of vertices V. This means that each vertex $v \in V$ is expanded with

Algorithm 8 Fixation phase

```
Input: set of vertices $V$
Output: set $\mathcal{S}^{fix}$ or $null$

1:  function FIXATION_PHASE($V$)
2:      $\mathcal{S}^{fix} \leftarrow \emptyset$
3:      $w \leftarrow$ arbitrary vertex from $V$
4:      if $w.T^A = \emptyset$ then
5:          return $\mathcal{S}^{fix}$                                  ▷ All TT jobs are finished
6:      $t^f \leftarrow \infty$
7:      for each $v \in V$ do
8:          for each $E_{i,j,k} \in v.E^A$ do
9:              $t^f \leftarrow \min(t^f,\ \max(v.r^{min}_{i,j,k}, v.EFT_{\sigma_{i,k}}))$   ▷ The latest time when some TT job may be fixed
10:     for each $T_{i,j,k} \in w.T^A$ do
11:         $s_{i,j,k} \leftarrow w.r^{TT}_{i,j,k}$
12:         for each $v \in V$ do
13:             $s_{i,j,k} \leftarrow \max(s_{i,j,k}, v.LFT_{\sigma'_{i,k}})$        ▷ The earliest possible start time of $T_{i,j,k}$
14:         if $s_{i,j,k} + c^{TT}_{i,j,k} > d^{TT}_{i,j,k}$ then
15:             return $null$                                       ▷ Deadline miss
16:         if $s_{i,j,k} \leq t^f$ then
17:             $\mathcal{S}^{fix} \cup \{(T_{i,j,k}, s_{i,j,k})\}$
18:     return $\mathcal{S}^{fix}$
```

$T_{i,j,k}$. The structure of the function EXPAND_BY_FIXING is similar to that of the function EXPAND_VERTEX from Algorithm 3 in Subsection 3.3.7. When creating a new vertex w, we can observe that $w.EFT_{\sigma'_{i,k}} = w.LFT_{\sigma'_{i,k}}$ since we know that $T_{i,j,k}$ finishes its execution exactly at $s_{i,j,k} + c^{TT}_{i,j,k}$. Moreover, if $T_{i,j,k}$ has a direct successor $T_{i,j,k+1}$ in the precedence chain, then $T_{i,j,k+1}$ is released at the time when $T_{i,j,k}$ finishes its execution, which can be seen on line 8. The algorithm terminates when all vertices $v \in V$ are expanded, and it returns a new layer of vertices V^{fix}.

Complete FGG procedure

Algorithm 10 provides the pseudocode of the complete FGG procedure. The main loop in the FGG algorithm is situated in the recursive function GENERATE_RECURSIVELY. This function accepts a layer of vertices V and a set $\mathcal{S}^{temp}$ as its input parameters. The set $\mathcal{S}^{temp}$ locally stores the fixed TT jobs and their start times in the form of pairs $(T_{i,j,k}, s_{i,j,k})$. The FGG is launched by calling the function FGG_INIT. As can be seen on line 1, the tabu list $\mathcal{L}$ is a global variable initialized to an empty list when launching the FGG. The FGG starts with initializing a root vertex v_r. The initialization of v_r can be seen on lines 3-9. The first layer of vertices V consists of the root vertex. The function FGG_INIT then makes the first recursive call to the function GENERATE_RECURSIVELY with V and $\emptyset$ as the input parameters.

The main while-loop in the function GENERATE_RECURSIVELY starts with checking the condition on line 14. We will skip it for now and come back to it later since it is a terminating condition. The algorithm then performs the fixation phase described in Algorithm 8 upon V. There are two possible outcomes of the fixation phase.

The first outcome of the fixation phase is that there is an applicable TT job that cannot be fixed on V without missing its deadline. In that case, the $null$ pointer is returned from

Algorithm 9 Fixation of TT jobs in the fixation graph

```
    Input: set of vertices $V$, TT job $T_{i,j,k}$, start time $s_{i,j,k}$
    Output: set of vertices $V^{fix}$

 1: function EXPAND_BY_FIXING($V$, $T_{i,j,k}$, $s_{i,j,k}$)
 2:     $V^{fix} \leftarrow \emptyset$
 3:     for each $v \in V$ do
 4:         $w \leftarrow$ new vertex with $w.FTI \leftarrow v.FTI, w.E^F \leftarrow v.E^F, w.T^F \leftarrow v.T^F \cup \{T_{i,j,k}\}, w.out \leftarrow \emptyset$
 5:         $w.EFT_{\sigma'_{i,k}} \leftarrow s_{i,j,k} + c^{TT}_{i,j,k}$
 6:         $w.LFT_{\sigma'_{i,k}} \leftarrow s_{i,j,k} + c^{TT}_{i,j,k}$
 7:         if $T_{i,j,k}$ has a direct successor $T_{i,j,k+1}$ in the precedence chain then
 8:             $w.r^{TT}_{i,j,k+1} \leftarrow s_{i,j,k} + c^{TT}_{i,j,k}$
 9:         $e \leftarrow$ new edge with $e.J \leftarrow T_{i,j,k}, e.s \leftarrow v, e.d \leftarrow w$
10:         $w.in \leftarrow \{e\}$
11:         $v.out \leftarrow \{e\}$
12:         $w.E^A \leftarrow$ applicable ET jobs based on $w.E^F$
13:         $w.T^A \leftarrow$ applicable TT jobs based on $w.T^F$
14:         $V^{fix} \leftarrow V^{fix} \cup \{w\}$
15:     return $V^{fix}$
```

the fixation phase, and the algorithm backtracks by returning $\emptyset$ from the current recursive call of the function GENERATE_RECURSIVELY, which can be seen on line 19.

The second outcome of the fixation phase is that $\mathcal{S}^{fix}$ is a set containing pairs $(T_{i,j,k}, s_{i,j,k})$ and, therefore, some TT jobs may be fixed. The algorithm then iterates over all pairs $(T_{i,j,k}, s_{i,j,k}) \in \mathcal{S}^{fix}$. Note that $\mathcal{S}^{fix}$ can also be an empty set. In that case, there are no pairs to iterate over, and the algorithm continues with executing line 29. For a pair $(T_{i,j,k}, s_{i,j,k})$, the algorithm first checks whether the combination of $(T_{i,j,k}, s_{i,j,k})$ with the already-fixed TT jobs and their start times is known to result in a deadline miss. This is done by checking the condition on line 21. If this condition is satisfied, then $T_{i,j,k}$ is not fixed at $s_{i,j,k}$ and the algorithm continues by picking another pair from $\mathcal{S}^{fix}$. Otherwise, $T_{i,j,k}$ is fixed at $s_{i,j,k}$ and a new layer of vertices V^{fix} is generated as described in Algorithm 9. The function GENERATE_RECURSIVELY is then recursively called with V^{fix} and $\mathcal{S}^{temp} \cup \{(T_{i,j,k}, s_{i,j,k})\}$ as the input parameters. The function GENERATE_RECURSIVELY returns either a valid set of start times $\mathcal{S}$ if it is found or an empty set. Therefore, the condition on line 25 ensures that the valid set of start times $\mathcal{S}$ gets propagated through the recursive calls. Returning an empty set signals that the algorithm backtracked since the current combination of the fixed TT jobs and their start times caused a deadline miss to occur. In that case, the combination $\mathcal{S}^{temp} \cup \{(T_{i,j,k}, s_{i,j,k})\}$ is added to the tabu list $\mathcal{L}$. Moreover, all vertices and edges that were generated as a result of fixing $T_{i,j,k}$ at $s_{i,j,k}$ on layer V are removed from the fixation graph, which is expressed on line 28. If other TT jobs may be fixed on V, then the algorithm continues by fixing them one by one.

If all fixations of the TT jobs on V resulted in a deadline miss, then the algorithm continues with executing line 29. This line states that if all ET jobs are finished, then the algorithm needs to backtrack since it was not able to fix the unfinished TT jobs in such a way that all deadlines are met. If there are some unfinished ET jobs on V, then the expansion phase, which is described in Algorithm 3 in Subsection 3.3.7, is performed upon V. Note that when expanding a vertex v using the function EXPAND_VERTEX from Algorithm 3, the new vertex w is additionally initialized with $w.T^F = v.T^F$, and, therefore, with $w.T^A = v.T^A$ in

Algorithm 10 Fixation Graph Generation

```
Input: set of TT tasks $\mathcal{T}$, set of ET tasks $\mathcal{E}$, number of processors $m$
Output: valid set of start times $\mathcal{S}$ if it is found, $\emptyset$ otherwise

1: $\mathcal{L} \leftarrow [\,]$                                          ▷ Tabu list $\mathcal{L}$ is a global variable
2: function FGG_INIT($\mathcal{T}$, $\mathcal{E}, m$)
3:     $v_r \leftarrow$ new vertex with $v_r.E^F \leftarrow \emptyset, v_r.T^F \leftarrow \emptyset, v_r.in \leftarrow \emptyset, v_r.out \leftarrow \emptyset$
4:     $v_r.E^A \leftarrow$ applicable ET jobs based on $v_r.E^F$
5:     $v_r.T^A \leftarrow$ applicable TT jobs based on $v_r.T^F$
6:     for $\pi = 1$ to $m$ do
7:         $v_r.EFT_\pi \leftarrow 0$
8:         $v_r.LFT_\pi \leftarrow 0$
9:     $v_r.FTI \leftarrow \{[v_r.EFT_1, v_r.LFT_1], \ldots, [v_r.EFT_m, v_r.LFT_m]\}$
10:    $V \leftarrow \{v_r\}$
11:    return GENERATE_RECURSIVELY($V, \emptyset$)
12: function GENERATE_RECURSIVELY($V$, $\mathcal{S}^{temp}$)
13:    while true do
14:        if $v.E^A = \emptyset \wedge v.T^A = \emptyset$, $\forall v \in V$ then
15:            $\mathcal{S} \leftarrow$ collected start times from $\mathcal{S}^{temp}$
16:            return $\mathcal{S}$                                        ▷ Valid $\mathcal{S}$ found
17:        $\mathcal{S}^{fix} \leftarrow$ FIXATION_PHASE($V$)
18:        if $\mathcal{S}^{fix} = null$ then
19:            return $\emptyset$                 ▷ Deadline of some applicable TT job cannot be met
20:        for each $(T_{i,j,k}, s_{i,j,k}) \in \mathcal{S}^{fix}$ do
21:            if $\mathcal{L}$ contains $\mathcal{S}^{temp} \cup \{(T_{i,j,k}, s_{i,j,k})\}$ then
22:                continue          ▷ $\mathcal{S}^{temp} \cup \{(T_{i,j,k}, s_{i,j,k})\}$ is known to result in a deadline miss
23:            $V^{fix} \leftarrow$ EXPAND_BY_FIXING($V, T_{i,j,k}, s_{i,j,k}$)
24:            $\mathcal{S} \leftarrow$ GENERATE_RECURSIVELY($V^{fix}, \mathcal{S}^{temp} \cup \{(T_{i,j,k}, s_{i,j,k})\}$)
25:            if $\mathcal{S} \neq \emptyset$ then
26:                return $\mathcal{S}$                                    ▷ Propagate valid $\mathcal{S}$
27:            add $\mathcal{S}^{temp} \cup \{(T_{i,j,k}, s_{i,j,k})\}$ to $\mathcal{L}$
28:            remove all edges and vertices beyond $V$
29:        if $v.E^A = \emptyset$, $\forall v \in V$ then
30:            return $\emptyset$                                     ▷ All ET jobs are finished
31:        $V^{ex} \leftarrow$ EXPANSION_PHASE($V$)
32:        for each $v \in V^{ex}$ do
33:            $E_{i,j,k} \leftarrow e.J$ where $v.in = \{e\}$              ▷ $e$ is the only edge in $v.in$
34:            if $v.LFT_{\sigma_{i,k}} > d^{ET}_{i,j,k}$ then
35:                return $\emptyset$                                       ▷ Deadline miss
36:        MERGE_PHASE($V^{ex}$)
37:        $V \leftarrow V^{ex}$
```

the fixation graph. If a deadline miss occurs during the expansion phase, then the algorithm backtracks as can be seen on line 35. Otherwise, the merge phase, which is described in Algorithm 5 in Subsection 3.3.7, is performed upon the result of the expansion phase V^{ex}. The main loop then starts its next iteration with the result of the merge phase being assigned to V. Note that even though not stated explicitly, the function GENERATE_RECURSIVELY and all functions called within GENERATE_RECURSIVELY have access to the inputs of Algorithm 10, i.e., to $\mathcal{T}$, $\mathcal{E}$, and m.

When the condition on line 14 is satisfied, then we know that all ET and TT jobs are finished and no deadline miss occurred. Hence, the TT jobs are fixed in such a way that their start times form a valid set of start times. Therefore, we obtain the valid set of start times $\mathcal{S}$ simply by collecting the start times of the fixed TT jobs from $\mathcal{S}^{temp}$, i.e., the second element of each $(T_{i,j,k}, s_{i,j,k}) \in \mathcal{S}^{temp}$ is collected. $\mathcal{S}$ is then propagated through the recursive calls of the function GENERATE_RECURSIVELY. Eventually, Algorithm 10 terminates and returns $\mathcal{S}$. If all combinations of start times of the TT jobs found by the FGG algorithm result in a deadline miss, then Algorithm 10 terminates and returns $\emptyset$ to signal that it was not able to find a valid set of start times.

5.2.5 Discussion

As already mentioned, the proposed FGG algorithm is safe but pessimistic. We have identified three main sources of pessimism in the FGG algorithm. First, the FGG inherits the expansion and the merge phase from the proposed SAG approach to schedulability analysis. Therefore, the FGG algorithm also inherits the pessimism rooted in the expansion phase from the SAG approach. The second source of pessimism stems from the fact that the FGG is not able to find all possible start times for the TT jobs since the fixation phase only considers vertices of a single layer. The fixation phase would have to operate on multiple levels of the fixation graph. The third source of pessimism stems from the fact that TT jobs are always fixed as early as possible, and, therefore, not all possible combinations of start times are considered. The second and the third source of pessimism are discussed in [14, 15] as well. We have yet to come up with a scalable solution that would eradicate the inexactness of the proposed FGG algorithm in the dedicated multiprocessor setting.

However, we propose a speedup for the FGG algorithm. The logic behind this speedup is that when fixing the TT jobs, the algorithm backtracks more eagerly. Specifically, when the algorithm finds out that the combination of the fixed TT jobs and their start times is in the tabu list $\mathcal{L}$, we make the algorithm backtrack instead of continuing to fix other TT jobs on the current layer. Therefore, this speedup can be simply achieved by substituting the "**continue**" on line 22 in Algorithm 10 for "**return** $\emptyset$". We will call the modified version of the FGG algorithm Fixation Graph Generation with Eager Backtracking (FGG-EB). Note that this approach has another source of pessimism stemming from the possibility of not considering some combinations of start times found by the fixation phase. The performance of both FGG and FGG-EB is evaluated in Chapter 6.

Chapter 6

ET+TT solution evaluation

This chapter discusses the evaluation of the proposed algorithms from Chapter 5. The algorithms are implemented in C++. The implementation supports the C++20 standard. The evaluation is done empirically, i.e., the results are derived from experiments and measurements conducted on datasets of randomly generated instances, which are publicly available on GitHub[1]. The experiments were run on a system with 2x Intel® Xeon® E5-2690 v4 CPU, 14 Cores/CPU; 2.6GHz; 35 MB SmartCache; with 256 GB DDR4, ECC. Each instance was run on a single core and the memory available to each instance was restricted to 8 GB. The time limit was set to 10 minutes.

6.1 Instance generation

The instances were generated utilizing the generation procedure from Section 4.2. This procedure is described to generate instances consisting of ET tasks. However, it can be easily modified such that it generates instances of both ET and TT tasks. First, the network model from Section 4.1 can be utilized for the TT tasks in the same way as it is utilized for the ET tasks. The generation procedure additionally utilizes parameter n' that determines the total number of TT tasks of the instance. Moreover, for the generation of the TT tasks, the parameters A^j, A^c, p_{min}, and p_{max} are not considered since the TT tasks are defined to have no release jitter, no execution time variation, and no priority. The values for τ_i^{TT}, r_i^{TT}, and d_i^{TT} are obtained analogically to τ_i^{ET}, r_i^{max}, and d_i^{ET}, respectively. Furthermore, when generating c_i^{TT} and c_i^{max}, the aim is that the utilization of each processor is at most equal to U. Therefore, U_π, which is the utilization of processor π, is calculated as

$$U_\pi = \sum_{\{\mathcal{E}_i \in \mathcal{E} \mid \pi \in \sigma_i\}} \frac{c_i^{max}}{\tau_i^{ET}} + \sum_{\{\mathcal{T}_i \in \mathcal{T} \mid \pi \in \sigma'_i\}} \frac{c_i^{TT}}{\tau_i^{TT}}$$

when an instance consisting of both ET and TT tasks is generated. To summarize, for the combination of ET and TT tasks, the generation procedure utilizes its input parameters as follows:

- **Parameters utilized for both ET and TT tasks** – $\left|\mathcal{V}^{NET}\right|$, A^η, A^τ, U, A^r, A^d.
- **Parameters utilized exclusively for ET tasks** – n, A^j, A^c, p_{min}, p_{max}.
- **Parameters utilized exclusively for TT tasks** – n'.

[1] `https://github.com/halasluk/ETTT_dedicated_multicore`

6.1.1 Generated datasets

First, one dataset $\mathcal{D}_0^b$ was generated for the experiments in Section 6.2. $\mathcal{D}_0^b$ contains 10000 instances. For the generation of $\mathcal{D}_0^b$, a single network model was generated with $\left|\mathcal{V}^{NET}\right| = 4$. The instances were generated with the following parameters: $n = 3$, $n' = 3$, $A^\eta = 12$, $A^\tau = 6$, $U = 0.3$, $A^r = A^d = 0.6$, $A^j = A^c = 0.6$, $p_{min} = 1$, $p_{max} = 3$. Therefore, $\mathcal{D}_0^b$ contains rather simple instances since they consist of only 3 ET tasks and 3 TT tasks and since their target hyperperiod is very low.

Then, we generated 9 datasets for the experiments in Section 6.3. Each of these datasets contains 1000 instances. For the generation of these 9 datasets, a single network model was generated with $\left|\mathcal{V}^{NET}\right| = 10$. Each of these datasets was then generated with the following parameters: $n = 8$, $1 \leq n' \leq 20$, $A^\eta = 10^7$, $A^\tau = 10^6$, $U = 0.3$, $p_{min} = 1$, $p_{max} = 4$. Furthermore, each of these datasets was generated with a different combination of values of parameters A^r, A^d, A^j, and A^c. The combination of values of these parameters for each dataset can be seen in Table 6.1.

	$A^r = A^d = 0.1$	$A^r = A^d = 0.2$	$A^r = A^d = 0.3$
$A^j = A^c = 0$	$\mathcal{D}_1^b$	$\mathcal{D}_2^b$	$\mathcal{D}_3^b$
$A^j = A^c = 0.3$	$\mathcal{D}_4^b$	$\mathcal{D}_5^b$	$\mathcal{D}_6^b$
$A^j = A^c = 0.6$	$\mathcal{D}_7^b$	$\mathcal{D}_8^b$	$\mathcal{D}_9^b$

Table 6.1: Values of parameters A^r, A^d, A^j, and A^c used for generation of datasets $\mathcal{D}_1^b, \ldots, \mathcal{D}_9^b$.

Datasets $\mathcal{D}_1^b, \ldots, \mathcal{D}_9^b$ contain arguably harder instances than dataset $\mathcal{D}_0^b$ since the instances consist of 8 ET tasks and up to 20 TT tasks and since the target hyperperiod is nearly 10^6 times larger than the target hyperperiod of the instances from $\mathcal{D}_0^b$.

6.2 Evaluation of pessimism in the FGG algorithm and the FGG-EB algorithm

To evaluate the amount of pessimism in the FGG algorithm and the FGG-EB algorithm, we needed to generate a dataset of sufficiently small instances such that the exact brute force algorithm can verify the schedulability of each instance in a reasonable amount of time. Therefore, dataset $\mathcal{D}_0^b$ was generated. The schedulability of the instances from $\mathcal{D}_0^b$ for each of the evaluated algorithms can be seen in Table 6.2. We can observe that the false negative rate of the FGG algorithm is approximately 17.56 %, i.e., the FGG algorithm is not able to find a valid set of start times even though it exists in 17.56 % of the instances. Therefore, the true positive rate of the FGG algorithm is approximately 82.44 %, i.e., the FGG algorithm is able to find a valid set of start times when it exists in 82.44 % of the instances.

Furthermore, we evaluate the false negative rate and the true positive rate of the modified FGG algorithm, i.e., the FGG-EB algorithm. We can observe that the false negative rate of the FGG-EB algorithm is approximately 19.61 %. Therefore, the true positive rate of the FGG-EB algorithm is approximately 80.39 %.

To summarize, the FGG algorithm is slightly more successful in finding a valid set of start times when it exists than the more pessimistic FGG-EB algorithm. The difference amounts

	All instances	BF schedulable only	BF unschedulable only
BF schedulable	8373	8373	0
BF unschedulable	1627	0	1627
FGG schedulable	6903	6903	0
FGG unschedulable	3097	1470	1627
FGG-EB schedulable	6731	6731	0
FGG-EB unschedulable	3269	1642	1627

Table 6.2: Resulting schedulability of the instances from $\mathcal{D}_0^b$ for the brute force (BF) algorithm, the FGG algorithm, and the FGG-EB algorithm.

to approximately 2.05 % of the instances when the solution exists.

6.3 Evaluation of schedulability and runtime of the FGG algorithm and the FGG-EB algorithm

The runtimes of the FGG algorithm and the FGG-EB algorithm were measured for datasets $\mathcal{D}_1^b, \dots, \mathcal{D}_9^b$. Figure 6.1 and Figure 6.2 show the runtimes of the FGG algorithm and the FGG-EB algorithm, respectively, relative to the number of TT tasks of each instance for each dataset. Figure 6.3 and Figure 6.4 show the runtimes of the FGG algorithm and the FGG-EB algorithm, respectively, relative to the number of TT jobs of each instance for each dataset. Note that the layout of the subfigures in each of these figures corresponds to the layout of datasets $\mathcal{D}_1^b, \dots, \mathcal{D}_9^b$ in Table 6.1 in Subsection 6.1.1, i.e., the runtimes for dataset $\mathcal{D}_1^b$, for example, are shown in the subfigure in the top left corner of each of these figures. Note that the label "Memout" is used for instances that reached the memory limit, which is set to 8 GB.

For both FGG and FGG-EB, we can observe that the instances from datasets with higher release jitter and execution time variation tend to have longer runtimes. Recall that release jitter and execution time variation are considered only for ET tasks, and each instance contains 8 ET tasks. Therefore, even the smallest instances from datasets $\mathcal{D}_7^b, \mathcal{D}_8^b, \mathcal{D}_9^b$ that contain just a few TT jobs reached the time limit of 10 minutes, for both FGG and FGG-EB. Moreover, for both FGG and FGG-EB, we can observe a general pattern, especially for datasets $\mathcal{D}_1^b, \dots, \mathcal{D}_6^b$, that the schedulable instances tend to have lower runtimes than the unschedulable instances. This stems from the fact that, for schedulable instances, the algorithms terminate when a solution is found, however, for infeasible instances, the algorithms explore a much larger search space. Furthermore, the smallest instance that reached the memory limit across all evaluated datasets contained 45 TT jobs when running the FGG algorithm, whereas the smallest instance that reached the memory limit across all evaluated datasets contained 126 TT jobs when running the FGG-EB algorithm. The largest solved instance consists of 404 TT jobs and 128 ET jobs. Note that both FGG and FGG-EB were able to find a solution for this instance.

The schedulability of the instances from each dataset is shown in Table 6.3. Note that the layout of Table 6.3 corresponds to the layout of Table 6.1 in Subsection 6.1.1. Each cell in Table 6.3 that corresponds to a dataset contains the results for both FGG and FGG-EB. When we inspect Table 6.3 column-wise, we can observe that the datasets with the largest slack time, i.e., datasets $\mathcal{D}_1^b, \mathcal{D}_4^b, \mathcal{D}_7^b$, contain the maximal number of schedulable instances in their

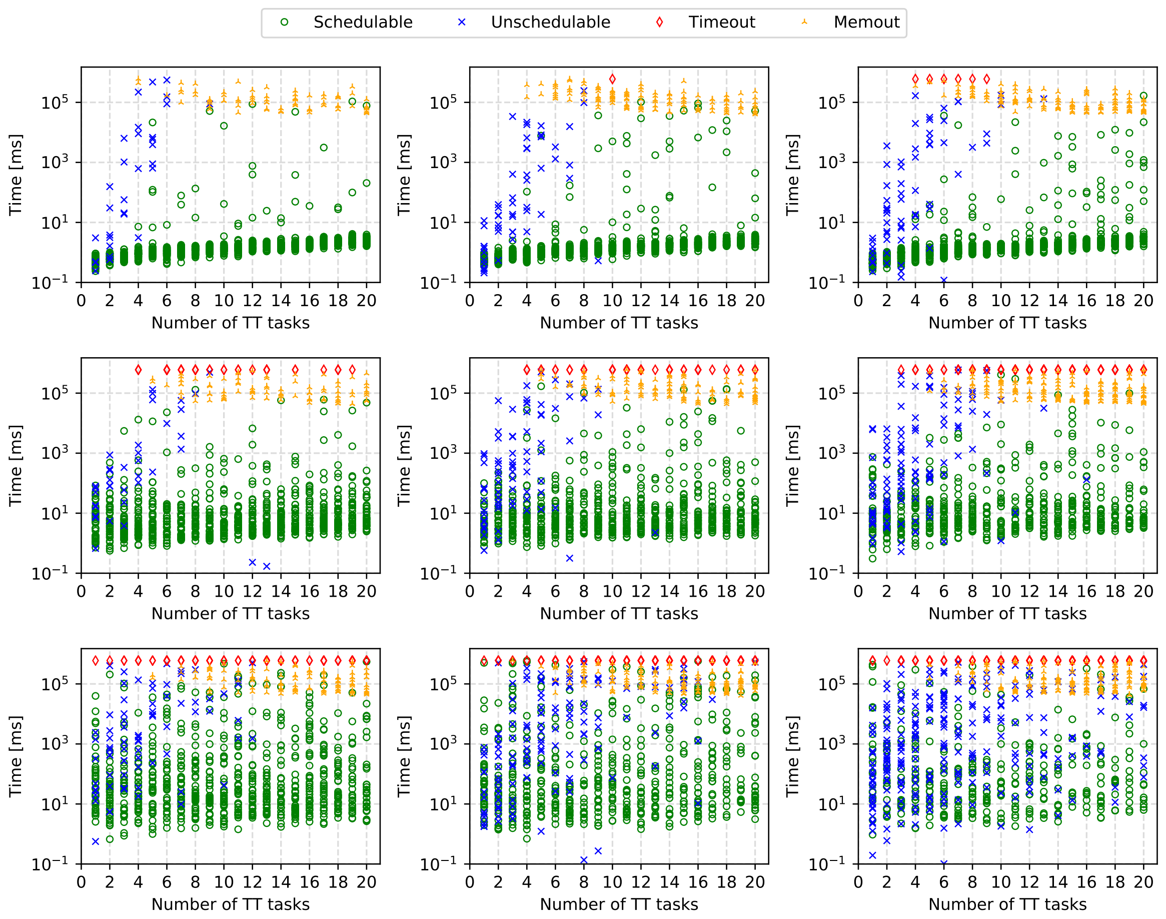

Figure 6.1: Runtimes of the FGG algorithm for datasets $\mathcal{D}_1^b, \ldots, \mathcal{D}_9^b$ relative to the number of TT tasks.

respective rows, for both FGG and FGG-EB. For datasets with smaller slack time, the number of schedulable instances decreases, whereas the number of unschedulable instances increases. Moreover, we can observe that the number of instances that reached the time limit or the memory limit is lower for FGG-EB across all evaluated datasets. However, for each evaluated dataset, the FGG algorithm was able to solve more instances than the FGG-EB algorithm even though the FGG algorithm reached the time limit or the memory limit more often than the FGG-EB algorithm. To summarize, for the FGG algorithm, dataset $\mathcal{D}_1^b$ has the highest schedulability ratio of approximately 96.4 %, whereas dataset $\mathcal{D}_9^b$ has the lowest schedulability ratio of 57.5 %. For the FGG-EB algorithm, dataset $\mathcal{D}_1^b$ has the highest schedulability ratio of approximately 89.8 %, whereas dataset $\mathcal{D}_9^b$ has the lowest schedulability ratio of approximately 36.5 %. Therefore, although the FGG-EB algorithm is faster than the FGG algorithm, the FGG-EB algorithm yields poorer results in terms of schedulability than the FGG algorithm.

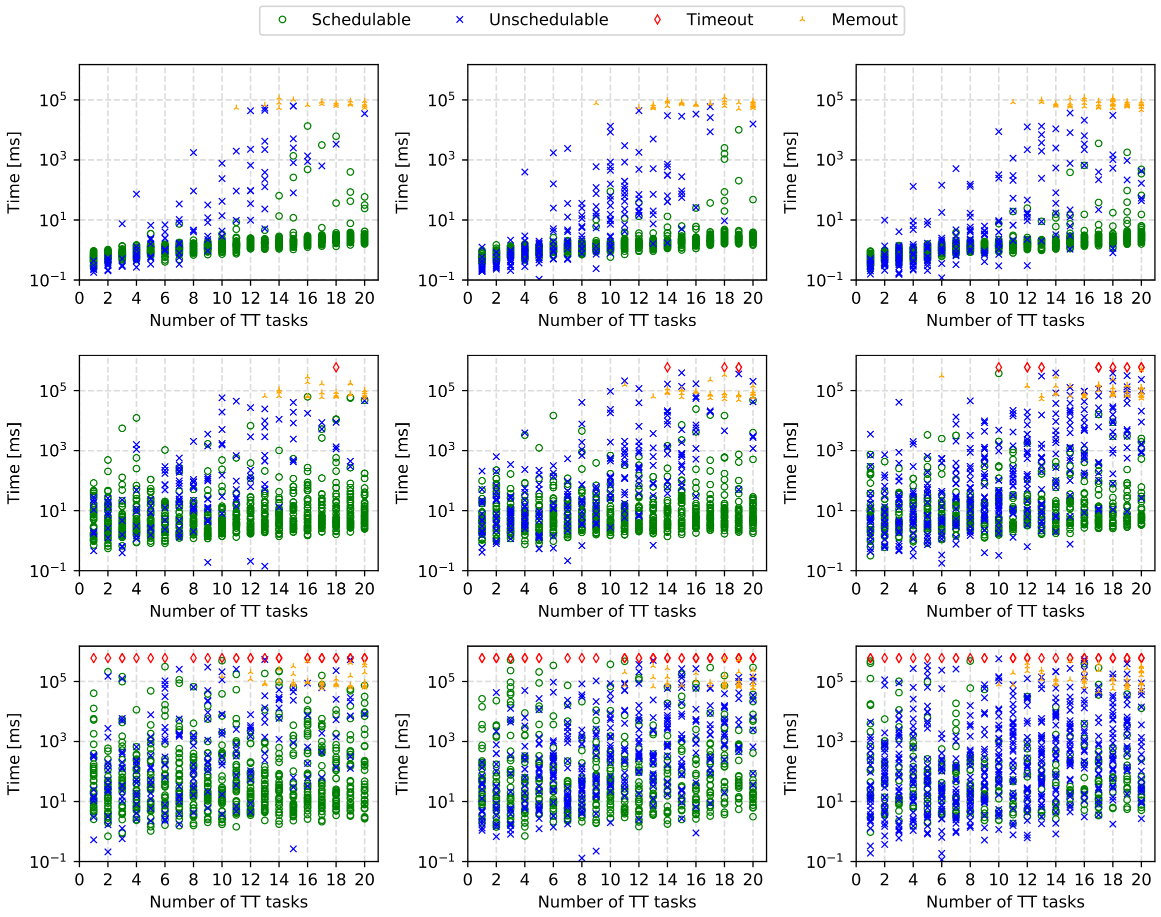

Figure 6.2: Runtimes of the FGG-EB algorithm for datasets $\mathcal{D}_1^b, \ldots, \mathcal{D}_9^b$ relative to the number of TT tasks.

	$A^r = A^d = 0.1$	$A^r = A^d = 0.2$	$A^r = A^d = 0.3$
$A^j = A^c = 0$	891, 33, 0, 76 879, 100, 0, 21	815, 52, 1, 132 778, 187, 0, 35	796, 65, 6, 133 750, 202, 0, 48
$A^j = A^c = 0.3$	838, 37, 37, 88 833, 139, 1, 27	745, 75, 53, 127 735, 230, 4, 31	574, 115, 123, 188 554, 382, 17, 47
$A^j = A^c = 0.6$	736, 74, 77, 113 727, 195, 38, 40	528, 135, 171, 166 523, 368, 63, 46	345, 255, 226, 174 329, 572, 49, 50

Table 6.3: Schedulability of instances from datasets $\mathcal{D}_1^b, \ldots, \mathcal{D}_9^b$. Each cell corresponding to a dataset contains two rows. The first row contains the results of the FGG algorithm and the second row contains the results of the FGG-EB algorithm. Each row in a cell corresponding to a dataset consists of the total number of schedulable instances, the total number of unschedulable instances, the total number of instances that reached the time limit, and the total number of instances that reached the memory limit in this order.

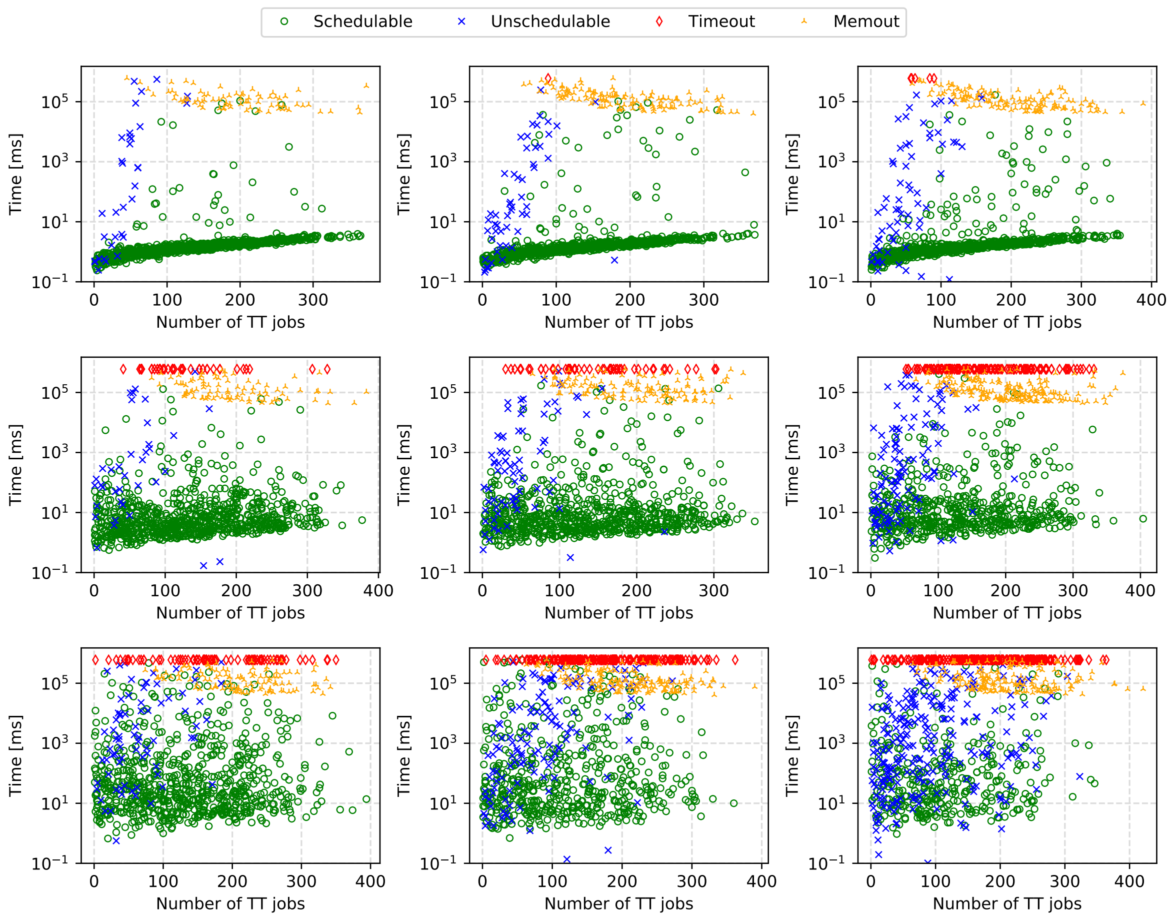

Figure 6.3: Runtimes of the FGG algorithm for datasets $\mathcal{D}_1^b, \dots, \mathcal{D}_9^b$ relative to the number of TT jobs.

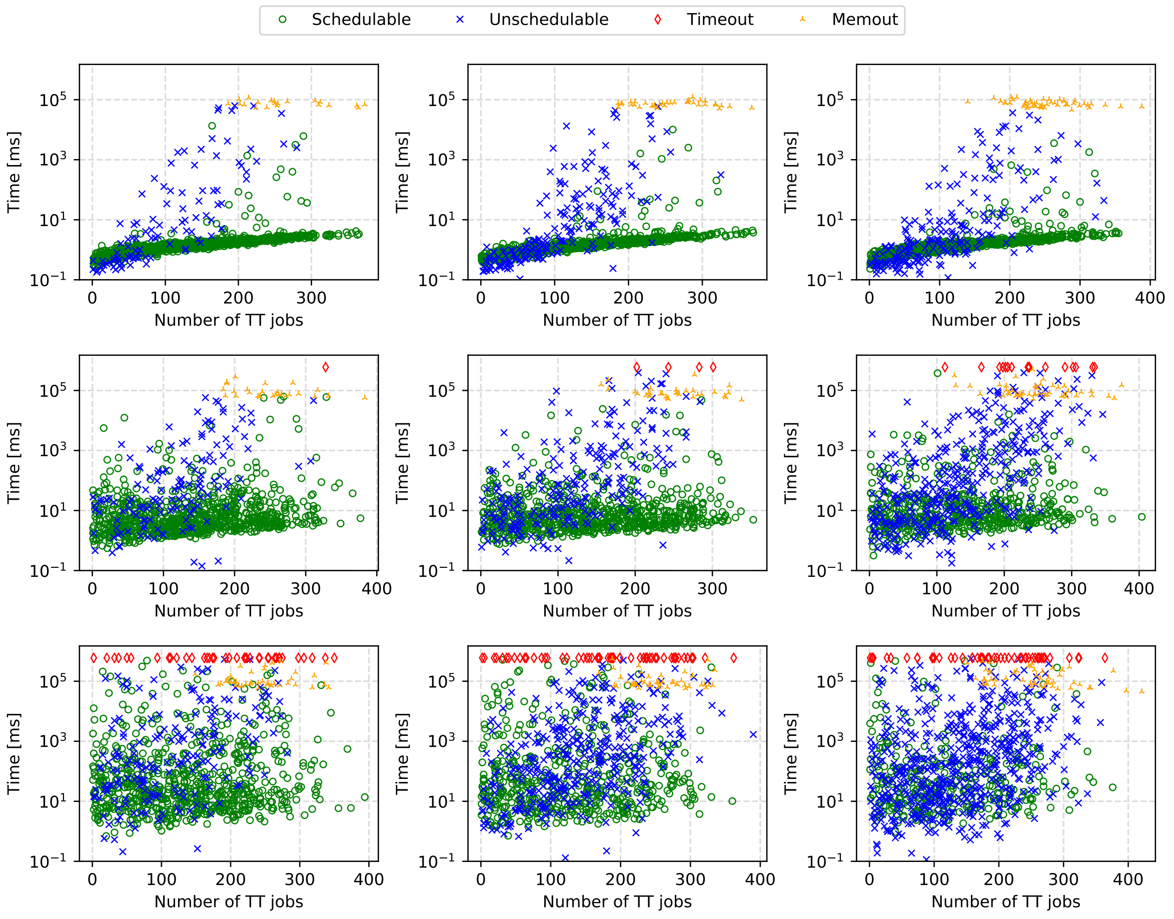

Figure 6.4: Runtimes of the FGG-EB algorithm for datasets $\mathcal{D}_1^b, \ldots, \mathcal{D}_9^b$ relative to the number of TT jobs.

Chapter 7

Conclusion

This thesis addressed the problem of the combination of TT and ET scheduling with dedicated resources and precedences. The first part of this thesis addressed the ET schedulability analysis in our setting. We described an exact yet non-scalable brute force approach to schedulability analysis. Furthermore, we proposed a scalable approach to schedulability analysis based on the SAG generation algorithm, which is safe but pessimistic. The SAG approach also incorporates a speedup in the form of a necessary worst-case approach to schedulability analysis. The proposed algorithms utilized the FP-EDF policy.

Then, the proposed algorithms were empirically evaluated on datasets of randomly generated instances. First, we measured the amount of pessimism in the SAG approach by comparing its results to the results of the exact brute force approach on small instances. We found out that the SAG approach successfully verifies that the instance is schedulable in 99.73 % of the instances. Moreover, the SAG approach is able to verify that the instance is unschedulable in a matter of tens of microseconds in 81.54 % of the unschedulable instances thanks to the worst-case approach speedup. The runtime of the SAG approach increases with the amount of release jitter and execution time variation. The largest instance with non-zero release jitter and execution time variation that was verified to be schedulable by the SAG approach within the time limit of 10 minutes consisted of 350 ET jobs. The schedulability ratio of the SAG approach decreases with the decrease in slack time. For the evaluated datasets, the schedulability ratio of the SAG approach ranged from 49.9 % up to 82.4 %.

The second part of the thesis addressed the problem of finding a valid set of start times for the TT jobs. We described an exact yet non-scalable brute force algorithm. Moreover, we proposed the FGG algorithm that extends the SAG generation algorithm. The FGG algorithm is scalable and safe but pessimistic. Furthermore, we proposed a slightly modified version of the FGG algorithm which we call the FGG-EB algorithm.

The proposed algorithms for finding a valid set of start times for the TT jobs were empirically evaluated on datasets of randomly generated instances as well. First, we measured the amount of pessimism in both FGG and FGG-EB by comparing their results to the results of the exact brute force algorithm on small instances. We found out that the FGG algorithm is able to find a solution when it exists in 82.44 % of the instances, whereas the FGG-EB algorithm is able to find a solution when it exists in 80.39 % of the instances. Therefore, the FGG algorithm is less pessimistic than the FGG-EB algorithm. Similar to the SAG approach to ET schedulability analysis, the runtimes of both FGG and FGG-EB increase with the amount of release jitter and execution time variation as the runtimes depend not only on the number of TT jobs but also on the number of ET jobs. The largest solved instance consisted of 404 TT jobs and 128 ET jobs. Both FGG and FGG-EB were able to solve this instance. Furthermore, for all evaluated datasets, fewer instances reached the time limit of 10 minutes or the memory limit of 8 GB when running the FGG-EB algorithm than in the case of the FGG algorithm, however, the schedulability ratio of the FGG algorithm was higher than that of the FGG-EB algorithm. Similar to the SAG approach to ET schedulability analysis, the schedulability ratios of both FGG and FGG-EB decrease with the decrease in slack time. For the evaluated datasets, the schedulability ratio of the FGG algorithm ranged from 57.5 % up to 96.4 %, and the schedulability ratio of the FGG-EB algorithm ranged from 36.5 % up to 89.8 %. To summarize, the FGG-EB algorithm requires less time and memory than the FGG algorithm, however, the FGG algorithm provides better results as it is able to solve more instances than the FGG-EB algorithm.

Bibliography

[1] J. Real, S. Sáez, and A. Crespo, "A hierarchical architecture for time- and event-triggered real-time systems," *Journal of Systems Architecture*, vol. 101, p. 101652, Dec. 2019, https://doi.org/10.1016/j.sysarc.2019.101652.

[2] B. Potteiger, A. Dubey, F. Cai, X. Koutsoukos, and Z. Zhang, "Moving target defense for the security and resilience of mixed time and event triggered cyber–physical systems," *Journal of Systems Architecture*, vol. 125, p. 102420, Apr. 2022, https://doi.org/10.1016/j.sysarc.2022.102420.

[3] A. Finzi and S. S. Craciunas, "Integration of SMT-based Scheduling with RC Network Calculus Analysis in TTEthernet Networks," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, Sep. 2019, pp. 192–199, https://doi.org/10.1109/etfa.2019.8869365.

[4] A. Finzi, S. S. Craciunas, and M. Boyer, "A Real-time Calculus Approach for Integrating Sporadic Events in Time-triggered Systems," 2022, https://doi.org/10.48550/arXiv.2204.10264.

[5] S. Srinivasan, G. Nelissen, and R. J. Bril, "Work-in-Progress: Analysis of TSN Time-Aware Shapers using Schedule Abstraction Graphs," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, Dec. 2021, pp. 508–511, https://doi.org/10.1109/rtss52674.2021.00052.

[6] L. Leonardi, L. L. Bello, and G. Patti, "Combining Earliest Deadline First Scheduling with Scheduled Traffic Support in Automotive TSN-Based Networks," *Applied System Innovation*, vol. 5, no. 6, p. 125, Dec. 2022, https://doi.org/10.3390/asi5060125.

[7] M. Barzegaran, N. Reusch, L. Zhao, S. S. Craciunas, and P. Pop, "Real-Time Traffic Guarantees in Heterogeneous Time-sensitive Networks," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. ACM, Jun. 2022, pp. 46–57, https://doi.org/10.1145/3534879.3534921.

[8] A. Berisa, L. Zhao, S. S. Craciunas, M. Ashjaei, S. Mubeen, M. Daneshtalab, and M. Sjödin, "AVB-aware Routing and Scheduling for Critical Traffic in Time-sensitive Networks with Preemption," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. ACM, Jun. 2022, pp. 207–218, https://doi.org/10.1145/3534879.3534926.

[9] T. Pop, P. Eles, and Z. Peng, "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign - CODES 2002.* ACM Press, 2002, pp. 187–192, https://doi.org/10.1109/CODES.2002.1003623.

[10] A. Albert *et al.*, "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," *Embedded World*, vol. 171902, pp. 235–252, Jan. 2004. [Online]. Available: https://www.researchgate.net/publication/228803355_Comparison_of_event-triggered_and_time-triggered_concepts_with_regard_to_distributed_control_systems

[11] Y. Itami, T. Ishigooka, and T. Yokoyama, "A Distributed Computing Environment for Embedded Control Systems with Time-Triggered and Event-Triggered Processing," in *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.* IEEE, Aug. 2008, pp. 45–54, https://doi.org/10.1109/rtcsa.2008.38.

[12] P. Pedreiras and L. Almeida, "Combining event-triggered and time-triggered traffic in FTT-CAN: analysis of the asynchronous messaging system," in *2000 IEEE International Workshop on Factory Communication Systems. Proceedings (Cat. No.00TH8531).* IEEE, 2000, pp. 67–75, https://doi.org/10.1109/wfcs.2000.882535.

[13] L. Almeida, P. Pedreiras, and J. Fonseca, "The FTT-CAN protocol: why and how," *IEEE Transactions on Industrial Electronics*, vol. 49, no. 6, pp. 1189–1201, Dec. 2002, https://doi.org/10.1109/tie.2002.804967.

[14] M. Jaroš, "Combination of time-triggered and event-triggered scheduling," Master's thesis, Czech Technical University in Prague, 2022. [Online]. Available: http://hdl.handle.net/10467/101699

[15] M. Vlk, M. Jaroš, and Z. Hanzálek, "Combining Event-Triggered and Time-Triggered Scheduling Based on Fixation Graph," Under review.

[16] M. Nasri and B. B. Brandenburg, "An Exact and Sustainable Analysis of Non-preemptive Scheduling," in *2017 IEEE Real-Time Systems Symposium (RTSS).* IEEE, Dec. 2017, pp. 12–23, https://doi.org/10.1109/rtss.2017.00009.

[17] S. Ranjha, G. Nelissen, and M. Nasri, "Partial-Order Reduction for Schedule-Abstraction-based Response-Time Analyses of Non-Preemptive Tasks," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, May 2022, https://doi.org/10.1109/rtas54340.2022.00018.

[18] M. Nasri, G. Nelissen, and B. B. Brandenburg, "A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 9:1–9:23, https://doi.org/10.4230/LIPIcs.ECRTS.2018.9.

[19] M. Nasri, G. Nelissen, and B. B. Brandenburg, “Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling,” in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 21:1–21:23, https://doi.org/10.4230/LIPIcs.ECRTS.2019.21.

[20] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–44, Oct. 2011, https://doi.org/10.1145/1978802.1978814.

[21] K. Brejchová, “Heuristics for Periodic Scheduling,” Bachelor’s thesis, Czech Technical University in Prague, 2019. [Online]. Available: http://hdl.handle.net/10467/82385