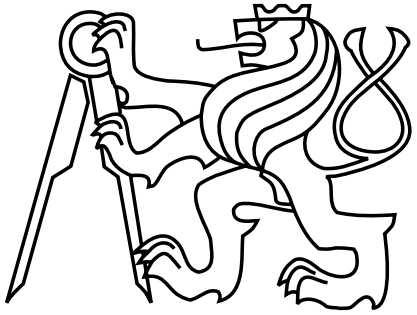
Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science



Combination of Time-triggered and Event-triggered Scheduling

Master's thesis

Author: Bc. Marek Jaroš
Supervisor: Mgr. Marek Vlk, Ph.D.
Year: 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Jaroš** Jméno: **Marek** Osobní číslo: **474734**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Kombinace time-triggered a event-triggered rozvrhování

Název diplomové práce anglicky:

Combination of time-triggered and event-triggered scheduling

Pokyny pro vypracování:

Real-time embedded systems mostly involve both time-triggered (TT) and event-triggered (ET) ways of communication/calculation. While the TT paradigm is necessary for the jobs of the highest criticality and allows easier certification of the system, ET scheduling brings higher utilization or throughput. The goal of this thesis is to design, develop, and evaluate an algorithm that incorporates both TT and ET scheduling in an effort to improve system properties such as schedulability ratio, latencies, timeliness guarantees, etc.
The first step will be to implement a schedulability test for jobs with variation in release times and execution times (ET jobs) inspired by [1]. The second step will be to propose an algorithm for scheduling jobs without variations in release times and execution times (TT jobs) such that no execution scenario of the ET and TT jobs would violate a deadline constraint. The idea is to first craft a schedule for TT jobs and then invoke the schedulability test for ET jobs with the TT jobs already fixed. If the schedulability test discovers a deadline miss, it will identify the source of conflict. This information about the conflict will be suitably used to update the TT schedule regarding the discovered inconsistency. This process will be iteratively repeated until it finds a feasible solution or proves infeasibility.
The work will include an experimental evaluation focusing on the success rate (schedulability) and the speed of the proposed algorithms (performance). First, the proposed schedulability test will be compared against that of [1] on 960 instances for various release jitter and execution time variation. Second, the implemented scheduling policies will be evaluated on 1800 instances for various utilizations. Finally, the algorithm combining the offline generation of static schedules for TT jobs and the schedulability test for ET jobs will be evaluated on at least 2000 instances.

Seznam doporučené literatury:

[1] Nasri, Mitra, and Bjorn B. Brandenburg. "An exact and sustainable analysis of non-preemptive scheduling." 2017 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2017.
[2] Isovic, Damir, and Gerhard Fohler. "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints." In Proceedings 21st IEEE Real-Time Systems Symposium, pp. 207-216. IEEE, 2000.
[3] Albert, Amos. "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems." Embedded world 2004 (2004): 235-252.
[4] Pop, Traian, Petru Eles, and Zebo Peng. "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems." In Proceedings of the tenth international symposium on Hardware/software codesign, pp. 187-192. 2002.
[5] Wen, Shixi, Ge Guo, and Wing Shing Wong. "Hybrid event-time-triggered networked control systems: Scheduling-event-control co-design." Information Sciences 305 (2015): 269-284.
[6] Murshed, Ayman, Roman Obermaisser, Hamidreza Ahmadian, and Ala Khalifeh. "Scheduling and allocation of time-triggered and event-triggered services for multi-core processors with networks-on-a-chip." In 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), pp. 1424-1431. IEEE, 2015.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Mgr. Marek Vlk, Ph.D. optimalizace CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **02.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

Mgr. Marek Vlk, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Declaration

I declare that the presented thesis was developed independently and that I have listed all used materials and information sources.

Prague, ...

Acknowledgement

I would like to thank my supervisor Marek Vlk for his excellent guidance and support throughout the making of this thesis. I would also like to thank Zdeněk Hanzálek for the opportunity to work on such an interesting topic.

Abstrakt

Time-triggered (TT) systémy poskytují spolehlivé chování, zatímco event-triggered (ET) systémy poskytují vyšší flexibilitu a efektivněji využívají časovou šířku pásma. Tato práce formuluje framework kombinující ET a TT rozvrhování nepreemptivních úloh na jednoprocesoru tak, aby bylo dosaženo výhod obou systémů.

Přístup, založený na generování rozvrhovacího grafu, navržený autory Nasri a Brandenburg [RTSS 2017, s. 12–23], se hodí pro kombinaci rozvrhování ET a TT úloh. Zjistili jsme však, že analýza rozvrhovatelnosti prezentovaná autory Nasri a Brandenburg není exaktní. První část diplomové práce je zaměřena na analýzu rozvrhovatelnosti, která rovněž sestrojuje rozvrhovací graf, ale je zároveň exaktní. Experimentální vyhodnocení navíc ukazuje, že naše analýza rozvrhovatelnosti je výrazně rychlejší.

Druhá část diplomové práce popisuje heuristický algoritmus, který hledá platnou množinu časů pro zahájení TT úloh tak, aby byla zajištěna rozvrhovatelnost pro TT i ET úlohy. V experimentálním vyhodnocení tento algoritmus našel řešení v 97.8 % případů, kdy řešení existovalo. Navíc byl obecně schopný vyřešit instance s 20 TT a 20 ET úlohami během několika sekund.

Klíčová slova: analýza rozvrhovatelnosti, kolísání doby vydání, kolísání doby zpracování, online a offline rozvrhování, rozvrhovací graf

Abstract

The time-triggered (TT) systems provide reliable behavior, while the event-triggered (ET) systems provide higher flexibility and make use of the bandwidth more efficiently. To attain the advantages of both, this thesis formulates a framework for combining ET and TT scheduling of non-preemptive tasks on a uniprocessor.

We believe that the approach based on schedule graph generation, proposed by Nasri and Brandenburg [RTSS 2017, pp. 12–23], is well suited for combining ET and TT scheduling. However, we found out that the schedulability analysis presented by Nasri and Brandenburg is not exact. The first part of the thesis focuses on a schedulability analysis that also constructs the schedule graph but is exact. Additionally, the experimental evaluation shows that our schedulability analysis is substantially faster.

In the second part of the thesis, we propose a heuristic algorithm that searches for a valid set of start times for TT tasks while ensuring schedulability for both TT and ET tasks. In an experimental evaluation, the heuristic algorithm was able to find a solution in 97.8 % of cases where a solution existed. Additionally, it was generally able to solve instances of 20 TT and 20 ET tasks in a matter of seconds.

Keywords: schedulability analysis, release jitter, execution time variation, online and offline scheduling, schedule graph

Contents

Chapter 1

Introduction

Time-triggered (TT) systems are dependable and robust and can easily detect dropped messages. However, they lack flexibility as run time and start time of TT tasks need to be known in advance. On the other hand, event-triggered (ET) systems are very flexible as they can quickly react to unknown events. The downside of ET systems is they do not guarantee that a task will be completed in a certain time.

In TT systems, the tasks are scheduled offline, and the start time of each task is predetermined. On the other hand, tasks in ET systems are scheduled online (dynamically), and the start time of each task is unknown a priori.

1.1 Related work

The combination of TT and ET systems may keep the advantages of both systems. This has been explored mostly for distributed automotive networks [1][17][8]. Specialized protocols which combine TT and ET scheduling such as FTT-CAN [2][16][9] or FlexRay [18] have been developed.

The combination of TT and ET systems has also been explored in interprocessor communication [11] and for wireless communication [19].

These approaches do not assume prior information about the release and execution time of ET tasks and many of them focus on preemptive scheduling. Assumption of prior information about the execution time of ET tasks was partly explored in [7] which combines periodic, aperiodic, and sporadic tasks. The tasks are however preemptive and the analysis is only heuristic.

Scheduling of non-preemptive tasks with release jitter and execution time variation may exhibit anomalies where the worst-case scenario does not result in a deadline miss but a different scenario does. This is discussed in [12] where an exact schedulability test is described. We wish to extend on the ideas presented in [12] to guarantee that our solutions are sustainable. A schedulability test is sustainable if any task deemed schedulable by the test remains schedulable even for a "better" scenario [3].

1.2 This thesis

This thesis formulates a scheduling problem with two types of tasks. The first type is ET tasks with fixed priority, release jitter, and execution time variation. The second type is TT tasks with no release jitter or execution time variation. We assume that an upper and lower bound on the execution time and release time of ET tasks is known a priori.

Both types of tasks are non-preemptive, periodic, and have deadlines. All tasks are executed on a single unary resource (also known as uniprocessor or mono-processor), which means that at most one task can be executed at any time. The goal is to find start times for the TT tasks such that they satisfy their deadlines and do not cause deadline misses for the ET tasks. The ET tasks may also not overlap with each other as well. These start times are computed offline. The ET tasks are scheduled online during the run time along with the TT tasks. In other words, we will do an offline analysis so that the online scheduling may proceed without deadline misses.

The contribution of this thesis is a formal description of the problem, algorithms that solve the problem, and an evaluation of these algorithms. The remaining part of this thesis is organized as follows: Chapter 2 formally describes the problem. Chapter 3 focuses only on the scheduling of ET tasks and the solutions are evaluated in Chapter 4. Likewise, Chapter 5 focuses on the combination of ET and TT tasks, and the solutions are evaluated in Chapter 6. The thesis concludes in Chapter 7.

Chapter 2

Formal problem description

This section gives a formal description of the problem. The ET tasks, their properties, and scheduling is introduced first. Then, TT tasks and their combination with ET tasks is explained. All defined variables in Chapter 2 are integers. All proposed algorithms in later sections also work only with integer values.

For integrity reasons most of the notation, definitions, and terminology regarding ET tasks is kept the same or similar to [12].

2.1 Event-triggered tasks and jobs

Let us consider a set of ET tasks $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_n)$, where each task $\mathcal{E}_i$ is comprised of an earliest release time r_i^{min}, latest release time r_i^{max}, best case execution time c_i^{min}, worst case execution time c_i^{max}, deadline d_i^{ET}, priority p_i, and period τ_i^{ET}. These values are assumed to satisfy the following conditions:

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

$$0 < c_i^{min} \leq c_i^{max}$$

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

$$0 \leq p_i$$

Each ET task $\mathcal{E}_i$ is comprised of a set of jobs $E_{i,1}, \ldots, E_{i,h}$ where $E_{i,j}$ is the j-th job occurrence and h is an index of the last occurrence. The index h depends on other ET tasks and will be discussed shortly. Each ET job $E_{i,j}$ has earliest release time $r_{i,j}^{min}$, latest release time $r_{i,j}^{max}$, best case execution time $c_{i,j}^{min}$, worst case execution time $c_{i,j}^{max}$, deadline $d_{i,j}^{ET}$ and priority $p_{i,j}$. These variables are defined according to the associated ET task thusly:

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

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

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

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

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

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

Simply put, an ET job $E_{i,j}$ inherits execution times and priority from its corresponding ET task $\mathcal{E}_i$ and its release times and deadline shift by τ_i^{ET} $(j-1)$ times.

2.2 Scheduling of event-triggered jobs

The base of the problem is online scheduling of a set of ET jobs defined by a set of ET tasks $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_n)$ on a uniprocessor. The ET jobs repeat until they reach hyperperiod defined as $\eta = LCM(\tau_1^{ET}, \ldots, \tau_n^{ET})$ where LCM is the least common multiple. In other words, each ET task $\mathcal{E}_i$ has η/τ_i^{ET} job occurrences.

During run time, each ET job $E_{i,j}$ is released at a priori unknown time $r_{i,j}^{ET} \in [r_{i,j}^{min}, r_{i,j}^{max}]$ where $r_{i,j}^{ET}$ is an integer. This behavior is referred to as *release jitter*. Once the value $r_{i,j}^{ET}$ is known, ET job $E_{i,j}$ is *released*. If an ET job $E_{i,j}$ is *executed* at time t_e then it occupies the processor during interval $[t_e, t_e + c_{i,j}^{ET})$. Here $c_{i,j}^{ET} \in [c_{i,j}^{min}, c_{i,j}^{max}]$ is an unknown a priori execution time and is also an integer. This behavior is referred to as *execution time variation*. Time t_e is referred to as *time of execution*. If a job $E_{i,j}$ finishes execution at time $t_e + c_{i,j}^{ET} > d_{i,j}^{ET}$, then the online scheduler yields a *deadline miss*.

Additionally, $E_{i,j}$ is *finished* if it has been executed, i.e., it was picked by the online scheduler and then occupied the processor for $c_{i,j}^{ET}$ units of time. An ET job is *unfinished* if it is not finished. An ET job $E_{i,j}$ is *certainly released* at time t if $r_{i,j}^{max} \leq t$ and *possibly released* if $r_{i,j}^{min} \leq t < r_{i,j}^{max}$.

Applicable jobs $E^A = (E_{1,j}, \ldots, E_{k,l})$ is a set of ET jobs that contains all jobs $E_{i,j}$ which satisfy: $E_{i,j}$ is unfinished $\wedge$ $(j = 1 \vee E_{i,j-1}$ is finished). Put differently, applicable jobs contain the first unfinished occurrence of a job from each task. This set will have at most as many elements as there are ET tasks. The set E^A will contain a lower number of elements if there is an ET task for which all ET job occurrences are finished.

2.3 Scheduling policies

Scheduling policy is a function $\mathcal{P}(t, E^A)$ which for a given set of applicable jobs E^A and time t returns a job E_e, which should be scheduled next. We assume that no scheduling policy would return a job that is not applicable if it was given a set of unfinished jobs instead. The policy can also return *null*, which can happen if the set E^A is empty or the policy chooses to wait for an unreleased job to release.

The scheduling policy is invoked at time $t = 0$, after a job has finished execution, or when a job is released and no jobs are currently being executed. It is assumed that the scheduling overhead resulting from the run time of an online scheduler executing a scheduling policy is insignificant and is therefore ignored.

An example of a scheduling policy is earliest deadline first (EDF) which out of all jobs which are applicable and released picks one with the earliest deadline, i.e., smallest $d_{i,j}^{ET}$. If there is no job that is both applicable and released, then the policy returns *null*.

The priority value p of ET tasks and jobs is used only in scheduling policies. Tasks with the lowest value p_i have the highest priority.

2.4 Execution scenario

For a set of ET tasks $\mathcal{E} = (\mathcal{E}_i, \ldots, \mathcal{E}_n)$ an *execution scenario* $\gamma = (C, R)$ is a set of execution times $C = (C_1, \ldots, C_n)$ and release times $R = (R_1, \ldots, R_n)$ where $C_i = (c_{i,1}^{ET}, \ldots, c_{i,h}^{ET})$, $R_i = (r_{i,1}^{ET}, \ldots, r_{i,h}^{ET})$, $c_{i,j}^{ET} \in [c_{i,j}^{min}, c_{i,j}^{max}]$ and $r_{i,j}^{ET} \in [r_{i,j}^{min}, r_{i,j}^{max}]$. In other words, an execution scenario specifies the release time and execution time for every ET job from $\mathcal{E}$.

A set of ET tasks $\mathcal{E}$ is *schedulable* under policy $\mathcal{P}$ if there exists no execution scenario resulting in a deadline miss for an online scheduler that uses policy $\mathcal{P}$. In other words, if the online scheduler used policy $\mathcal{P}$ and had to schedule jobs from $\mathcal{E}$, then this could not result in a deadline miss.

It is assumed that all tasks have unique identification numbers which are used in scheduling policies to break ties so that an execution scenario is deterministic.

2.5 Time-triggered tasks and jobs

Just like ET tasks, TT tasks $\mathcal{T} = (\mathcal{T}_1, \ldots, \mathcal{T}_m)$ are periodic but they have different properties. Each TT task $\mathcal{T}_i$ consists of a release time r_i^{TT}, execution time c_i^{TT}, deadline d_i^{TT}, and period τ_i^{TT}. These values are assumed to satisfy the following conditions:

$$\begin{gathered} 0 \leq r_i^{TT} \\ 0 < c_i^{TT} \\ r_i^{TT} + c_i^{TT} \leq d_i^{TT} \leq \tau_i^{TT} \end{gathered}$$

Each TT task $\mathcal{T}_i$ is comprised of TT jobs $(T_{i,j}, \ldots, T_{i,h})$ where $T_{i,j}$ is the j-th job occurrence and h is an index of the last occurrence. Each TT job $T_{i,j}$ has release time $r_{i,j}^{TT}$, execution time $c_{i,j}^{TT}$ and deadline $d_{i,j}^{TT}$. These variables are defined thusly:

$$\begin{aligned} r_{i,j}^{TT} &= r_i^{TT} + (j-1) \cdot \tau_i^{TT} \\ c_{i,j}^{TT} &= c_i^{TT} \\ d_{i,j}^{TT} &= d_i^{TT} + (j-1) \cdot \tau_i^{TT} \end{aligned}$$

2.6 Combining ET and TT tasks

When combining ET tasks and TT tasks, the aim is to find start times $S = (S_1, \ldots, S_m)$ where m is the number of TT tasks and $S_i \in [r_i^{TT}, d_i^{TT} - c_i^{TT}]$ such that when scheduling

variable name	$\mathcal{E}_i$	$E_{i,j}$	$\mathcal{T}_i$	$T_{i,j}$
earliest release time	r_i^{min}	$r_{i,j}^{min}$	–	–
latest release time	r_i^{max}	$r_{i,j}^{max}$	–	–
release time	–	$r_{i,j}^{ET}$	r_i^{TT}	$r_{i,j}^{TT}$
best case execution time	c_i^{min}	$c_{i,j}^{min}$	–	–
worst case execution time	c_i^{max}	$c_{i,j}^{max}$	–	–
execution time	–	$c_{i,j}^{ET}$	c_i^{TT}	$c_{i,j}^{TT}$
deadline	d_i^{ET}	$d_{i,j}^{ET}$	d_i^{TT}	$d_{i,j}^{TT}$
priority	p_i	$p_{i,j}$	–	–
period	τ_i^{ET}	–	τ_i^{TT}	–

Table 2.1: Summary of the ET and TT task and job notation.

a set of ET jobs, each TT job $T_{i,j}$ is always executed at time $S_i + \tau_i^{TT} \cdot (j-1)$ and the resulting schedule does not yield a deadline miss for any execution scenario. A TT job $T_{i,j}$ is *finished* once it completes its execution at time $S_i + \tau_i^{TT} \cdot (j-1) + T_{i,j}.c$.

In other words, the aim is to find a start time for each TT task that specifies the time each TT job will execute and the online scheduler will never yield a deadline miss for any ET job. Also, note that an ET job cannot be interrupted so that a TT job can begin execution at its predetermined start time. If a TT job does not get executed at its predetermined start time, then the online scheduler yields a deadline miss.

The hyperperiod is defined as $\eta = LCM(\tau_1^{ET}, \ldots, \tau_n^{ET}, \tau_1^{TT}, \ldots, \tau_m^{TT})$ when combining ET tasks with TT tasks. If a set of start times S does solve the problem, then the set is called *valid start times*. This problem may not have a solution, for instance when the set of ET tasks is not schedulable.

2.7 Summary and examples

A summary of the notation can be seen in Table 2.1. Note that $r_{i,j}^{ET}$ and $c_{i,j}^{ET}$ are both unknown a priori unlike all other values and are revealed during run time execution.

To give a better understanding of the discussed concepts, we present two examples. The first example consists only of ET tasks and the goal is to determine if the tasks are schedulable. The second example combines ET and TT tasks and the goal is to find a set of valid start times.

2.7.1 Example 1 - ET schedulability test

Let us consider a set of ET tasks $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3)$ and no TT tasks. The parameters of these ET tasks can be seen in Table 2.2.

In this example, we will use the EDF scheduling policy that does not make use of

	r^{min}	r^{max}	c^{min}	c^{max}	d^{ET}	τ^{ET}
$\mathcal{E}_1$	2	5	5	7	16	20
$\mathcal{E}_2$	1	1	2	4	8	10
$\mathcal{E}_3$	0	0	1	1	5	5

Table 2.2: Parameters of ET tasks $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3)$.

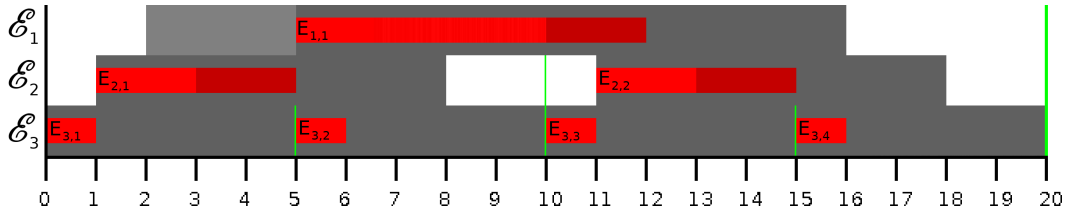


Figure 2.1: A loose Gantt chart of ET tasks $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3)$. Each row represents a different ET task. The top row represents task $\mathcal{E}_1$, the middle row represents task $\mathcal{E}_2$ and the bottom row represents task $\mathcal{E}_3$.

priority values p and they are therefore undefined.

The hyperperiod for this instance is $\eta = LCM(20, 10, 5) = 20$. This means that ET tasks $\mathcal{E}_1, \mathcal{E}_2$, and $\mathcal{E}_3$ will have 1, 2, and 4 jobs respectively. Individual jobs are denoted as $\mathcal{E}_1 = (E_{1,1})$, $\mathcal{E}_2 = (E_{2,1}, E_{2,2})$ and $\mathcal{E}_3 = (E_{3,1}, E_{3,2}, E_{3,3}, E_{3,4})$.

The instance is visualized in Figure 2.1 but bear in mind that this visualization displays only defined properties of ET tasks $\mathcal{E}$ and says nothing about the used policy or when a job executes. This is due to the fact that in our case, the time of execution of each job may vary based on an execution scenario. Despite this, the visualization still provides some insight into the instance. We call this type of visualization a *loose Gantt chart*. Because the time of execution of each job is unknown a priori, each job is placed at its maximal release time.

In the following Gantt chart visualizations, gray areas show where a job can proceed with the execution. Release jitter is displayed using bright gray color. Minimal and maximal execution times are displayed using bright and dark red colors. Green lines denote periods.

By using an execution scenario and a scheduling policy, the instance can be visualized using a regular Gantt chart. Let us consider the EDF scheduling policy and execution scenario in which all jobs have the worst execution times and latest release times. This means that:

$$c_{1,1}^{ET} = 7, r_{1,1}^{ET} = 5$$

$$c_{2,1}^{ET} = c_{2,2}^{ET} = 4, r_{2,1}^{ET} = 1, r_{2,2}^{ET} = 11$$

$$c_{3,1}^{ET} = c_{3,2}^{ET} = c_{3,3}^{ET} = c_{3,4}^{ET} = 1, r_{3,1}^{ET} = 0, r_{3,2}^{ET} = 5, r_{3,3}^{ET} = 10, r_{3,4}^{ET} = 15$$

A Gantt chart using this scenario and the EDF scheduling policy is visualized in Figure 2.2. This execution scenario does not result in a deadline miss. However, this does not mean that the ET tasks $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3)$ are schedulable.

Let us consider a different execution scenario where ET job $E_{1,1}$ releases at $r_{1,1}^{ET} = r_{1,1}^{min}$ instead of $r_{1,1}^{max}$ and execution time of ET job $E_{2,1}$ is $c_{2,1}^{ET} = c_{2,1}^{min}$ instead of $c_{2,1}^{max}$.

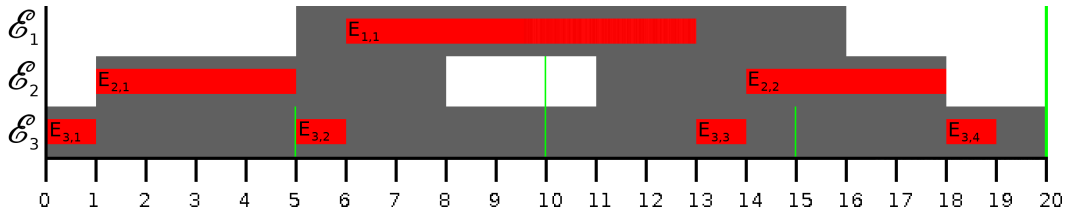


Figure 2.2: A Gantt chart of ET tasks $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3)$ assuming the worst case scenario.

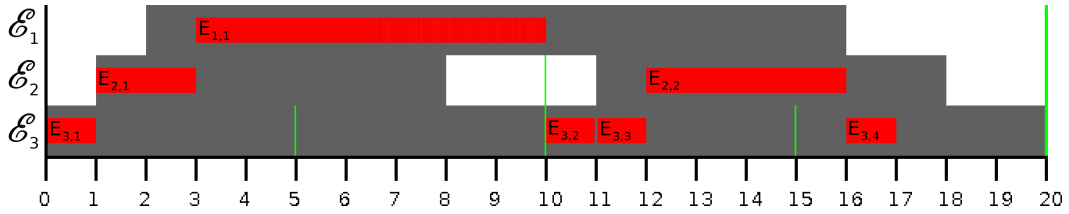


Figure 2.3: A Gantt chart of ET tasks $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3)$ with execution scenario that results in a deadline miss.

This execution scenario results in a deadline miss for ET job $E_{3,2}$ as can be seen in Figure 2.3.

This example shows that we cannot simply consider only the worst-case scenario when testing schedulability and our tests, therefore, aim to be sustainable. [3]

2.7.2 Example 2 - finding valid start times

In this example we have one ET task $\mathcal{E}_1$ and one TT task $\mathcal{T}_1$. These are defined as:

$$r_1^{TT} = 2, c_1^{TT} = 4, d_1^{TT} = 8, \tau_1^{TT} = 10$$

$$r_1^{min} = r_1^{max} = 0, c_1^{min} = 2, c_1^{max} = 3, d_1^{ET} = 5, \tau_1^{ET} = 5$$

The tasks are visualized in Figure 2.4. Unlike in the first example, our goal is to find a valid set of start times. Here we have only one TT task, therefore we have to find only a single start time $S_1 \in [2, 4]$. Because S_1 is an integer, the only possible values of S_1 are 2, 3, and 4.

Let us first look if $S_1 = 2$ is viable. In an execution scenario where ET job $E_{1,1}$ has $c_{1,1}^{ET} = 3$, the TT job $T_{1,1}$ cannot start at the determined start time and therefore the start time is not valid. This is illustrated in Figure 2.5.

Now let us consider $S_1 = 3$. The ET job $E_{1,1}$ no longer interferes with $T_{1,1}$. To prove

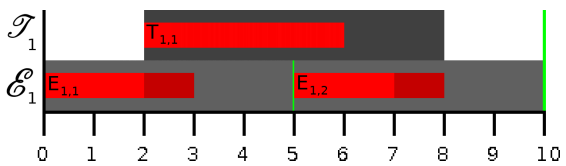


Figure 2.4: A loose Gantt chart of ET task $\mathcal{E}_1$ and TT task $\mathcal{T}_1$. The top row represents $\mathcal{T}_1$ and the bottom row represents $\mathcal{E}_1$.

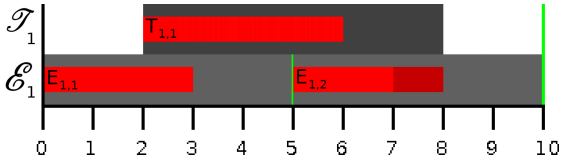


Figure 2.5: A loose Gantt chart of ET task $\mathcal{E}_1$ and TT task $\mathcal{T}_1$. Here $c_{1,1}^{ET}$ is set to 3 to show that the start time $S_1 = 2$ is not valid.

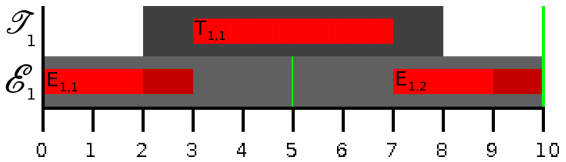


Figure 2.6: A loose Gantt chart of ET task $\mathcal{E}_1$ and TT task $\mathcal{T}_1$. As can be seen, $E_{1,1}$ will never overlap with $T_{1,1}$ and $T_{1,1}$ will never overlap with $E_{1,2}$. Start time $S_1 = 3$ is therefore valid.

that the start time $S_1 = 3$ is valid, each execution scenario has to be accounted for. For brevity, we show that the start time $S_1 = 3$ is valid using Figure 2.6.

Start time $S_1 = 4$ is not valid as it would cause a deadline miss for job $E_{1,2}$ in an execution scenario where $c_{1,2}^{ET} = 3$. Therefore the only valid set of start times is $S = (S_1) = (3)$.

Chapter 3

ET solutions

Before trying to solve the problem of finding start times for TT jobs, we will first focus only on the schedulability of a set of ET tasks.

This section describes more scheduling policies and then algorithms used for testing the schedulability of ET tasks under these policies. A test deciding the schedulability of a set of ET tasks will be referred to as an ET schedulability test.

The following pseudocodes use OOP notation. For instance a deadline $d_{i,j}^{ET}$ of an ET job $E_{i,j}$ will be simply denoted as $E_{i,j}.d$.

3.1 Scheduling policies

This section describes more scheduling policies and provides their pseudocodes.

3.1.1 Earliest Deadline First (EDF)

We have already introduced the EDF scheduling policy in Chapter 2 as an example. A pseudocode of the EDF scheduling policy can be seen in Algorithm 1. Note that this policy does not make use of the priority values p.

Algorithm 1 EDF scheduling policy

Input: Time t and applicable jobs E^A
Output: Selected job E_e or *null*

```
1: function EDF_POLICY(t,E^A)
2:     E^R ← subset of E^A with only released jobs
3:     if E^R = ∅ then
4:         return null
5:     return job from E^R with the lowest deadline
```

3.1.2 EDF - Fixed Priority (EDF-FP)

The EDF-FP is a version of the EDF policy that uses priority value p of ET jobs. If multiple jobs E have the same lowest value p, the one with the lowest deadline is chosen.

A pseudocode of the EDF-FP policy can be seen in Algorithm 2. Line 5 returns the same result as sorting all jobs E^R in lexicographical order primarily by p in ascending order, secondarily by d in ascending order, and then picking the first job.

Algorithm 2 EDF-FP scheduling policy

Input: Time t and applicable jobs E^A
Output: Selected job E_e or *null*

```
1: function EDF-FP_POLICY(t,E^A)
2:     E^R ← subset of E^A with only released jobs
3:     if E^R = ∅ then
4:         return null
5:     return job from E^R with the lowest p value, then the lowest d value
```

By using priority values p, the resulting schedule may execute jobs with higher priority earlier. Another useful property of the EDF-FP policy is that it can be easily changed to the EDF policy by setting the priority of all jobs to the same value. It can also be changed to *Fixed Priority policy* (FP policy), which schedules jobs based only on priority values p. This is done by setting unique priority values p for each task. Due to these advantages, we use this as the go-to scheduling policy in the following policies.

3.1.3 Work-conserving and non-work-conserving policies

The EDF, FP, and EDF-FP scheduling policies are *work-conserving* policies, which means that they never idle when there is at least one released and unfinished job. Although work-conserving schedulers finish jobs earlier, a set of ET tasks may be schedulable under a non-work-conserving policy while not being schedulable under a work-conserving policy. This is due to the fact that non-work-conserving schedulers insert idle times between jobs to avoid deadline misses. We will now describe three non-work-conserving policies.

3.1.4 Precautious Rate-Monotonic (P-RM)

Under the P-RM scheduling policy, a job with a priority different than the highest one cannot be scheduled if it may cause a deadline miss for some job with the highest priority. The highest priority job is defined by having the lowest r^{max} out of all jobs for which $p = 0$. [13]

The highest priority job is denoted as E^c in the pseudocode of the P-RM scheduling policy, which can be seen in Algorithm 3. Jobs that are released, applicable, and do not cause a deadline miss to E^c are called *viable jobs* and are denoted as E^V.

Unlike in [13], our version of the P-RM policy follows the EDF-FP policy for jobs that can be scheduled without causing a deadline miss for the highest priority job.

Additionally, we define the highest priority as $p = 0$.

Algorithm 3 P-RM scheduling policy

Input: Time t and applicable jobs E^A
Output: Selected job E_e or *null*

1: **function** P-RM_POLICY(t,E^A)
2: $E^c \leftarrow$ job from E^A with $p = 0$ and the lowest r^{max}
3: **if** E^c is *null* **then**
4: **return** EDF-FP_POLICY(t,E^A)
5: $t^c \leftarrow E^c.d - E^c.c^{max}$
6: $E^R \leftarrow$ subset of E^A with only released jobs
7: $E^V \leftarrow \{E_{i,j} | E_{i,j} \in E^R \wedge (t + E_{i,j}.c^{max} \leq t^c \ \vee E_{i,j} = E^c)\}$
8: **if** $E^V = \emptyset$ **then**
9: **return** *null*
10: **return** job from E^V with the lowest p value, then the lowest d value

3.1.5 Critical Point (CP)

The CP scheduling policy follows the EDF-FP scheduling policy but does not schedule a job if it will cause a deadline miss to an unfinished job with the lowest deadline. This job is denoted as E^c and is called *critical job*. The pseudocode of the CP scheduling policy can be seen in Algorithm 4.

Algorithm 4 CP scheduling policy

Input: Time t and applicable jobs E^A
Output: Selected job E_e or *null*

1: **function** CP_POLICY(t,E^A)
2: **if** $E^A = \emptyset$ **then**
3: **return** null
4: $E^c \leftarrow$ job from E^A with the lowest d value
5: $t^c \leftarrow E^c.d - E^c.c^{max}$
6: $E^R \leftarrow$ subset of E^A with only released jobs
7: $E^V \leftarrow \{E_{i,j} | E_{i,j} \in E^R \wedge (t + E_{i,j}.c^{max} \leq t^c \ \vee E_{i,j} = E^c)\}$
8: **if** $E^V = \emptyset$ **then**
9: **return** *null*
10: **return** job from E^V with the lowest p value, then the lowest d value

This policy is similar to P-RM in the sense that both define a critical job and no other job can be executed if it will cause a deadline miss for the critical job. The only difference between the two policies is how the critical job is selected and that the P-RM policy may not have a critical job while the CP policy always has a critical job if E^A is not empty.

3.1.6 Critical Window (CW)

The CW policy differs from the CP policy by determining its critical time using all applicable jobs instead of a single job. Its pseudocode can be seen in Algorithm 5. Note that on line 4, the set of jobs E^S is an ordered set of jobs, which is relevant in the for cycle on line 6. This scheduling policy is inspired by Critical Window EDF from [12].

Algorithm 5 CW scheduling policy

Input: Time t and applicable jobs E^A
Output: Selected job E_e or *null*

```
1:  function CW_POLICY(t,E^A)
2:      if E^A = ∅ then
3:          return null
4:      E^S ← E^A sorted by d in descending order
5:      t^c ← ∞
6:      for each E_{i,j} ∈ E^S do
7:          if E_{i,j}.d < t^c then
8:              t^c ← E_{i,j}.d − E_{i,j}.c^max
9:          if E_{i,j}.d ≥ t^c then
10:             t^c ← t^c − E_{i,j}.c^max
11:     E^c ← the last job in E^S
12:     E^R ← subset of E^A with only released jobs
13:     E^V ← {E_{i,j} | E_{i,j} ∈ E^R ∧ (t + E_{i,j}.c^max ≤ t^c ∨ E_{i,j} = E^c)}
14:     if E^V = ∅ then
15:         return null
16:     return job from E^V with the lowest p value, then the lowest d value
```

3.2 Brute force ET schedulability test

The most simple approach for implementing an ET schedulability test is a brute force algorithm. The brute force algorithm simulates the work of an online scheduler for every execution scenario.

3.2.1 Advantages and disadvantages

The main benefit of this approach is its ease of implementation. This does not only concern the algorithm itself but also its variations for different scheduling policies.

Its only disadvantage is that it greatly suffers from combinatorial explosion. For instance, let us consider 10 ET jobs each with $r^{min} = 5$, $r^{max} = 10$, $c^{min} = 5$ and $c^{max} = 10$. Each job has $6 \cdot 6 = 36$ unique combinations of release and execution times. For 10 ET jobs, this means a total of $36^{10} \approx 3,66 \cdot 10^{15}$ unique combinations. The brute force algorithm still has a practical use in empirical verification, which will be discussed later.

3.2.2 Pseudocode

A pseudocode of the algorithm can be seen in Algorithm 6. In this case, the algorithm uses the EDF-FP scheduling policy as can be seen on line 16. However, it can be replaced with any other scheduling policy function and the algorithm will still be correct. The pseudocode consists of two functions. Function BRUTE_FORCE_TEST receives a set of tasks $\mathcal{E}$ and for each execution scenario assigns release and execution times to all jobs. After each assignment, BRUTE_FORCE_TEST executes the SIMULATE_SCHEDULING function which determines if the given scenario would result in a deadline miss on an online scheduler.

Algorithm 6 Brute force ET schedulability test

Input: Set of ET tasks $\mathcal{E}$
Output: if the instance is schedulable

```
1:  function BRUTE_FORCE_TEST(E)
2:      for each unique execution scenario γ = (C, R) do
3:          for each job E_{i,j} ∈ E do
4:              set E_{i,j}.c and E_{i,j}.r according to γ
5:          success ← SIMULATE_SCHEDULING(E)
6:          if ¬success then
7:              return false
8:      return true
9:  function SIMULATE_SCHEDULING(E)
10:     t ← 0
11:     set all jobs from E to unfinished
12:     while true do
13:         E^A ← applicable jobs from E
14:         if E^A = ∅ then
15:             return true          ▷ Every job is finished with no deadline misses
16:         E_e ← EDF-FP_POLICY(t,E^A)
17:         if E_e is null then
18:             t ← min{E_{i,j}.r | E_{i,j} ∈ E^A ∧ E_{i,j}.r > t}
19:             continue                          ▷ No jobs are released, wait
20:         if E_e.c + t > E_e.d then
21:             return false                                  ▷ Deadline miss
22:         t ← t + E_e.c
23:         set E_e to finished
```

The SIMULATE_SCHEDULING function can set jobs to be finished or unfinished (such as on line 11). Although maybe obvious, what is important to note is that the SIMULATE_SCHEDULING function does not actually occupy a processor for $E_{i,j}.c$ units of time when simulating execution of a job $E_{i,j}$ and instead increases its local t variable.

An interesting thing of note is that using $E_e.c^{max}$ instead of $E_e.c$ on line 20 gives us the same result faster. The pseudocode uses $E_e.c$ to be more illustrative of the idea behind the algorithm, which is testing every scheduling scenario individually.

3.3 Schedule graph introduction

In 2017 Mitra Nasri and Björn B. Brandenburg introduced a scalable algorithm that implements an ET schedulability test using a so-called *schedule graph* in their paper "An Exact and Sustainable Analysis of Non-Preemptive Scheduling". This graph encapsulates all possible execution scenarios and merges similar scenarios while keeping the schedulability test exact. [12] We believe that this solution is not exact for non-work-conserving policies. We describe this issue in more detail in Appendix A.

This section describes a schedule graph generation algorithm that is heavily inspired by [12] and which is exact even for non-work-conserving policies. First, we focus on the basics of the schedule graph. We then provide a rough description of the generation algorithm and an example. The precise rules for schedule graph generation are described afterward.

3.3.1 Schedule graph

An intuitive understanding of the schedule graph is that each vertex v_i contains a set of finished jobs and a range of times $[e_i, l_i]$. The schedule graph is a directed graph where an edge characterizes the execution of an ET job. A job can finish in a range of times for instance due to execution time variation.

The graph starts with a root vertex v_r (sometimes also denoted as v_0) with no finished jobs and a range of times $[0, 0]$. The schedule graph is gradually built from the root vertex up to a graph that encapsulates all possible sequences of job executions.

Formally, the schedule graph $\mathcal{G} = (V, \Sigma)$ is defined as a directed acyclic graph where each vertex has a label consisting of earliest finish time e (EFT) and latest finish time l (LFT). These two variables define a *finish time interval* $[e, l]$. Furthermore, each edge $\sigma \in \Sigma$ has a label corresponding to a single job that is denoted as $\sigma.E$. Multiple edges can have the same job label.

Usually, a set of edges is denoted as E, however E already denotes a set of ET jobs, so Σ is chosen instead. Additionally, because e already denotes EFT, an edge is denoted as σ.

It is useful to see the schedule graph as a directed level-structured graph. This means V can be split into disjoint sets based on the distance from root vertex v_r. [5] Here, distance of a vertex v_i from the root vertex v_r is the length of the shortest path from v_r to v_i. Let V_i denote set of vertices for which the distance from root vertex v_r is i. Note that $V_0 = \{v_r\}$.

3.3.2 Graph generation

The schedule graph is generated based on a set of ET tasks $\mathcal{E}$ and a scheduling policy $\mathcal{P}$. The generation algorithm initializes with root vertex v_r where $[e_r, l_r] = [0, 0]$ and $V_0 = \{v_r\}$. It then keeps performing two alternating phases called *expansion phase* and *merge phase*. The expansion phase generates a new set of vertices V_{i+1}^{ex} from V_i. The merge phase then tries to merge some vertices from V_{i+1}^{ex} with each other which results in V_{i+1}.

	r^{min}	r^{max}	c^{min}	c^{max}	d^{ET}	τ^{ET}
$\mathcal{E}_1$	0	0	1	2	10	10
$\mathcal{E}_2$	0	0	1	1	3	5
$\mathcal{E}_3$	1	3	3	4	9	10

Table 3.1: Parameters of ET jobs used in the schedule graph example.

The algorithm starts with V_0 and keeps executing the two alternating phases until V_j is generated where j is the total number of jobs in $\mathcal{E}$. The algorithm may also end earlier if it finds a deadline miss. If a deadline miss is detected, a set of tasks $\mathcal{E}$ is not schedulable under policy $\mathcal{P}$. If the entire graph is generated with no deadline misses found, then the set of tasks $\mathcal{E}$ is schedulable under policy $\mathcal{P}$.

3.3.3 Expansion phase

Vertex v_i can be characterized by $[e_i, l_i]$ time range and its position in the graph. The position in the graph determines which jobs are finished by the path taken from the root vertex. This set of jobs is denoted as E^{v_i} and can be used to determine applicable jobs for vertex v_i.

As previously mentioned, the expansion phase takes vertices V_i and generates a new set of vertices V_{i+1}^{ex}. The expansion phase evaluates each vertex individually by using its applicable jobs E^A and time interval $[e, l]$. It determines at which time which jobs can be executed using this information. This in turn means generating new vertices and edges. The result is a set of new vertices, some of which may be merged in the merge phase.

3.3.4 Merge phase

The merge phase takes the result of expansion phase V_{i+1}^{ex} and attempts to merge its vertices together. Two vertices v_i and v_j can be merged if $E^{v_i} = E^{v_j} \wedge [e_i, l_i] \cap [e_j, l_j] \neq \emptyset$. If vertices v_i and v_j are merged, then all edges pointing to v_j are reoriented so that they point to v_i, time interval of v_i is updated as $[e_i, l_i] \leftarrow [e_i, l_i] \cup [e_j, l_j]$ and v_j is removed from the graph. Note that the schedule graph is not a multigraph, which means that some edges that are reoriented from v_j to v_i may be removed instead.

3.3.5 Example

Let us consider 3 ET tasks $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3)$. The parameters of these ET tasks can be seen in Table 3.1. We will use the EDF scheduling policy which does not make use of priority values p and they are therefore undefined. The loose Gantt chart of this instance can be seen in Figure 3.1.

The entire schedule graph generation process is visualized in Figure 3.2. The algorithm starts at level 0 with root vertex v_0, $[e_0, l_0] = [0, 0]$ and no jobs finished. At this point, there are two certainly released jobs $E_{1,1}$ and $E_{2,1}$. The expansion phase concludes that the EDF policy would pick $E_{2,1}$ at $t = 0$ as it has earlier deadline than $E_{1,1}$. Job

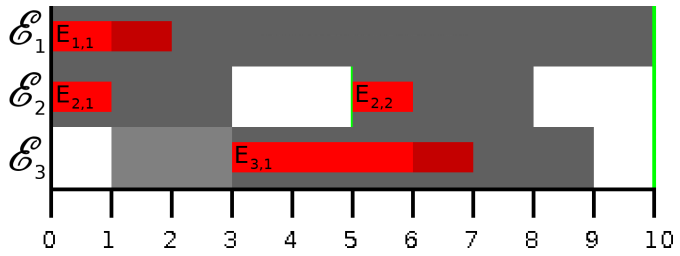


Figure 3.1: Loose Gantt chart of the schedule graph example instance.

$E_{3,1}$ is not considered as it would not be released at $t = 0$ for any execution scenario. The expansion results in creating a new vertex v_1 with $[e_1, l_1] = [e_0 + c_{2,1}^{min}, l_0 + c_{2,1}^{max}] = [1, 1]$. Additionally, new edge pointing from v_0 to v_1 is created with label of the picked job $E_{2,1}$. This concludes the expansion phase on level 0. The merge phase is executed next but it does not change the graph in any way as the result of expansion phase is only one vertex.

The algorithm continues with expansion phase on $V_1 = (v_1)$. Here, two scenarios may occur. In the first scenario, job $E_{3,1}$ has just released at $t = 1$. If that is the case, $E_{3,1}$ would be picked by EDF policy as it has earlier deadline compared to $E_{1,1}$. In the second scenario, job $E_{3,1}$ does not release at $t = 1$ and $E_{1,1}$ is picked instead as there are no other released jobs. These two scenarios result in vertices v_2 and v_3 with $[e_2, l_2] = [e_1 + c_{2,1}^{min}, l_1 + c_{2,1}^{max}] = [2, 3]$ and $[e_3, l_3] = [e_1 + c_{3,1}^{min}, l_1 + c_{3,1}^{max}] = [4, 5]$. This is the result of the expansion phase. The merge phase does not merge vertices v_2 and v_3 as $E^{v_2} \neq E^{v_3}$.

At level $V_2 = (v_2, v_3)$, the expansion phase individually evaluates v_2 and v_3. When expanding a vertex which has a different earliest and latest start time, the expansion phase has to check all times in interval $[e, l]$. In the case of v_2, the only jobs left are $E_{3,1}$ and $E_{2,2}$, but $E_{2,2}$ will not release until $t = 5$ and is therefore ignored. At $t = 2$, there are two scenarios. If $E_{3,1}$ releases at $t = 2$, then it would be picked by EDF policy. If $E_{3,1}$ releases at $t = 3$, then EDF policy would return null and the scheduler would idle for one time unit. At $t = 3$ the job $E_{3,1}$ is certainly released and would be picked by EDF policy. To summarize, EDF will always pick job $E_{3,1}$ in time interval [2,3]. The earliest time it would execute is $t = 2$ and the latest time is $t = 3$. This results in a new vertex v_4 with $[e_4, l_4] = [e_2 + c_{3,1}^{min}, l_2 + c_{3,1}^{max}] = [5, 7]$.

The expansion phase is not done yet with level V_2 because it needs to evaluate v_3 as well. Vertex v_3 has finish time interval $[e_3, l_3] = [4, 5]$. At time $t = 4$, job $E_{1,1}$ would be picked by EDF because $E_{2,2}$ is not yet released and there are no other unfinished jobs. However, at time $t = 5$ job $E_{2,2}$ is certainly released and because it has lower deadline than $E_{1,1}$, it would be picked by EDF instead. This results in two new vertices v_5 and v_6 where $[e_5, l_5] = [4 + c_{2,1}^{min}, 4 + c_{2,1}^{max}] = [5, 6]$ and $[e_6, l_6] = [5 + c_{2,2}^{min}, 5 + c_{2,2}^{max}] = [6, 6]$. Note that because different jobs would execute at different times, the new finish time intervals of v_5 and v_6 are computed by the earliest and latest time they would be picked by the EDF policy.

With the expansion phase done with level V_2, the merge phase takes over. Because $E^{v_4} = E^{v_5}$ and $[e_4, l_4] \cap [e_5, l_5] \neq \emptyset$ vertices v_4 and v_5 are merged. This is done by changing the parameters of v_4, redirecting edges pointing to v_5 to v_4 and removing v_5. In this case one edge is redirected and the resulting interval is $[e_4, l_4] = [5, 7] \cup [5, 6] = [5, 7]$.

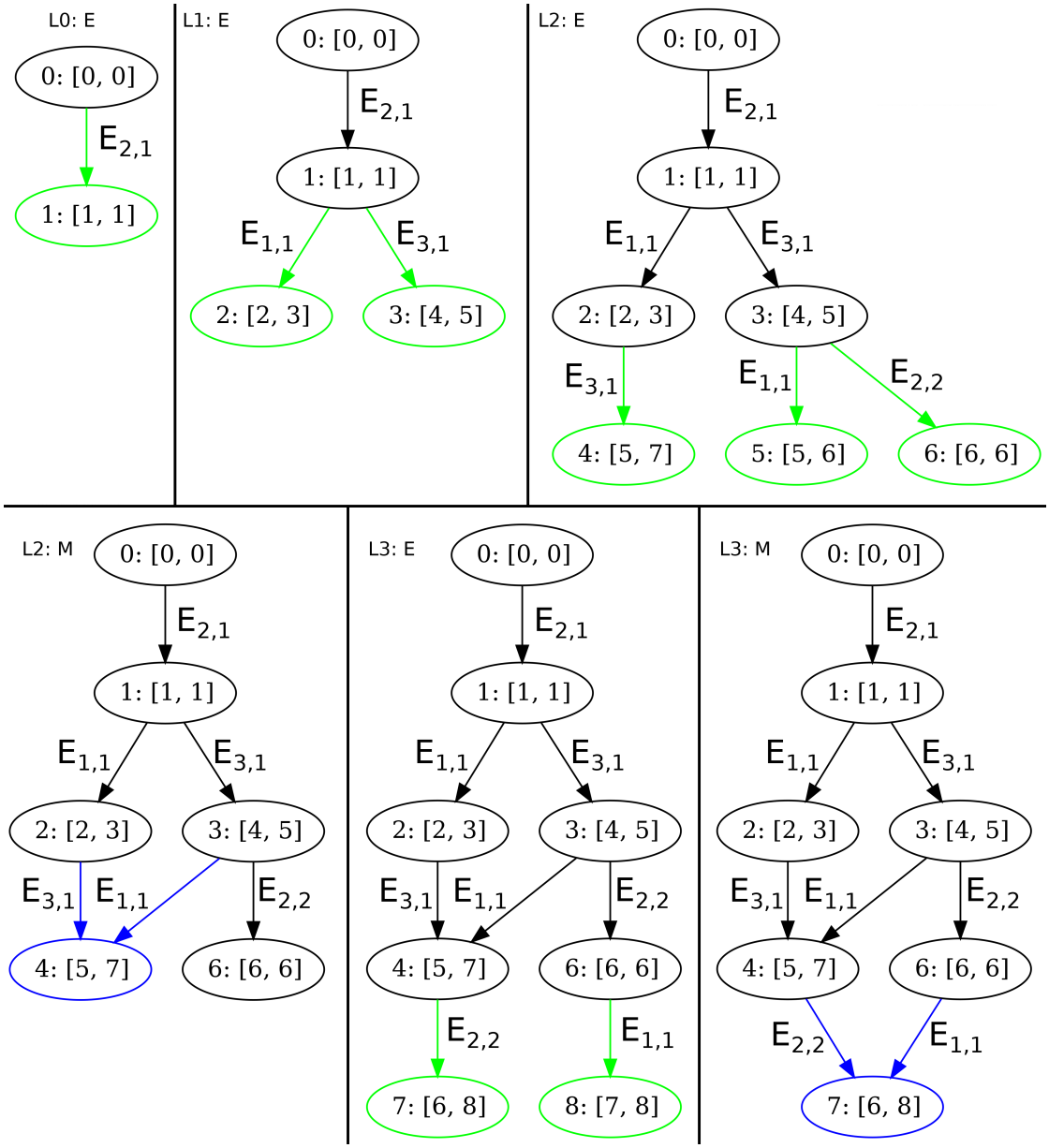


Figure 3.2: The process of generating the schedule graph. Inside of each vertex is its index and finish time interval $[e, l]$. The stages are annotated with level and type of phase (either E for expansion or M for merge). The merge phase on levels 0 and 1 does not change the graph and is therefore omitted from the visualization.

On level $V_3 = (v_4, v_6)$, the final iteration of expansion and merge phase starts. The expansion phase individually evaluates v_4 and v_6. For both of these vertices, only one unfinished job remains. For v_4 the last remaining job is $E_{2,2}$ which releases at $t = 5$ and for v_6 the last remaining job is $E_{1,1}$ which released at $t = 0$. The expansion phase creates vertex v_7 as expansion of v_4 and v_8 as expansion of v_6. The resulting finish time intervals are $[e_7, l_7] = [e_4 + c^{min}_{2,2}, l_4 + c^{max}_{2,2}] = [6, 8]$ and $[e_8, l_8] = [e_6 + c^{min}_{1,1}, l_6 + c^{max}_{1,1}] = [7, 8]$. Because $E^{v_7} = E^{v_8}$ and $[e_7, l_7] \cap [e_8, l_8] \neq \emptyset$, vertices v_7 and v_8 are merged.

There was no deadline miss in this example to show the complete construction of a schedule graph. A deadline miss can be detected during the expansion phase. When creating a new vertex v_i from vertex v_j with job $E_{k,l}$, there is a deadline miss if $v_i.l > E_{k,l}.d$.

3.4 A formal description of schedule graph generation for work-conserving policies

We first describe schedule graph generation rules only for work-conserving policies. We discuss generation rules for non-work-conserving policies in a later section.

3.4.1 Parameters of vertices and edges

Before we describe the rules and algorithms, let us define parameters of each vertex and edge. Every vertex v has earliest finish time $v.e$, latest finish time $v.l$, incoming edges $v.in$ and outgoing edges $v.out$. Every edge σ has job label $\sigma.E$, source vertex $\sigma.s$, and a destination vertex $\sigma.d$.

As mentioned in previous sections, we can determine the applicable jobs of a vertex v from its position in the graph. This is done by taking any path from root vertex v_r to the vertex v and then transforming the set of path's edges into a set of jobs by taking the label of each edge. This set is then equivalent to a set of finished jobs which can be used to determine the applicable jobs. Applicable jobs of a vertex v will be denoted as $v.E^A$.

3.4.2 Job eligibility

A job $E_{i,j}$ has higher *policy priority* than $E_{k,l}$ if $\mathcal{P}(\infty, \{E_{i,j}, E_{k,l}\})$ returns job $E_{i,j}$, where $\mathcal{P}$ is the used policy function. This means that $E_{i,j}$ would be picked by the scheduling policy given that both $E_{i,j}$ and $E_{k,l}$ are certainly released. If one of the jobs is null, then the other one is automatically picked. Behavior for both jobs being null is undefined. The policy function $\mathcal{P}(\infty, \{E_{i,j}, E_{k,l}\})$ can be replaced for instance with EDF-FP_POLICY$(\infty, \{E_{i,j}, E_{k,l}\})$ if we wished to use the EDF-FP policy.

For a given set of applicable jobs E^A 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 and there is no other certainly released applicable job at time t with higher policy priority than E^t_{ce}. Formally, job $E^t_{ce} \in E^A$ is certainly-eligible at time t iff

$$E^t_{ce}.r^{max} \leq t \wedge \nexists E_{i,j} \in E^A \; st. \; E_{i,j} \neq E^t_{ce} \wedge E_{i,j}.r^{max} \leq t \wedge E_{i,j} = \mathcal{P}(\infty, \{E_{i,j}, E^t_{ce}\})$$

Note that there may be at most one certainly-eligible job at time t. If there is no certainly-eligible job at time t, we say that E^t_{ce} does not exist.

Similarly, ET job E^t_{pe} is *possibly-eligible* at time t if it is possibly released at time t and the E^t_{ce} has lower policy priority than E^t_{pe}. Formally, we define the set of possibly-eligible jobs at time t as

$$\{E_{i,j} \mid E_{i,j} \in E^A \ \wedge \ E_{i,j}.r^{min} \leq t < E_{i,j}.r^{max} \ \wedge \ E_{i,j} = \mathcal{P}(\infty, \{E_{i,j}, E^t_{ce}\})\}$$

3.4.3 Explanation of job eligibility

The certainly-eligible job E^t_{ce} is the job that would be picked by the policy function in a scenario where all possibly-eligible jobs are released at time $t+1$ or later. This is due to the fact that possibly-eligible jobs are not released at time t and jobs that are not certainly-eligible or possibly-eligible at time t either have a lower policy priority than E^t_{ce} or cannot be released at time t.

If some of the possibly-eligible jobs at time t do release at time t or earlier, then one of these jobs is picked by the policy function because it has higher policy priority than the certainly-eligible job E^t_{ce}. However, we do not know in advance which possibly-eligible jobs will release at time t or earlier. Any of the possibly-eligible jobs may release at time t or earlier while all other possibly-eligible jobs release at time $t+1$ or later. This means that each of the possibly-eligible jobs may be picked by the policy function in some scenario.

Given a set of applicable jobs $v.E^A$ and a work-conserving scheduling policy, jobs that are certainly-eligible or possibly-eligible at time $t \in [v.e, l_{ext}]$ where $l_{ext} = min\{t \mid t \geq v.l \wedge E^t_{ce}\ exists\}$ may begin execution in some execution scenario at time t. The reason we are not using interval $[v.e, v.l]$ is due to one exception where there are no certainly released jobs in the interval $[v.e, v.l]$, i.e., there might be a scenario, where the online scheduler does not have any available jobs and needs to wait. Because of this fact, we are extending the interval to a time where a certainly-eligible job exists.

3.4.4 Expansion phase

The goal of the expansion phase is to find times, where each job is either possibly or certainly eligible and expand vertices based on this information. A pseudocode of the expansion phase can be seen in Algorithm 7.

What needs a more detailed explanation is the idea of converting integers into ranges as can be seen on line 14 in Algorithm 7. Example of converting integers into ranges is {2,3,4,5,7,8,10,14,15,16} being converted into {[2,5], [7,8], [10,10], [14,16]}. The integers may not be sorted prior to conversion, but we assume that they are sorted beforehand.

Due to the properties of work-conserving policies, there will always be only one range in variable t_R. In addition, the left boundary of this range *est* will always be equal to $max(v.e, E_{i,j}.r^{min})$. These two rules do not apply to non-work-conserving policies.

Algorithm 7 Expansion phase

Input: set of vertices V_i
Output: set of vertices V_{i+1}^{ex}

1: **function** EXPANSION_PHASE(V_i)
2: $V_{i+1}^{ex} \leftarrow \emptyset$
3: **for each** vertex $v \in V_i$ **do**
4: $V_n \leftarrow$ NEXT_NODES(v)
5: $V_{i+1}^{ex} \leftarrow V_{i+1}^{ex} \cup \{V_n\}$
6: **return** V_{i+1}^{ex}
7: **function** NEXT_NODES(v)
8: **if** $v.E^A = \emptyset$ **then**
9: **return** $\emptyset$ ▷ All jobs are completed
10: $V_n \leftarrow \emptyset$
11: $l_{ext} \leftarrow min\{t \mid t \geq v.l \wedge E_{ce}^t \; exists\}$
12: **for** $E_{i,j} \in E^A$ **do**
13: $t_{EL} \leftarrow$ all times $t \in [e, l_{ext}]$ where $E_{i,j}$ is certainly or possibly eligible
14: $t_R \leftarrow$ times t_{EL} converted into ranges
15: **for** each range $[est, lst] \in t_R$ **do**
16: $v_n \leftarrow$ EXPAND_VERTEX($v, E_{i,j}, est, lst$)
17: $V_n \leftarrow V_n \cup \{v_n\}$
18: **return** V_n
19: **function** EXPAND_VERTEX(v, E, est, lst)
20: $v_n \leftarrow$ new vertex with $v_n.e = est + E.c^{min}$ and $v_n.l = lst + E.c^{max}$
21: $\sigma_n \leftarrow$ new edge with $\sigma_n.E = E$, $\sigma_n.s = v$ and $\sigma_n.d = v_n$
22: $v_n.in \leftarrow v_n.in \cup \{\sigma_n\}$
23: $v.out \leftarrow v.out \cup \{\sigma_n\}$
24: **return** v_n

3.4.5 Low-level view of the expansion phase

Algorithm 7 provides high-level pseudocode of the expansion phase. The way of finding times during which a job is certainly or possibly eligible on line 13 is not explained. We, therefore, provide another pseudocode of function NEXT_NODES_CONSERVING in Algorithm 8 which is more representative of an actual implementation for work-conserving policies. On line 12, there might not be any certainly released job, in which case E^t_{ce} is null. Line 10 defines the job E^t_{ce} such that it always has a higher policy priority. This means that its parameters are set so that the used policy prioritizes E^t_{ce} over all other jobs. For the EDF-FP and EDF policies, this could be achieved by setting $E^t_{ce}.d = E^t_{ce}.p = -1$.

Algorithm 8 More detailed version of NEXT_NODES (work-conserving)

Input: vertex v
Output: set of vertices V_n

```
1:  function NEXT_NODES_CONSERVING(v)
2:      if v.E^A = ∅ then
3:          return ∅
4:      l_ext ← min{t | t ≥ v.l ∧ E^t_ce exists}
5:      CE ← null
6:      PE ← ∅
7:      V_n ← ∅
8:      for t = v.e to l_ext + 1 do
9:          if t = l_ext + 1 then
10:             E^t_ce ← newly created ET job st. E^t_ce = P(∞, {E^t_ce, E_i,j}) for any job E_i,j ∈ E^A
11:         else
12:             E^t_ce ← certainly-eligible job at time t or null
13:         if E^t_ce ≠ null ∧ E^t_ce ≠ CE then
14:             if CE ≠ null then
15:                 v_n ← EXPAND_VERTEX(v, CE, max(v.e, CE.r^min), t − 1)
16:                 V_n ← V_n ∪ {v_n}
17:             if E^t_ce ∈ PE then
18:                 PE ← PE \ {E^t_ce}
19:             CE ← E^t_ce
20:             for each E_i,j ∈ PE do
21:                 if E_i,j is not possibly eligible at time t then
22:                     v_n ← EXPAND_VERTEX(v, E_i,j, max(v.e, E_i,j.r^min), t − 1)
23:                     V_n ← V_n ∪ {v_n}
24:                     PE ← PE \ {E_i,j}
25:         for each E_i,j ∈ E^A do
26:             if E_i,j is possibly eligible at time t ∧ E_i,j ∉ PE then
27:                 PE ← PE ∪ {E_i,j}
28:     return V_n
```

This pseudocode goes through each integer time $t \in [v_n.e, l_{ext} + 1]$ and notes how the certainly and possibly released jobs change. The certainly-eligible job is in variable

CE and the set of possibly eligible jobs is in variable PE. In each time $t \in [v_n.e, l_{ext}]$ the algorithm finds the certainly-eligible job E^t_{ce} and, if needed, changes job CE and jobs in PE according to E^t_{ce}. Lastly, it adds new possibly-eligible jobs to PE.

When changing job CE or removing some job from PE, the algorithm expands vertex v. One way to explain this behavior is that because a job is no longer certainly or possibly eligible, we now know the time interval during which the job was eligible. This means that we know the parameters needed to expand vertex v.

During the last iteration of the algorithm, i.e., $t = l_{ext} + 1$, the certainly-eligible job is a made-up job that has higher policy priority than any other applicable job in $v.E^A$. This way, the jobs left in variables CE or PE are removed, which means that they expand the vertex v.

3.4.6 Merge phase

Unlike the expansion phase, the implementation of the merge phase is the same for both work-conserving and non-work-conserving policies. A pseudocode of the merge phase can be seen in Algorithm 9.

Algorithm 9 Merge phase

Input: a set of vertices V_i^{ex}
Output: none, the changes are done locally

```
1: function MERGE_PHASE(V_i^ex)
2:     while ∃v_m, v_x ∈ V_i^ex st. E^{v_m} = E^{v_x} ∧ [v_m.e, v_m.l] ∩ [v_x.e, v_x.l] ≠ ∅ do
3:         v_m.e ← min(v_m.e, v_x.e)
4:         v_m.l ← max(v_m.l, v_x.l)
5:         for each edge σ_x ∈ v_x.in do
6:             if ∄σ_m ∈ v_m.in st. σ_x.E = σ_m.E ∧ σ_x.s = σ_m.s then
7:                 σ_x.d = v_m
8:                 v_m.in = v_m.in ∪ {σ_x}
9:         remove v_x and all edges {σ_x | σ_x.d = v_x} from the graph
```

The merge phase keeps merging vertices until there are no two vertices that satisfy the conditions to be merged. When merging two vertices v_m and v_x, vertex v_m has its finish time range changed to $[v_m.e, v_m.l] \cup [v_x.e, v_x.l]$ and all edges directed to v_x are redirected to v_m. Because the schedule graph is not a multigraph, an edge can be deleted instead if there already exists an edge directed to v_m with the same label.

3.4.7 Complete schedule graph generation

Schedule graph generation alternates between the described expansion and merge phase. It starts with root vertex v_r where $v_r.e = v_r.l = 0$ and $v_r.in = v_r.out = \emptyset$. Pseudocode of the graph generation algorithm can be seen in Algorithm 10. Although the input of the algorithm $\mathcal{E}$ is not used directly, every vertex computes the applicable jobs from the given set of tasks $\mathcal{E}$.

The graph is generated and changed only in the expansion and merge phases. The

Algorithm 10 Schedule graph generation

Input: a set of ET tasks $\mathcal{E}$, policy function $\mathcal{P}$
Output: if set of ET tasks $\mathcal{E}$ is schedulable under the given policy

```
1:  function GRAPH_GENERATION(E, P)
2:      v_r ← new vertex with v_r.e = v_r.l = 0 and v_r.in = v_r.out = ∅
3:      V_0 ← {v_r}
4:      for i = 1 to |E| do                    ▷ |E| is the total number of jobs in E
5:          V_i^ex ← EXPANSION_PHASE(V_{i-1})
6:          for each v_{i,j}^ex ∈ V_i^ex do
7:              E_{i,j}^ex ← label of the only edge in v_{i,j}^ex.in
8:              if v_{i,j}^ex.l > E_{i,j}^ex.d then
9:                  return false                ▷ Deadline miss detected
10:         V_i ← MERGE_PHASE(V_i^ex)
11:     return true          ▷ Generation completed with no deadline misses
```

function GRAPH_GENERATION combines the two phases while detecting deadline misses.

The part of the code which detects deadlines (lines 6-9) can be executed after the merge phase on a potentially smaller amount of vertices. However, there is an issue with this approach. If the expansion phase yields a vertex with a deadline miss, then the deadline would be found after the merge phase was conducted. In this case, the execution of the merge phase would be redundant.

The merge phase does not need to be done in the very last iteration of the algorithm. Instead, the deadline miss detection would be executed directly on the expanded vertices V_i^{ex} and the merge phase would be omitted as its result is not used.

3.5 A formal description of schedule graph generation for non-work-conserving policies

In this section, we expand on the ideas of schedule graph generation for work-conserving policies and describe schedule graph generation for non-work-conserving policies.

3.5.1 Generalization of non-work-conserving policies

We have intentionally made all described non-work-conserving policies follow the EDF-FP policy when they are not trying to insert idle times. This makes it easier to create a single algorithm that works for all described policies.

A commonality of the described non-work-conserving policies is that they all define some critical time t^c using a job or a set of jobs. A job cannot be picked by any of the policies if it may finish after t^c. This rule does not apply to one job, which we call critical job E^c.

Note that both t^c and E^c can be obtained from a set of applicable jobs, which means that each vertex has its own t^c and E^c. This will be denoted as $v.t^c$ and $v.E^c$ for vertex

Notation	Name	Type
$v.e$	earliest finish time	integer
$v.l$	latest finish time	integer
$v.in$	incoming edges	set of edges
$v.out$	outgoing edges	set of edges
$v.E^A$	applicable jobs	set of jobs
$v.t^c$	critical time	integer
$v.E^c$	critical job	job

Table 3.2: All variables that are associated with a single vertex. Variables $v.E^A$, $v.t^c$ and $v.E^c$ are used to simplify the notation and terminology. They do not have to be stored and can be computed when needed using the other variables.

v. We won't be adding any more variables to a vertex and we, therefore, provide a summary of vertex notation in Table 3.2.

How to obtain the t^c and E^c has been described in pseudocodes of each policy. However, for clarity, we define a function for each of the non-work-conserving policies which returns t^c and E^c for a non-empty set of applicable jobs E^A. These functions can be seen in Algorithm 11. The t^c and E^c are the same for all work-conserving policies.

This model is much simpler compared to [12] as deadline prevention of the whole expansion phase of a single vertex is described only using two variables while [12] uses a function with three arguments.

3.5.2 Job eligibility for non-work-conserving policies

To accommodate for the properties of non-work-conserving policies, we need to redefine certainly and possibly eligible jobs. In general, these properties can be defined on a set of applicable jobs E^A, critical time t^c, critical job E^c, and time t. We defined them based on a vertex v and time t to ease off the notation.

For a given vertex v and time t, a job $E_{i,j} \in v.E^A$ *violates time* $v.t^c$ if $E_{i,j}.c^{max} + t > v.t^c \wedge E_{i,j} \neq v.E^c$. That is, $E_{i,j}$ is not the critical job and if it was executed at time t it might finish after $v.t^c$.

For a given vertex v and time t, a job E^t_{ce} is *certainly-eligible* at time t if E^t_{ce} is certainly released at time t, does not violate $v.t^c$ and there is no other certainly released applicable job at time t which does not violate $v.t^c$ and has a higher policy priority than E^t_{ce}. Formally, given vertex v, job $E^t_{ce} \in v.E^A$ is certainly-eligible at time t iff

$$E^t_{ce}.r^{max} \leq t \ \wedge \ (t + E^t_{ce}.c^{max} \leq v.t^c \ \vee \ E^t_{ce} = v.E^c) \ \wedge \ \nexists E_{i,j} \in v.E^A \ st. \ E_{i,j} \neq E^t_{ce} \ \wedge$$
$$E_{i,j}.r^{max} \leq t \ \wedge \ (t + E_{i,j}.c^{max} \leq v.t^c \vee E_{i,j} = v.E^c) \ \wedge \ E_{i,j} = \mathcal{P}(\infty, \{E_{i,j}, E^t_{ce}\})$$

In other words, from a set of certainly released jobs that do not violate $v.t^c$ the certainly-eligible job is the one with the highest policy priority out of all of them. Note that there is at most one certainly-eligible job at time t.

Similarly, a job E^t_{pe} is *possibly-eligible* at time t if it is possibly released at time t,

Algorithm 11 How to get t^c and E^c for each policy

Input: Applicable jobs E^A (non-empty)
Output: t^c and E^c of a given policy

1: **function** EDF-FP_C(E^A)
2: **return** (∞, *null*)
3: **function** P-RM_C(E^A)
4: $E^c \leftarrow$ job from E^A with $p = 0$ and the lowest r^{max}
5: **if** E^c is *null* **then**
6: **return** (∞, *null*)
7: **else**
8: $t^c \leftarrow E^c.d - E^c.c^{max}$
9: **return** (t^c, E^c)
10: **function** CP_C(E^A)
11: $E^c \leftarrow$ job from E^A with the lowest d value
12: $t^c \leftarrow E^c.d - E^c.c^{max}$
13: **return** (t^c, E^c)
14: **function** CW_C(E^A)
15: $E^S \leftarrow E^A$ sorted by d in descending order
16: $t^c \leftarrow \infty$
17: **for each** $E^s_{i,j} \in E^S$ **do**
18: **if** $E^s_{i,j}.d < t^c$ **then**
19: $t^c \leftarrow E^s_{i,j}.d - E^s_{i,j}.c^{max}$
20: **if** $E^s_{i,j}.d \geq t^c$ **then**
21: $t^c \leftarrow t^c - E^s_{i,j}.c^{max}$
22: $E^c \leftarrow$ the last job in E^S
23: **return** (t^c, E^c)

does not violate $v.t^c$ and has higher policy priority than E^t_{ce}. Formally, we define the set of possibly-eligible jobs at time t as

$$\{E_{i,j} \mid E_{i,j} \in v.E^A \ \wedge \ E_{i,j}.r^{min} \leq t < E_{i,j}.r^{max} \ \wedge \\ (t + E_{i,j}.c^{max} \leq v.t^c \ \vee \ E_{i,j} = v.E^c) \ \wedge \ E_{i,j} = \mathcal{P}(\infty, \{E_{i,j}, E^t_{ce}\})\}$$

Just as with work-conserving policies, jobs that are certainly-eligible or possibly-eligible at time $t \in [v.e, v.l]$ may begin execution in some execution scenario at time t. This means that the previous pseudocodes in Algorithm 7, Algorithm 9, and Algorithm 10 apply for non-work-conserving policies as long as we use the modified definition of certainly and possibly eligible jobs.

3.6 Possible improvements

In Algorithm 8 we provided a low-level description of the expansion phase for work-conserving policies. One possible way of improving this algorithm is to not iterate over all times $t \in [v_n.e, l_{ext} + 1]$. Instead, we can iterate only over times $t \in \{t \mid E_{i,j} \in E^A \wedge (E_{i,j}.r^{min} = t \vee E_{i,j}.r^{max} = t)\} \cup \{l_{ext} + 1\}$ and the result of the expansion phase will be the same. However, we need to make sure that we iterate over those times in ascending order.

A similar rule applies for the expansion phase of non-work-conserving policies. Here we need to add times where a job $E_{i,j}$ may no longer be eligible because it would violate critical time. Formally, we need to iterate only over times $t \in (\{t \mid E_{i,j} \in E^A \wedge (E_{i,j}.r^{min} = t \vee E_{i,j}.r^{max} = t)\} \cup \{l_{ext} + 1\} \cup \{t \mid E_{i,j} \in E^A \wedge E_{i,j} \neq v.E^c \wedge t = v.t^c - E_{i,j}.c^{max} + 1\})$. Once again, these times need to be iterated over in ascending order.

Chapter 4

ET solution evaluation

In this section, we discuss the evaluation of the described algorithms. We won't try to give asymptotic complexity as it is very difficult to describe due to the branching factor being hard to identify. Instead, we are doing empiric measurements on randomly generated instances.

4.1 System information

The ET schedulability brute force test and the ET schedulability test using schedule graph were implemented using Java 8. The source code of this implementation and instances used in benchmarking are publicly available on GitHub[1]. The code implements the improvements mentioned in Chapter 3.6. These are the relevant specifications of the computer the benchmarking was done on:

- CPU: 3550Mhz, 8 cores, 16 threads
- Memory: 16GB, DDR4, 2666MHz (Java VM used at most 4GB of memory)

4.2 Instance generation

We developed a function which returns a random instance based on several parameters. The aim of the function is to have utilization as close as possible to

$$U = \sum_{i=1}^{n} \frac{\mathcal{E}_i.c^{max}}{\mathcal{E}_i.\tau}$$

for a set of ET tasks $\mathcal{E}$. Utilization could be described as a percentage of time a processor will spent executing jobs if $E_{i,j}.c = E_{i,j}.c^{max}$ for all $E_{i,j} \in \mathcal{E}$. This function accepts the following parameters:

- **Number of tasks** $A^{\mathcal{E}}$ - Total number of tasks of the instance.

[1] `https://github.com/redakez/ettt-scheduling`

- **Target hyperperiod** A^{η} - Maximum possible hyperperiod of the instance. For large amount of tasks, the hyperperiod of the instance usually equals to the target hyperperiod.
- **Minimum task period** A^{τ} - Minimum period τ a task may have
- **Utilization** A^{U} - Utilization of the instance
- **Utilization swaps** A^{U}_{s} - For how long should the utilization be randomized
- **Utilization swap amount** A^{U}_{a} - How much should utilization differ between tasks
- **Release jitter percentage** A^{P}_{j} - The amount of release jitter
- **Execution time variation percentage** A^{P}_{c} - The amount of execution time variation
- **Release shift percentage** A^{P}_{r} - How far should the release time be from the beginning of a period
- **Deadline shift percentage** A^{P}_{d} - How far should the deadline be from the end of a period
- **Random shift percentage** A^{P} - How much should the previous 4 percentages randomly differ between tasks
- **Minimum priority** A^{p}_{min} - Minimum priority a task can have
- **Maximum priority** A^{p}_{max} - Maximum priority a task can have
- **Seed** A^{S} - Seed of the instance

To achieve utilization as close as possible to the provided argument, each task $\mathcal{E}_i$ is assigned a number $U_i = A^U/A^{\mathcal{E}}$. Then the algorithm picks two random tasks $\mathcal{E}_i$ and $\mathcal{E}_j$ and sets $U_i = U_i + U_j \cdot A^U_a$ and $U_j = U_j - U_j \cdot A^U_a$. This is done A^U_s times.

Execution time variation percentage is defined as $(\mathcal{E}_i.c^{max} - \mathcal{E}_i.c^{min})/(\mathcal{E}_i.c^{max} - 1)$ for task $\mathcal{E}_i$. Similarly, release jitter percentage is defined as $(\mathcal{E}_i.r^{max} - \mathcal{E}_i.r^{min})/\mathcal{E}_i.r^{max}$. The release jitter and execution time variation percentage can be understood as: the higher the percentage the more release jitter or execution time variation. If both percentages are set to 0, then $\mathcal{E}_i.r^{min} = \mathcal{E}_i.r^{max}$ and $\mathcal{E}_i.c^{min} = \mathcal{E}_i.c^{max}$ for all $\mathcal{E}_i \in \mathcal{E}$. The instance generation algorithm tries to make all tasks have release jitter percentage as close as possible to A^P_j and execution time variation percentage as close as possible to A^P_c.

Similarly, the instance generation algorithm tries to make deadline of task $\mathcal{E}_i$ as close as possible to $\mathcal{E}_i.d = (\mathcal{E}_i.\tau + \mathcal{E}_i.c^{max})/2 + (1 - A^P_d) \cdot (\mathcal{E}_i.\tau - \mathcal{E}_i.c^{max})/2$ and the maximum release time of a task as close as possible to $\mathcal{E}_i.r^{max} = A^P_r \cdot (\mathcal{E}_i.\tau - \mathcal{E}_i.c^{max})/2$. These two percentages limit in what interval a job can be executed. Note that $A^P_d = 0$ results in $\mathcal{E}_i.d = \mathcal{E}_i.\tau$ and $A^P_r = 0$ results in $\mathcal{E}_i.r^{max} = 0$. On the other hand, if both values are to 1 then $\mathcal{E}_i.r^{max} + \mathcal{E}_i.c^{max} = \mathcal{E}_i.d$.

The random shift percentage A^P gives a bound on how much can the previous four percentage variables differ. For instance, if $A^P = 0.5$ and $A^P_d = 0.2$ then each task will have its deadline defined as if the value A^P_d was taken uniformly from range $[A^P \cdot A^P_d, A^P_d + (1 - A^P_d) \cdot A^P] = [0.5 \cdot 0.2, 0.2 + 0.8 \cdot 0.5] = [0.1, 0.6]$. The same formula would

apply for any other percentage variable. All of the percentage arguments can be set to any double in range $[0, 1]$.

We will use this function to generate instances for future run time measurement tests.

4.3 Empirical correctness verification

Using the instance generator, we generated 10 million instances with arguments $A^{\mathcal{E}} = 5$, $A^{\eta} = 10$, $A^{\tau} = 5$, $A^{U} = 0.3$, $A^{U}_{s} = 20$, $A^{U}_{a} = 0.1$, $A^{P}_{j} = 0.3$, $A^{P}_{c} = 0.3$, $A^{P}_{r} = 0.1$, $A^{P}_{d} = 0.1$, $A^{P} = 0.5$, $A^{p}_{min} = 1$ and $A^{p}_{max} = 2$. After each instance was generated, it was tested for schedulability using both the brute force ET schedulability test and schedule graph ET schedulability test. The results did not differ on any of the instances for any of the described policies.

4.4 Comparison with SANS schedulability test

The aim of this subsection is to compare the run times of our implementation to implementation based on [12] which is currently maintained by Björn B. Brandenburg on GitHub[2]. We will call this implementation the SANS schedulability test. This application implements the schedule graph generation algorithm based on [12] as well as [15] and [14]. However, we will be using only part of the application which is described in [12], i.e., uniprocessor analysis. The application is written in C++.

4.4.1 Generated datasets

Because our approach and the approach of [12] differs in non-work-conserving policies (see Appendix A), we will be comparing the 2 schedulability tests only for the EDF policy.

We generated 3 datasets $\mathcal{D}^{s}_{1}$, $\mathcal{D}^{s}_{2}$, and $\mathcal{D}^{s}_{3}$. The first dataset $\mathcal{D}^{s}_{1}$ was generated with $A^{P}_{j} = A^{P}_{c} = 0$, the second dataset $\mathcal{D}^{s}_{2}$ with $A^{P}_{j} = A^{P}_{c} = 0.3$, and the third dataset $\mathcal{D}^{s}_{3}$ with $A^{P}_{j} = A^{P}_{c} = 0.6$. Each dataset contains 960 instances.

Because EDF policy has a very low schedulability, all generated instances had utilization of 0.3 which should result in both several schedulable and non-schedulable instances. All instances were generated with the following arguments: $2 \leq A^{\mathcal{E}} \leq 61$, $A^{\eta} = 1000000000$, $A^{\tau} = 10000000$, $A^{U} = 0.3$, $A^{U}_{s} = 300$, $A^{U}_{a} = 0.1$, $A^{P}_{r} = 0.05$, $A^{P}_{d} = 0.05$, $A^{P} = 0$, $A^{p}_{min} = A^{p}_{max} = 1$. The high periods and non-zero deadline and release shift is set due to the fact that both ET schedulability tests handle tie-breakers differently and these values make it extremely unlikely for them to happen. Each instance was generated with a different seed.

[2] `https://github.com/gnelissen/np-schedulability-analysis`

4.4.2 How was the benchmarking conducted

The benchmarking was done using a Java program, which executed a dataset of instances. Both schedulability tests were launched only once for each instance and ran on a single processor. However, Java's garbage collector did run in parallel.

All tested instances were generated beforehand. The input file of our schedulability test is a set of ET tasks while the input file for the SANS schedulability test contains a set of ET jobs. Because the input format differs, two input files were created for each instance, one for our test and one for the SANS test. No schedulability tests ran in parallel, i.e., there was always only one schedulability test running at any time. The measured time includes parsing of the input file.

4.4.3 Results

The results for dataset $\mathcal{D}_1^s$ can be see in Figure 4.1, $\mathcal{D}_2^s$ in Figure 4.2, and $\mathcal{D}_3^s$ in Figure 4.3. Generally speaking, the higher the release jitter and execution time variation percentage the higher the run time. Note the increasing scale on the y axis with each subsequent dataset. During the benchmarking, none of the instances yielded a conflicting result between the two schedulability tests. As can be seen in the charts, the schedulability tests seem to have the same asymptotic behavior. The instances that yielded non-schedulable finish quicker, as they do not compute the entire schedule graph unlike in the schedulable instances.

Additionally, we provide a box plot of speedups between our schedulability test and SANS schedulability test for each of the three datasets in Figure 4.4. Speedup in this case is defined as

$$\mathcal{S} = \frac{\mathcal{S}_s}{\mathcal{S}_o}$$

where $\mathcal{S}_o$ is the run time of our test and $\mathcal{S}_s$ is the run time of the SANS schedulability test. As can be seen in the figure, for the datasets $\mathcal{D}_1^s$ and $\mathcal{D}_2^s$, the median speedup is about 15, and for dataset $\mathcal{D}_3^s$ the median speedup is about 10. This might be due to the fact that the definition of job eligibility in [12] differs from our approach, although the result is the same.

4.5 Policy schedulability and performance

In this section, we evaluate the run times and schedulability of all described policies.

4.5.1 Generated datasets

Once again, we generated 3 datasets $\mathcal{D}_1^p$, $\mathcal{D}_2^p$, and $\mathcal{D}_3^p$. The first dataset $\mathcal{D}_1^p$ was generated with $A_j^P = A_c^P = 0$, the second dataset $\mathcal{D}_2^p$ with $A_j^P = A_c^P = 0.3$, and the third dataset $\mathcal{D}_3^p$ with $A_j^P = A_c^P = 0.6$. Each dataset contains 200 instances for each utilization $A^U \in \{0.1, 0.2, \ldots, 0.9\}$, i.e., 1800 instances for each dataset. All instances were generated with arguments: $2 \leq A^{\mathcal{E}} \leq 61$, $A^{\eta} = 1000000000$, $A^{\tau} = 10000000$, $A_s^U = 300$, $A_a^U = 0.1$, $A_r^P = 0.05$, $A_d^P = 0.05$, $A^P = 0$.These arguments are the same as for datasets $\mathcal{D}_1^s$, $\mathcal{D}_2^s$, and

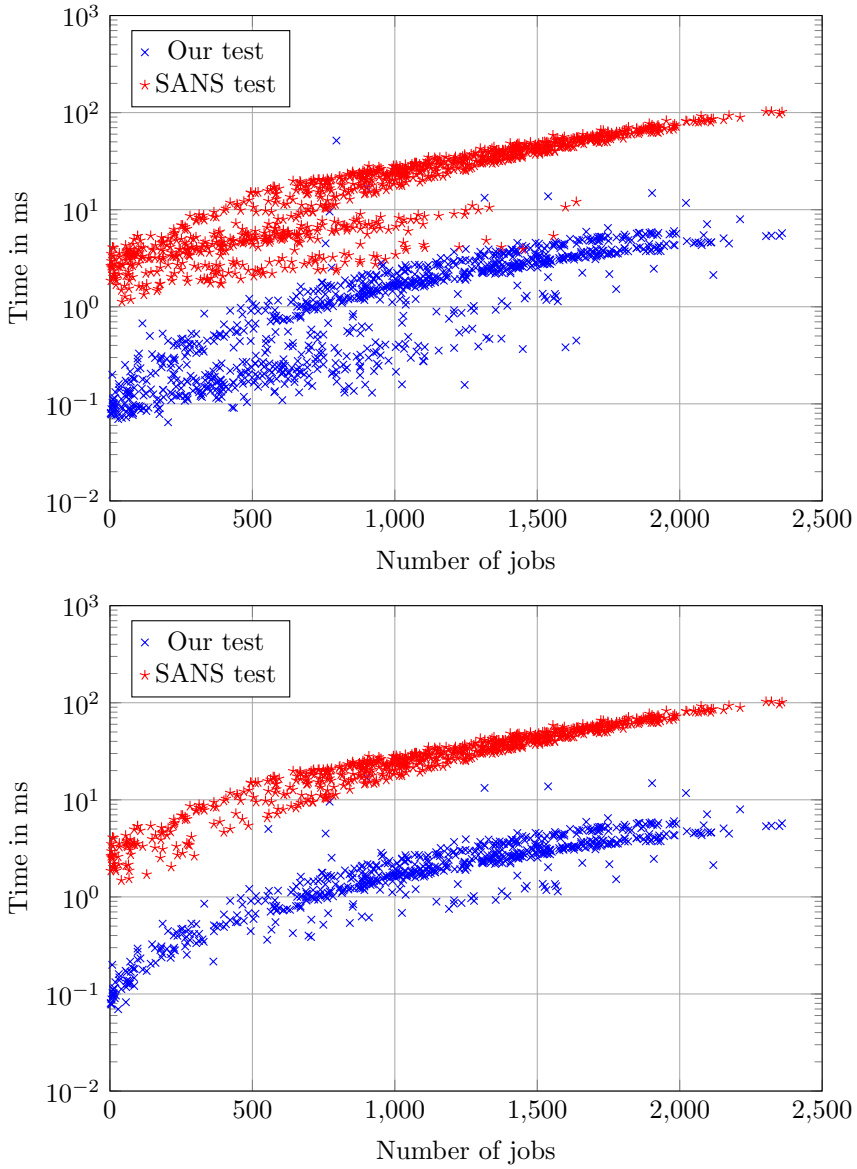


Figure 4.1: Run times of our schedulability test and SANS schedulability test on dataset $\mathcal{D}_1^s$. The top chart shows measurements on all instances and the bottom chart only measurements that yielded schedulable.

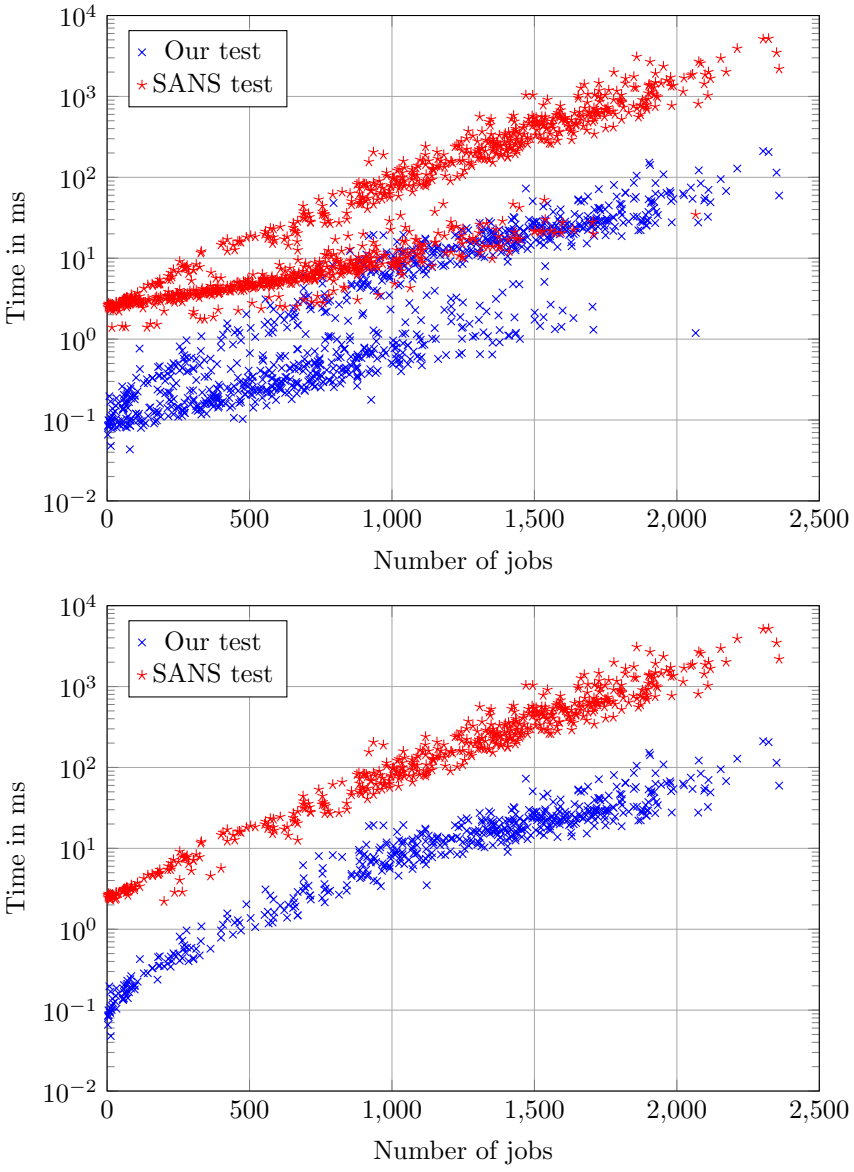


Figure 4.2: Run times of our schedulability test and SANS schedulability test on dataset $\mathcal{D}_2^s$. The top chart shows measurements on all instances and the bottom chart only measurements that yielded schedulable.

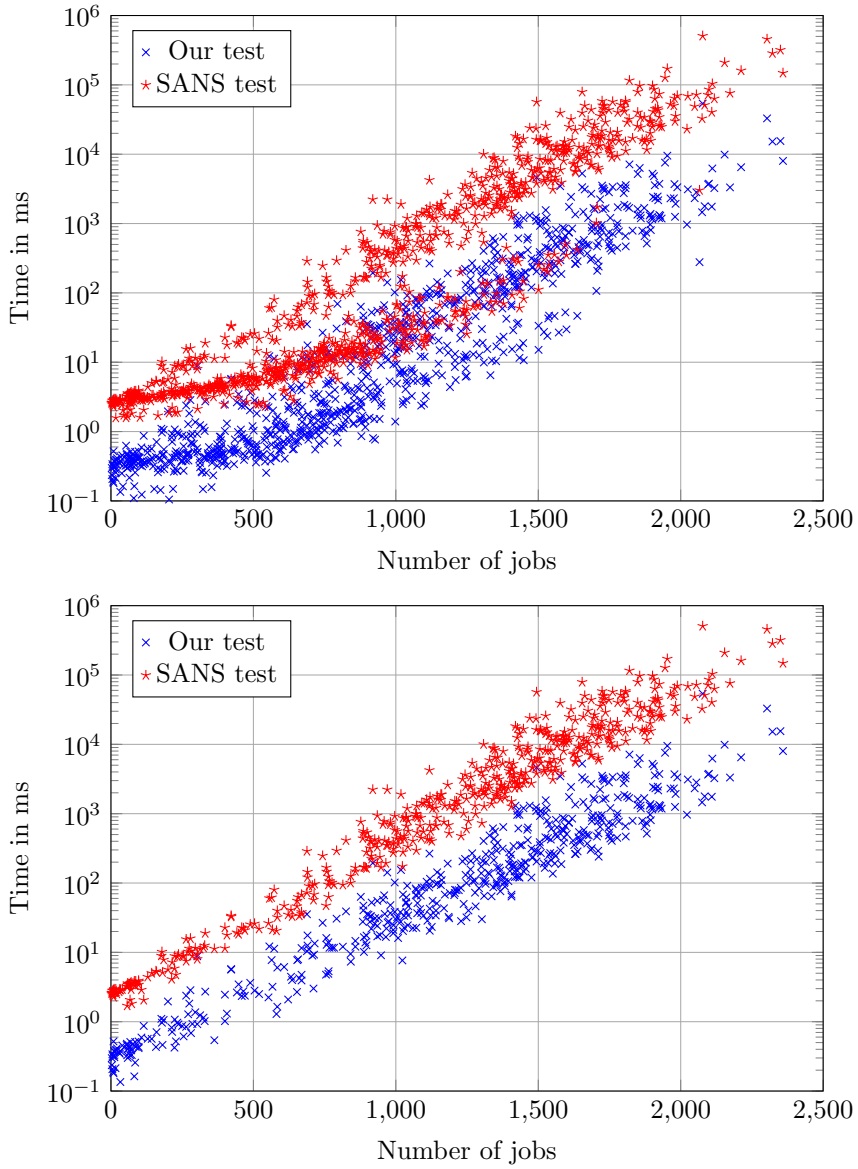


Figure 4.3: Run times of our schedulability test and SANS schedulability test on dataset $\mathcal{D}_3^s$. The top chart shows measurements on all instances and the bottom chart only measurements that yielded schedulable.

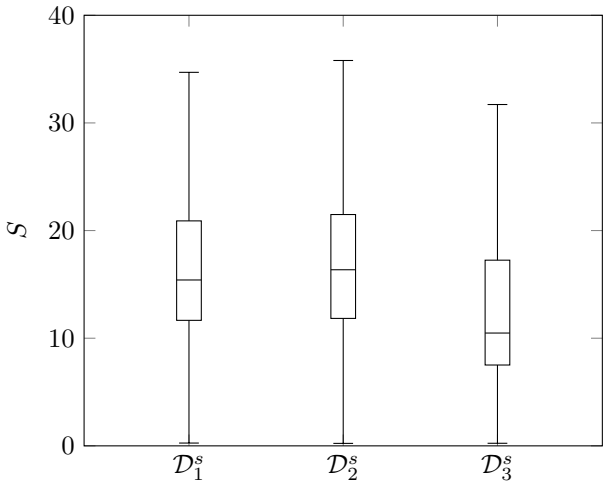


Figure 4.4: Box plots of speedups for the datasets $\mathcal{D}_1^s$, $\mathcal{D}_2^s$, and $\mathcal{D}_3^s$ between our schedulability test and SANS schedulability test.

$\mathcal{D}_3^s$, only now the utilization is variable instead of fixed. Additionally, all task priorities are set to zero so that the P-RM policy has the maximum possible schedulability.

4.5.2 Results

The resulting run times on dataset $\mathcal{D}_1^p$ can be see in Figure 4.5, $\mathcal{D}_2^p$ in Figure 4.6, and $\mathcal{D}_3^p$ in Figure 4.7. For dataset $\mathcal{D}_3^p$, two generated instances had to be left out of the measurements as they would not finish its run time due to lack of memory. These two instances had unusually high branching factor and tens of millions of vertices on a single level of the schedule graph.

To get a better understanding of the relative run times between different policies, we provide Figure 4.8 which shows how much slower the non-work-conserving policies were compared to the EDF policy. More formally, we show the inverse of speedup which is in this case defined as

$$\mathcal{S}^{-1} = \frac{\mathcal{S}_n}{\mathcal{S}_e}$$

where $\mathcal{S}_n$ is the run time of some non-work-conserving policy and $\mathcal{S}_e$ is the run time of the EDF policy. As can be seen in the figure, the P-RM and CP policies are about 2 times slower than the EDF policy on all datasets. The median $\mathcal{S}^{-1}$ for the CW policy is between 3 and 5 depending on the dataset.

Schedulability of policies for all datasets can be seen in Figure 4.9. The schedulability of all policies decreases with increasing release jitter and execution time variation percentage except for the CW policy.

The EDF policy runs the fastest as it uses simplified rules of schedule graph generation. Run times of the CP and P-RM policy are about the same. This is likely due to the fact that both policies search through the entire set of applicable jobs to find the critical job and the critical time. The CW policy is the slowest as it has to sort the set of applicable jobs and then process each job in order to find the critical job and the critical time.

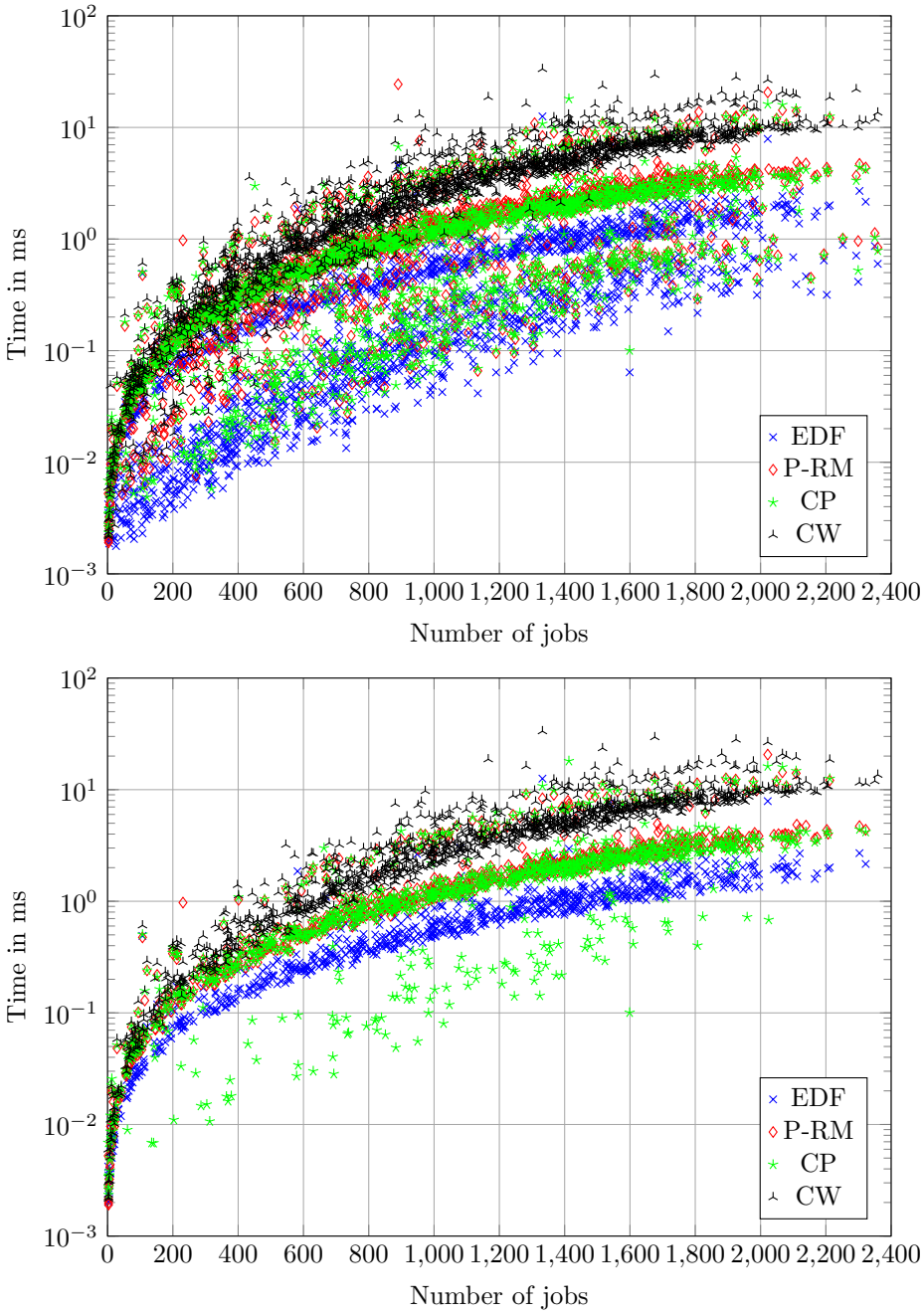


Figure 4.5: Run times of different scheduling policies on dataset $\mathcal{D}_1^p$. The top chart shows measurements on all instances and the bottom chart only measurements that yielded schedulable.

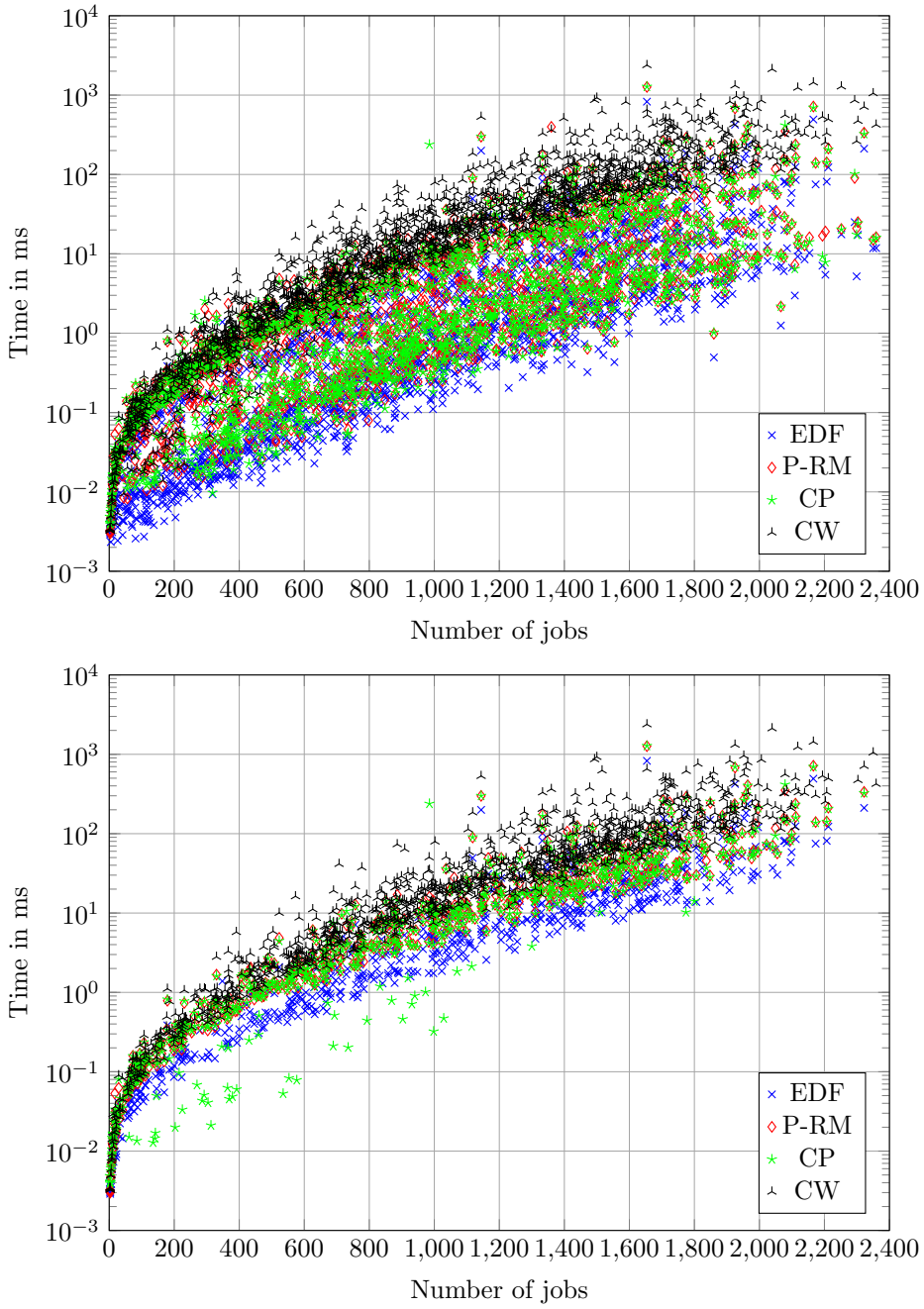


Figure 4.6: Run times of different scheduling policies on dataset $\mathcal{D}_2^p$. The top chart shows measurements on all instances and the bottom chart only measurements that yielded schedulable.

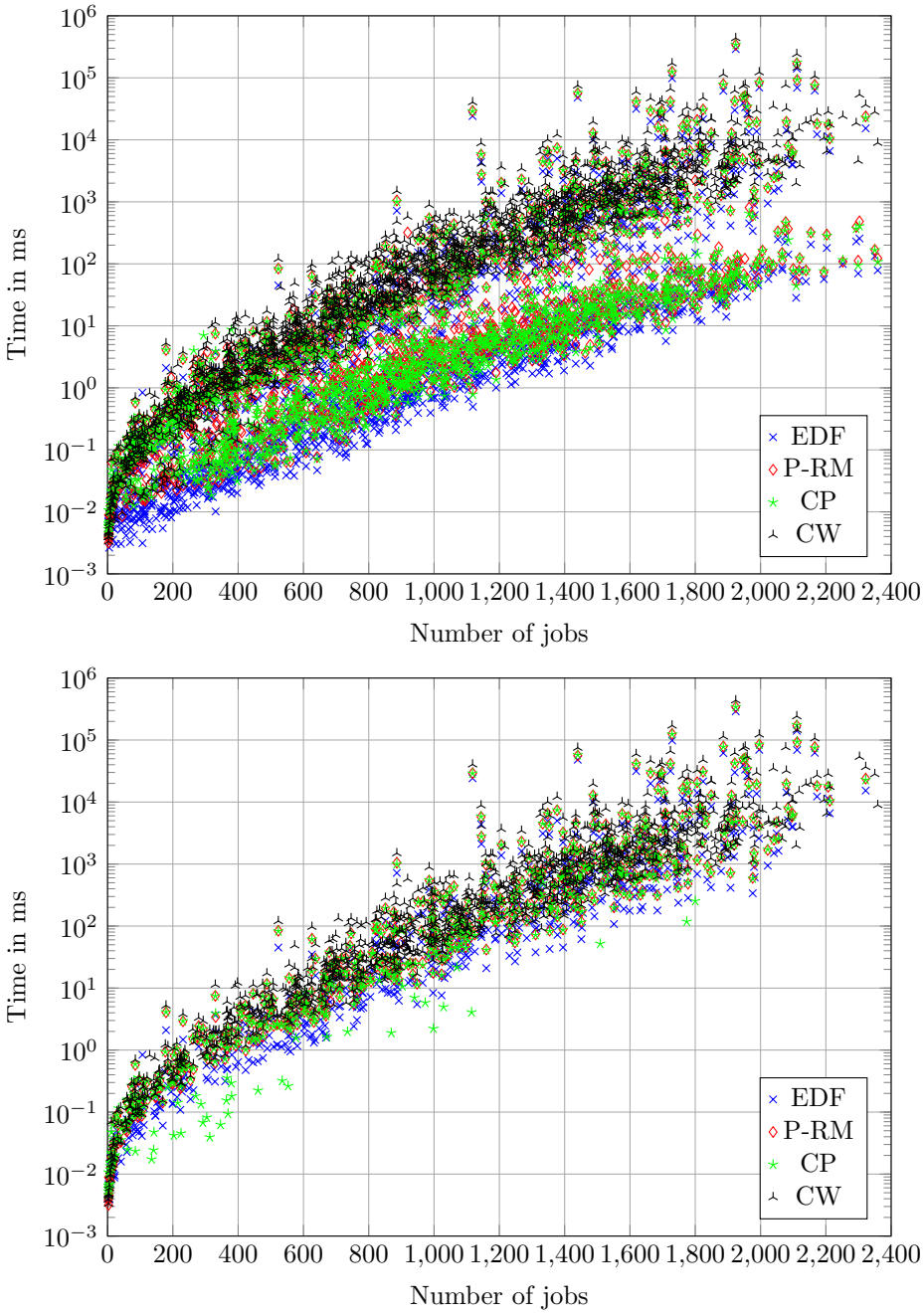


Figure 4.7: Run times of different scheduling policies on dataset $\mathcal{D}_3^p$. The top chart shows measurements on all instances and the bottom chart only measurements that yielded schedulable.

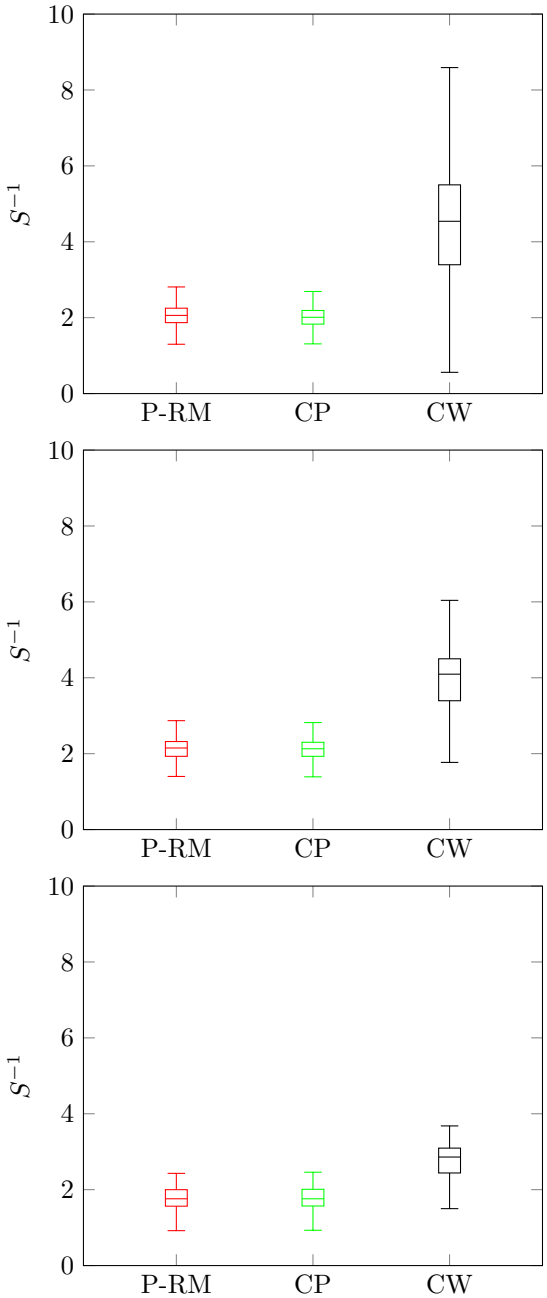


Figure 4.8: The inverse of speedup for each policy. The top chart shows inverse of speedup on the dataset $\mathcal{D}_1^p$, middle chart on dataset $\mathcal{D}_2^p$, and the bottom one on dataset $\mathcal{D}_3^p$.

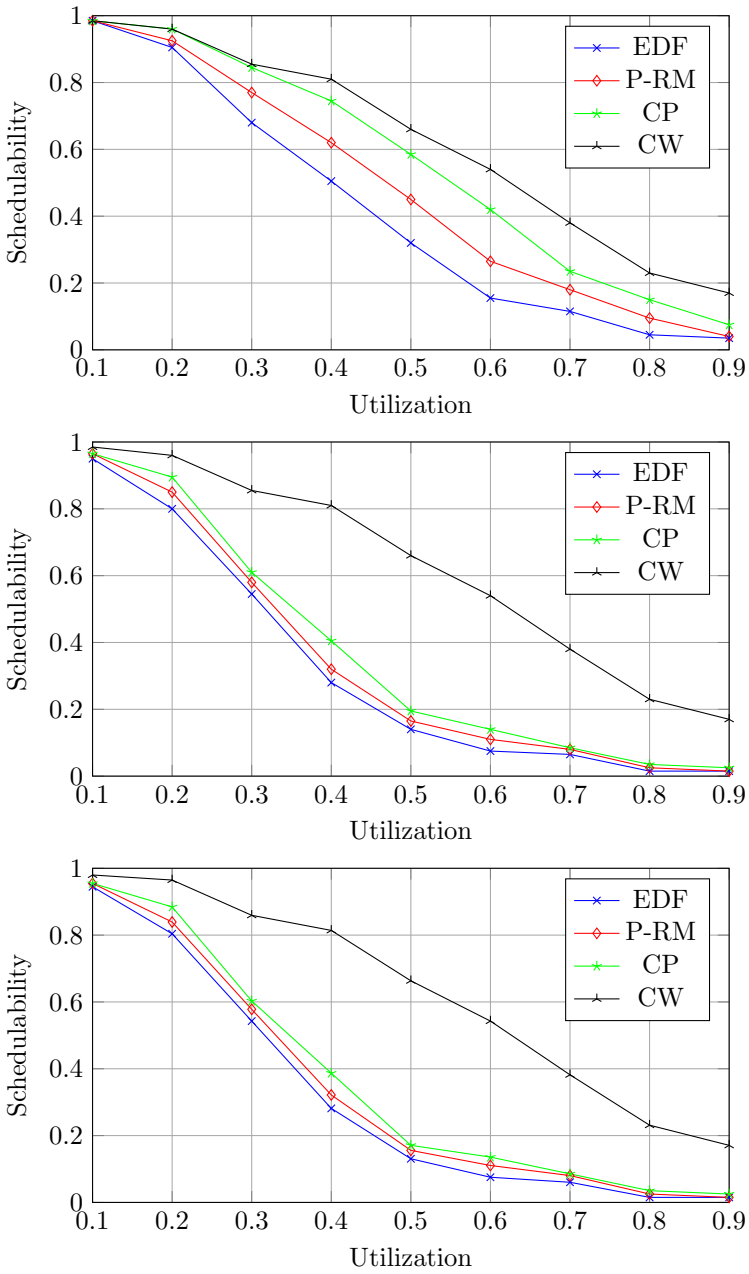


Figure 4.9: Schedulability of policies. The top chart shows schedulability on the dataset $\mathcal{D}_1^p$, middle chart on dataset $\mathcal{D}_2^p$, and the bottom one on dataset $\mathcal{D}_3^p$.

Chapter 5

ET+TT solutions

This section describes algorithms that for a set of both ET and TT tasks find a set of valid start times for TT tasks.

5.1 Fixation of TT jobs

As mentioned in Chapter 2, we wish to find a set of valid start times for the TT tasks. First we need to describe a method, which for a set of ET tasks $\mathcal{E}$, TT tasks $\mathcal{T}$, and start times S evaluates if S is a valid set of start times. This can be done by transforming the TT tasks $\mathcal{T}$ into ET tasks and running the ET schedulability test.

TT task $\mathcal{T}_i$ becomes *fixed* at a start time $S_i \in [\mathcal{T}_i.r, \mathcal{T}_i.d - \mathcal{T}_i.c]$ when it is transformed to an ET task $\mathcal{E}_i^f$ where $\mathcal{E}_i^f.r^{min} = \mathcal{E}_i^f.r^{max} = S_i$, $\mathcal{E}_i^f.c^{min} = \mathcal{E}_i^f.c^{max} = \mathcal{T}_i.c$, $\mathcal{E}_i^f.d = S_i + \mathcal{T}_i.c$ and $\mathcal{E}_i^f.p = 0$.

A set of TT tasks $\mathcal{T} = (\mathcal{T}_1, \ldots, \mathcal{T}_m)$ can be fixed by a set of start times $S = (S_1, \ldots, S_m)$ by fixing each TT task $\mathcal{T}_i$ using start time S_i. This then results in a set of fixed TT tasks $\mathcal{E}^f$. When we wish to check if a set of start times S is valid for a set of TT tasks $\mathcal{T}$ and ET tasks $\mathcal{E}$, we test schedulability on set $\mathcal{E} \cup \mathcal{E}^f$. If the test returns schedulable, then the set of start times S is valid.

To make sure that the fixed TT jobs are prioritized over the ET jobs, we add a constraint $\mathcal{E}_i.p > 0$ for every $\mathcal{E}_i \in \mathcal{E}$. If a fixed TT job $E_{i,j}^f$ if not executed at its predetermined start time, then the schedule results in a deadline miss due to the fact that $E_{i,j}^f.d = E_{i,j}^f.r + E_{i,j}^f.c$. This then follows the idea of time-triggered systems, where a job is always executed at a predetermined time.

5.2 Brute force algorithm

In this section, we describe a brute force algorithm which for a set of ET tasks $\mathcal{E}$ and TT tasks $\mathcal{T}$ finds the set of valid start times S. This algorithm goes through all possible combinations of start times, fixes the TT tasks for each combination, and runs an ET schedulability test. The algorithm terminates when it either finds a valid set of start times or goes through all possible combinations of start times and does not find a

solution.

A pseudocode of the brute force algorithm is in Algorithm 12. Note that $|\mathcal{T}|$ is the number of TT tasks.

Algorithm 12 ET+TT Brute force algorithm

Input: Set of TT tasks $\mathcal{T}$ and ET tasks $\mathcal{E}$
Output: Set of valid start times S or *null*

```
1:  return FIX_RECURSIVELY(1, array of |T| zeros)
2:
3:  function FIX_RECURSIVELY(i, S)
4:      if i > |T| then
5:          E^f ← fixed TT tasks T using start times S
6:          ET_success ← result of ET schedulability test for E ∪ E^f
7:          if ET_success then
8:              return S
9:      if i ≤ |T| then
10:         for s = T_i.r to T_i.d − T_i.c do
11:             S_i ← s
12:             success ← FIX_RECURSIVELY(i+1, S)
13:             if success ≠ null then
14:                 return S
15:     return null
```

5.2.1 Overlap check

What can improve the brute force algorithm is checking if fixing a TT task would create an overlap with another already fixed TT task. This can potentially prune many combinations while keeping the algorithm exact. The overlap checking can be done using an interval tree. The interval tree saves intervals and can check if a given interval overlaps with an interval in the interval tree. [10]

Pseudocode with a brute force algorithm with an overlap check can be seen in Algorithm 13. In the pseudocode, the interval tree is treated as a set of numbers. For some interval $[i, j]$ the operation of adding the interval to the interval tree is denoted as $IT \leftarrow IT \cup [i, j]$, operation of removing an interval as $IT \leftarrow IT \setminus [i, j]$, and operation of intersection as $IT \cap [i, j]$.

5.3 Fixation with and without jitter

In sections 5.1 and 5.2 the fixation of TT tasks was done without jitter, i.e., for each task, there was only one start time which determined the start time of each job repetition of that task. In Chapter 5.4 we will be fixing TT tasks with jitter, i.e., each job will have its own start time which determines its release time and deadline. A more formal description follows.

TT job $T_{i,j}$ becomes *fixed with jitter* at a start time $S_{i,j} \in [T_{i,j}.r, T_{i,j}.d - T_{i,j}.c]$ when

Algorithm 13 ET+TT Brute force algorithm with overlap check

Input: Set of TT tasks $\mathcal{T}$ and ET tasks $\mathcal{E}$
Output: Set of valid start times S or null

```
1: return FIX_RECURSIVELY(1, array of |𝒯| zeros, empty interval tree)
2:
3: function FIX_RECURSIVELY(i, S, IT)
4:     if i > |𝒯| then
5:         ℰ^f ← fixed TT tasks 𝒯 using start times S
6:         success ← result of ET schedulability test for ℰ ∪ ℰ^f
7:         if success then
8:             return S
9:     if i ≤ |𝒯| then
10:        for s = 𝒯_i.r : 𝒯_i.d − 𝒯_i.c do
11:            overlap ← false
12:            for j = 0 to η/𝒯_i.τ − 1 do
13:                if [s + j · 𝒯_i.τ, s + j · 𝒯_i.τ + 𝒯_i.c − 1] ∩ IT ≠ ∅ then
14:                    overlap ← true
15:                    break
16:            if overlap then
17:                continue
18:            for j = 0 to η/𝒯_i.τ − 1 do
19:                IT ← IT ∪ [s + j · 𝒯_i.τ, s + j · 𝒯_i.τ + 𝒯_i.c − 1]
20:            S_i ← s
21:            success ← FIX_RECURSIVELY(i+1, S, IT)
22:            if success ≠ null then
23:                return S
24:            for j = 0 to η/𝒯_i.τ − 1 do
25:                IT ← IT \ [s + j · 𝒯_i.τ, s + j · 𝒯_i.τ + 𝒯_i.c − 1]
26:    return null
```

it is transformed to an ET job $E^f_{i,j}$ where $E^f_{i,j}.r^{min} = E^f_{i,j}.r^{max} = S_{i,j}$, $E^f_{i,j}.c^{min} = E^f_{i,j}.c^{max} = T_{i,j}.c$, $E^f_{i,j}.d = S_{i,j} + T_{i,j}.c$ and $E^f_{i,j}.p = 0$. Note that a fixed ET job $E^f_{i,j}$ does not inherit its release time and deadline from ET task $\mathcal{E}^f_i$.

A set of TT tasks $\mathcal{T} = (\mathcal{T}_1, \ldots, \mathcal{T}_m)$ can be fixed with jitter by a set of start times $S = (S_{1,1}, \ldots, S_{1,h_1}, S_{2,1}, \ldots, S_{2,h_2}, \ldots\ldots, S_{m,1}, \ldots, S_{m,h_m})$, where $h_i = \eta/\mathcal{T}_i.\tau$, by fixing each TT job $T_{i,j}$ using start time $S_{i,j}$. This then results in a set of fixed TT tasks $\mathcal{E}^f$. When we wish to check if a set of start times S is valid for a set of TT tasks $\mathcal{T}$ and ET tasks $\mathcal{E}$, we test schedulability on set $\mathcal{E} \cup \mathcal{E}^f$. If the test returns schedulable, then the set of start times S is valid.

5.4 Heuristic algorithm for work-conserving policies

This section describes a scalable heuristic algorithm that for a set of ET tasks $\mathcal{E}$ and TT tasks $\mathcal{T}$ finds start times of TT jobs with jitter. This algorithm works only for work-conserving policies.

The algorithm uses the previously described schedule graph generation algorithm and adds a new phase called *fixation phase*. The resulting graph of the algorithm is called *fixation graph*.

First, we will focus on the basics of the fixation graph. Then we will provide a rough description of the generation algorithm and an example. The exact rules for fixation graph generation are described afterward.

5.4.1 Fixation graph structure

The schedule graph generation algorithm features a single type of vertex which is denoted as v. For clarity, we will call v a *regular vertex* from now on. The fixation graph contains another type of vertex called *decision vertex* denoted as w. It is called decision vertex because it decides start times of TT jobs. Unlike a regular vertex, the decision vertex does not contain a time interval $[e, l]$. Instead it contains time t, which is denoted as $w_i.t$ for vertex w_i. Just as in the schedule graph generation algorithm, the fixation graph generation algorithm starts with a root vertex v_r where $[e_r, l_r] = [0, 0]$ and $V_0 = \{v_r\}$.

Due to these modifications, a level V_i no longer contains vertices whose distance from the root vertex is i. The index i now only denotes how many times has the merge phase occurred.

5.4.2 Applicable TT jobs

So far we have considered applicable jobs to be a set of ET jobs. From now on we will call this set applicable ET jobs. Furthermore, we will now distinguish between applicable ET jobs E^A and applicable TT jobs T^A. Applicable TT jobs $T^A = (T_{1,j}, \ldots, T_{k,l})$ is a set of TT jobs that contains all jobs $T_{i,j}$ which satisfy: $T_{i,j}$ is unfinished $\wedge$ ($j = 1 \vee T_{i,j-1}$ is finished). Note that the definition differs from the definition of applicable ET jobs only by replacing ET jobs with TT jobs.

Furthermore, the set E^{v_i} contains only ET jobs encountered on a path from the root

vertex to vertex v_i. Set T^{v_i} contains only TT jobs encountered on a path from the root vertex to vertex v_i. A regular vertex v now contains applicable TT jobs, which is denoted as $v.T^A$. These jobs can still be computed using T^v.

5.4.3 Fixation phase

The schedule graph generation algorithm consists of two alternating phases, the expansion phase, and the merge phase. The fixation graph generation algorithm adds a new phase called *fixation phase*. This phase occurs before the expansion phase. The fixation phase tries different combinations of start times and in case a combination results in a deadline miss, it backtracks. It does this by adding decision vertices to the fixation graph in times, where TT jobs may be fixed. The fixation phase also incorporates a modified version of Bratley's algorithm, which is used to decide the start times of TT jobs.

5.4.4 Modified Bratley's algorithm

Bratley's algorithm is an algorithm that finds an optimal solution to scheduling problem $1|r_j, \tilde{d}_j|C_{max}$ [4][6], i.e., scheduling problem with non-periodic tasks, where each task has an execution time, release time, and deadline and the objective is to find a schedule that finishes the earliest possible. If any task finishes execution after its deadline, then the solution is not feasible. The Bratley algorithm finds an optimal solution by searching through space of all possible task order executions. To do this, the Bratley algorithm uses a graph we will call the *Bratley graph*.

A Bratley graph is contained in each decision vertex to determine the order of fixed TT jobs. The Bratley graph is made up of Bratley vertices. A Bratley vertex is denoted as $w_i.b_j$, where w_i is the decision vertex associated with the Bratley graph which contains vertex $w_i.b_j$. A Bratley vertex $w_i.b_j$ contains time $w_i.b_j.t$. All edges in the Bratley graph contain a single TT job, which is denoted as $\sigma_i.T$ for edge σ_i.

The Bratley graph used in the decision vertices is not generated using the original Bratley's algorithm, instead, a modified Bratley's algorithm is used. The modified version considers the fact that a TT job $T_{i,j}$ cannot be executed before $T_{i,j-1}$. It also generates all possible orderings by using a breadth-first-search instead of a depth-first-search. In order to prevent a combinatorial explosion of possible job orderings, the modified algorithm has an expansion and merge phase similar to the schedule graph generation algorithm.

Unlike the original algorithm, the modified Bratley algorithm does not generate the whole Bratley graph once called. Instead, it is called iteratively, generating the Bratley graph gradually with each call. Each Bratley vertex corresponds to fixing some TT jobs in some order. A Bratley vertex is *viable* if it satisfies certain conditions, for instance, not causing a deadline miss. These conditions are described in detail later.

5.4.5 Example

Let us consider 2 ET tasks $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2)$ and 2 TT tasks $\mathcal{T} = (\mathcal{T}_1, \mathcal{T}_2)$. The parameters of these tasks can be seen in Table 5.1. We will use the EDF scheduling policy which does

	r	c	d^{TT}	τ^{TT}
$\mathcal{T}_1$	4	2	9	10
$\mathcal{T}_2$	2	5	20	20

	r^{min}	r^{max}	c^{min}	c^{max}	d^{ET}	τ^{ET}
$\mathcal{E}_1$	1	1	1	3	8	10
$\mathcal{E}_2$	0	2	1	2	18	20

Table 5.1: Properties of TT and ET jobs used in the fixation graph example instance.

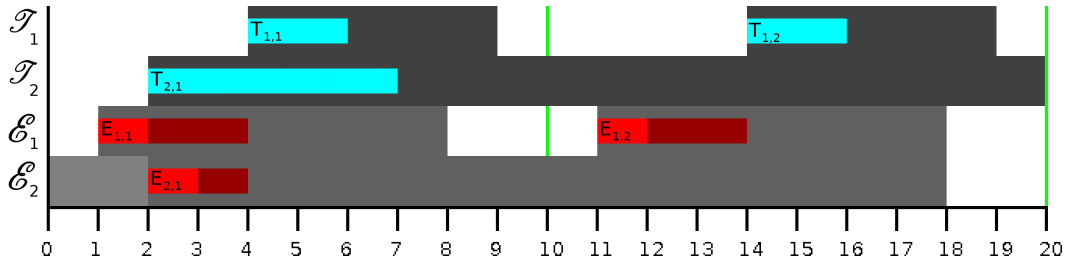


Figure 5.1: Loose Gantt chart of the fixation graph example instance.

not make use of priority values p and they are therefore undefined. The loose Gantt chart of this instance can be seen in Figure 5.1. The final fixation graph can be seen in Figure 5.2.

Level $V_0 = \{v_0\}$

The fixation graph generation starts with root vertex v_0. The fixation phase determines that an ET job may be executed before the first possible start time of a TT job. In this case, ET job $E_{2,1}$ may be executed at time $t = 0$, however the first possible start time is $S_{2,1} = 2$. The fixation phase makes no changes to the graph and the expansion phase proceeds next.

The expansion phase on $V_0 = \{v_0\}$ concludes that if ET job $E_{2,1}$ was released at time $t = 0$, then ET job $E_{2,1}$ is executed at time $t = 0$, otherwise ET job $E_{1,1}$ is executed at time $t = 1$. This results in vertices v_1 and v_2. The merge phase makes no changes to the graph.

Level $V_1 = \{v_1, v_2\}$

The fixation phase does not fix any TT jobs for vertices $V_1 = \{v_1, v_2\}$. Notice that if a TT job was fixed at time $t = 2$ and job $E_{1,1}$ began execution at time $t = 1$, then the schedule would result in a deadline miss for a scenario, where $E_{1,1}.c > 1$. The same logic applies for start time $t = 3$. Even for $t = 4$ the schedule would result in a deadline miss. This may happen for instance in a scenario where $E_{2,1}.r = 2$, $E_{1,1}.c = 2$ and $E_{2,1}.c = 2$. Same logic applies for $t = 5$.

The fixation phase does not make any changes to the graph for V_1. The expansion phase creates two new vertices, which are then merged in the subsequent merge phase into vertex v_3.

Level $V_2 = \{v_3\}$

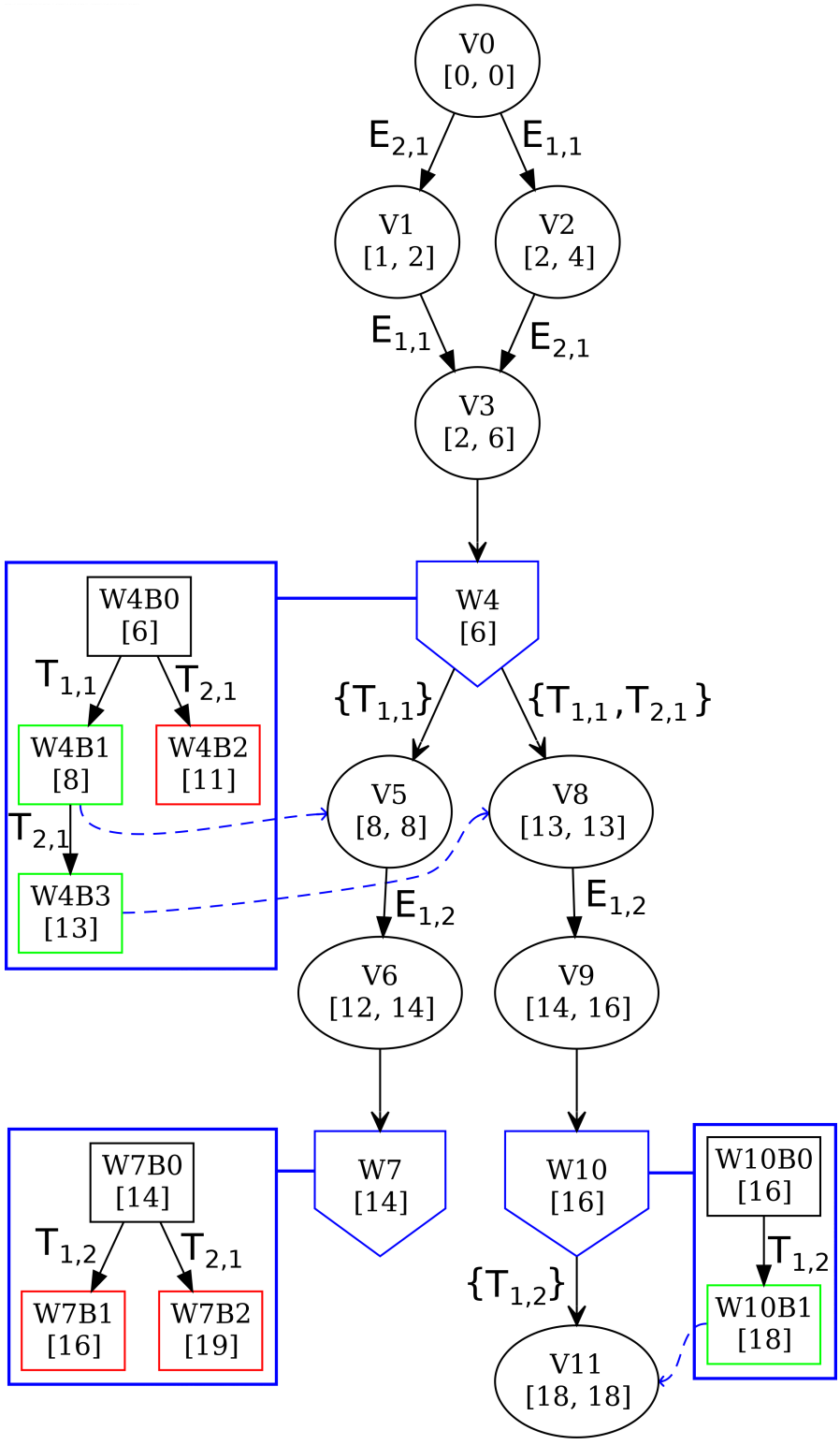


Figure 5.2: Fixation graph of the example instance. Viable Bratley vertices have a green outline while Bratley vertices that cause a deadline miss have a red outline. Note that a Bratley vertex $w_4.b_1$ does not actually contain a reference to vertex v_5. The dashed edge only shows that the vertex v_5 is created due to $w_4.b_1$ being viable. The same goes for Bratley vertices $w_4.b_3$ and $w_{10}.b_1$.

The fixation phase creates a decision vertex w_4 from $V_2 = \{v_3\}$. This is due to the fact that no unfinished ET jobs are released in the time interval $[v_3.e, v_3.l] = [2, 6]$ and a TT job may therefore be fixed at time $t = 6$. Which TT jobs, if any, will be fixed at this time is decided through the modified Bratley's algorithm.

The modified Bratley's algorithm starts by creating a root Bratley vertex $w_4.b_0$ with time $w_4.b_0.t = v_3.l = 6$. At this time one of TT jobs $T_{1,1}$ and $T_{2,1}$ may be fixed. The Bratley vertex $w_4.b_0$ corresponds to fixing no TT jobs at time $t = 6$. The earliest time an ET job releases is at time $t = 11$ and $T_{1,1}$ or $T_{2,1}$ would finish its execution before or at this time. This means that both can be fixed at time $t = 6$ and not interfere with other ET jobs. The modified Bratley's algorithm concludes that $w_4.b_0$ is not viable due to this fact and expands it, which results in vertices $w_4.b_1$ and $w_4.b_2$.

Vertex $w_4.b_1$ represents fixing TT job $T_{1,1}$ at time $w_4.b_0.t = 6$ and vertex $w_4.b_2$ represents fixing TT job $T_{2,1}$ at time $w_4.b_0.t = 6$. Therefore $w_4.b_1.t = w_4.b_0.t + T_{1,1}.c = 6 + 2 = 8$ and $w_4.b_2.t = w_4.b_0.t + T_{2,1}.c = 6 + 5 = 11$.

If TT job $T_{2,1}$ is fixed at time $S_{2,1} = 6$, then TT job $T_{1,1}$ cannot be fixed after $T_{2,1}$ at time $w_4.b_2.t = 11$ due to the latest start time of $T_{1,1}$ being $T_{1,1}.d - T_{1,1}.c = 9 - 2 = 7 < 11$. The modified Bratley's algorithm does not consider the vertex $w_4.b_2$ to be viable due to this fact. However, the vertex $w_4.b_1$ is considered viable and the fixation phase concludes by creating regular vertex v_5 with $[v_5.e, v_5.l] = [8, 8] = [w_4.b_1.t, w_4.b_1.t]$.

The fixation phase is done on level $V_2 = \{v_3\}$ and the expansion phase continues by creating a regular vertex v_6. The merge phase does not change the graph in any way.

Level $V_3 = \{v_6\}$

All ET jobs are finished for vertex v_6, but there are still unfinished TT jobs $T_{1,2}$ and $T_{2,1}$. The fixation phase therefore creates a decision vertex w_7 and its root Bratley vertex $w_7.b_0$. This Bratley vertex is not viable, because all ET jobs are finished, but there are still unfinished TT jobs. The vertex $w_7.b_0$ is expanded, which results in vertices $w_7.b_1$ and $w_7.b_2$.

In case of $w_7.b_1$ TT job $T_{2,1}$ cannot be fixed at time $w_7.b_1.t = 16$ because the latest start time of $T_{2,1}$ is $T_{2,1}.d - T_{2,1}.c = 20 - 5 = 15 < 16$. Same logic applies for $w_7.b_2$ and the modified Bratley algorithm concludes that there is no solution for vertex w_7.

The expansion phase continues by backtracking to the latest created decision vertex, which in this case is w_4. Here, the modified Bratley algorithm is called again with the goal to find another combination of start times. Because the vertex $w_4.b_2$ and its expansions would result in a deadline miss, only vertex $w_4.b_1$ is expanded. The result is Bratley vertex $w_4.b_3$. This vertex is considered to be viable. Therefore, the fixation phase creates a regular vertex v_8 with $[v_8.e, v_8.l] = [13, 13] = [w_4.b_3.t, w_4.b_3.t]$. The expansion phase then creates vertex v_9 from v_8. Merge phase does not make any changes to the graph.

Level $V_4 = \{v_9\}$

Just as with level V_3, all ET jobs are finished, but there is still an unfinished TT job $T_{1,2}$. The fixation phase therefore creates a decision vertex w_{10} and its root Bratley vertex $w_{10}.b_0$. This vertex is not viable and is expanded, which results in vertex $w_{10}.b_1$. This vertex is viable and the result of the fixation phase is vertex v_{11}. Both expansion and merge phases make no changes to the graph.

Level $V_5 = \{v_{11}\}$

At this point, the fixation graph algorithm concludes, because there exists a regular vertex that does not cause a deadline miss to any ET job, has empty applicable ET jobs, and empty applicable TT jobs. In other words, all jobs are finished with no deadline misses.

Start times of the TT jobs are gathered by backtracking through the fixation graph. An undirected path is taken from one of the last created vertices to the root vertex. For each decision vertex encountered on this path, we backtrack in its Bratley graph to find the start times.

In this example, we encounter two decision vertices w_{10} and w_4 on the undirected path from vertex v_{11} to the root vertex v_0. In case of w_{10}, the last used Bratley vertex is $w_{10}.b_1$, which corresponds to fixing job $T_{1,2}$ at time $w_{10}.b_0.t = 16$. Therefore its start time is $S_{1,2} = 16$. Similarly with w_4, the last used Bratley vertex is $w_4.b_3$, which corresponds to fixing job $T_{2,1}$ at time $w_4.b_1.t = 8$, and then $T_{1,1}$ at time $w_4.b_0.t = 6$. Therefore $S_{1,1} = 6$ and $S_{2,1} = 8$.

Note that the fixation graph keeps vertices v_5, v_6, w_7 in the graph, even though they could have been deleted once vertex v_8 was created. We call these vertices *dead vertices*. In the formal description of the fixation graph generation algorithm, we chose to leave dead vertices in the fixation graph to make the algorithm more comprehensible.

5.4.6 New notation

This section summarizes new notation which will be used in the following sections to formally describe the fixation graph generation algorithm. Notation of different types of vertices can be seen in Table 5.2.

The *next ET release time* is defined as $w_i.t^e = max\{w_i.t, min\{E_{i,j}.r^{min} \mid E_{i,j} \in w_i.E^A\}\}$ for a decision vertex w_i. A decision vertex w_i also contains so called *next TT fixation time* denoted as $w_i.t^f$. This value is a heuristic guess of the next time a decision vertex will be created. Finally, a decision vertex contains a set of Bratley vertices $w.B$ and an index $w.idx$.

Both regular vertex v and decision vertex w contain a *previous decision vertex* denoted as $v.PD$ and $w.PD$. The previous decision vertex is defined as the first encountered decision vertex on an undirected path from the vertex v or w to the root vertex of the fixation graph. If no decision vertex is found, the value is *null*.

The process of computing applicable TT jobs for Bratley vertices differs from that of regular and decision vertices. For a Bratley vertex $w_i.b_j$ the path $T^{w_i.b_j}$ contains all TT jobs encountered when taking a path from the root vertex v_r to decision vertex w_i and then a path from root Bratley vertex $w_i.b_r$ to vertex $w_i.b_j$. Applicable TT jobs are then computed from $T^{w_i.b_j}$.

The Bratley vertex also contains two so far unmentioned variables. The *best incoming candidate* $w.b.best$ specifies which edge in $w.b.in$ resulted in the time $w.b.t$ during its creation. This value is used to compute *fixed TT jobs* $w.b.\mathcal{T}^f$. This set contains all edges encountered when taking a path from $w.b$ to the root Bratley vertex by only taking edges $w.b.best$.

During creation of a decision vertex w_i, the value $w.idx$ is set to 0 and $w.B$ is set to $\{w_i.b_r\}$ where $w_i.b_r.t = w_i.t$, $w_i.b_r.in = w_i.b_r.out = \emptyset$, and $w_i.b_r.best = null$.

5.4.7 A formal description of modified Bratley's algorithm

The pseudocode of the modified Bratley's algorithm can be seen in Algorithm 14. The algorithm always takes a decision vertex w_i as an input and uses its variables $w_i.B$ and $w_i.idx$. The function $MOD\_BRATLEY\_NEXT$ is called repeatedly to get different combinations. Function $GENERATE\_NEXT\_LEVEL$ operates on the same principle as the expansion and merge phase from the schedule graph generation algorithm. Function $EXPAND\_BRATLEY\_VERTEX$ operates on the same principle as function $EXPAND\_VERTEX$ from Algorithm 7.

A Bratley vertex is *viable* if it does not cause a deadline miss, all TT jobs which would miss their deadlines by $w.t^f$ are fixed, and fixing more TT jobs would interfere with ET jobs. This condition applies if there are still unfinished ET jobs. In case all ET jobs are finished, then a Bratley vertex is viable if all TT jobs are also finished and there are no deadline misses. Formally Bratley vertex $w_i.b_j$ is viable iff

$$\begin{aligned}(\nexists \sigma_i \in w_i.b_j.in \ st.\ \sigma_i.s.t + \sigma_i.T.c > \sigma_i.T.d) \ \wedge \ ((w_i.E^A = \emptyset \ \wedge \ w_i.b_j.T^A = \emptyset) \ \vee \\ (w_i.E^A \neq \emptyset \ \wedge \ \nexists T_{i,j} \in w_i.b_j.T^A \ st.\ w_i.t^f + T_{i,j}.c > T_{i,j}.d \ \wedge \\ \nexists T_{i,j} \in w_i.b_j.T^A \ st.\ max(w_i.b_j.t, T_{i,j}.r) + T_{i,j}.c \leq w_i.t^e))\end{aligned}$$

Use of this property can be seen on line 9.

One of the reasons the fixation graph is heuristic is the fact that the modified Bratley algorithm tries to finish all TT jobs as soon as possible. It may happen that a schedule results in a deadline miss for an optimal order of TT jobs, but does not for a sub-optimal one.

5.4.8 A heuristic approach of the fixation phase

As can be seen in the example in Chapter 5.4.5, fixing a TT job in the time interval $[2, 5]$ would result in a deadline miss. This is due to the release jitter and execution time variation of ET jobs and the properties of work-conserving policies. If a TT job was fixed before time $t < v_2.l = 4$, then in some scenario, an ET job would be executing at time t and the TT job may not begin its execution, which results in a deadline miss. If a TT was fixed at time $t = v_2.l = 4$ then an ET job may finish its execution before $t = 4$ and another ET job may start its execution and still be executing at time $t = 4$. The fixation phase takes this into account to fix TT jobs in places, where they would not cause a deadline miss in a similar manner. A more general rule follows.

Let $v_j \in V_i$. If $v_j.e = v_j.l \wedge |V_i| = 1$, then a TT job may be fixed at time $v_j.l$. This is due to the fact that the online scheduler would finish executing a job at time $v_j.l$ in every scenario and a TT job may begin execution at that time.

If $\forall v_j \in V_i \ \nexists E_{k,l} \in v_j.E^A \ st.\ E_{k,l}.r^{min} < v_j.l$, then a TT job may be fixed at time $l_{max} = max\{v_j.l \mid v_j \in V_i\}$. Because no unfinished ET jobs are released before l_{max} in every scenario and a TT job may begin its execution at time l_{max}.

Notation	Name	Type
$v.e$	earliest finish time	integer
$v.l$	latest finish time	integer
$v.in$	incoming edges	set of edges
$v.out$	outgoing edges	set of edges
$v.PD$	previous decision vertex	decision vertex
$v.E^A$	applicable ET jobs	set of jobs
$v.T^A$	applicable TT jobs	set of jobs

Notation	Name	Type
$w.t$	time	integer
$w.in$	incoming edges	set of edges
$w.out$	outgoing edges	set of edges
$w.B$	Bratley vertex level	set of Bratley vertices
$w.idx$	Bratley vertex level index	integer
$w.PD$	previous decision vertex	decision vertex
$w.E^A$	applicable ET jobs	set of ET jobs
$w.T^A$	applicable TT jobs	set of TT jobs
$w.t^e$	next ET release time	integer
$w.t^f$	next TT fixation time	integer

Notation	Name	Type
$w.b.t$	time	integer
$w.b.in$	incoming edges	set of edges
$w.b.best$	best incoming candidate	edge
$w.b.out$	outgoing edges	set of edges
$w.b.T^A$	applicable TT jobs	set of TT jobs
$w.b.\mathcal{T}^f$	fixed TT jobs	set of TT jobs

Table 5.2: All variables that are associated with a single regular, decision, and Bratley vertex. Variables E^A, T^A, $\mathcal{T}^f$, t^f, PD, and t^e are used to simplify the notation and terminology. They do not have to be stored and can be computed when needed using the other variables.

Algorithm 14 Modified Bratley's algorithm

Input: a decision vertex w_i
Output: a Bratley vertex

1: **function** MOD_BRATLEY_NEXT(w_i)
2: **while** true **do**
3: **if** $w_i.idx = |w_i.B|$ **then**
4: GENERATE_NEXT_LEVEL(w_i)
5: **if** $w_i.B = \emptyset$ **then**
6: **return** null
7: $w_i.b_j \leftarrow w_i.B[w_i.idx]$
8: $w_i.idx \leftarrow w_i.idx + 1$
9: **if** $w_i.b_j$ is viable **then**
10: **return** $w_i.b_j$
11: **function** GENERATE_NEXT_LEVEL(w_i)
12: $B^{ex} \leftarrow \emptyset$
13: **for each** $w_i.b_j \in w_i.B$ **do**
14: **if** $\exists \sigma_i \in w_i.b_j.in \ st. \ \sigma_i.s.t + \sigma_i.T.c > \sigma_i.T.d$ **then**
15: **continue** ▷ $w_i.b_j$ caused a deadline miss and is not expanded
16: **for each** $T_{i,j} \in w_i.b_j.T^A$ **do**
17: **if** $w_i.b_j.t < T_{i,j}.r \ \wedge \ w_i.t^e < T_{i,j}.r$ **then**
18: **continue** ▷ An ET job could be executing at $t = T_{i,j}.r$
19: $B^{ex} \leftarrow B^{ex} \cup \{EXPAND\_BRATLEY\_VERTEX(w_i.b_j, T_{i,j})\}$
20: **while** $\exists w_i.b_m, w_i.b_x \in B^{ex} \ st. \ w_i.b_m.T^A = w_i.b_x.T^A \ \wedge \ w_i.b_m.t \leq w_i.b_x.t$ **do**
21: **for each** edge $\sigma_x \in w_i.b_x.in$ **do**
22: $\sigma_x.d = w_i.b_m$
23: $w_i.b_m.in = w_i.b_m.in \cup \{\sigma_x\}$
24: remove $w_i.b_x$ and all edges $\{\sigma_x \mid \sigma_x.d = w_i.b_x\}$ from the graph
25: $w_i.B \leftarrow B^{ex}$
26: $w_i.idx \leftarrow 0$
27: **function** EXPAND_BRATLEY_VERTEX($w_i.b_j$, $T_{i,j}$)
28: $w_i.b_n \leftarrow$ new Bratley vertex with $w_i.b_n.t = max\{w_i.b_j.t, T_{i,j}.r\} + T_{i,j}.c$
29: $\sigma_n \leftarrow$ new edge with $\sigma_n.T = T_{i,j}$, $\sigma_n.s = w_i.b_j$ and $\sigma_n.d = w_i.b_n$
30: $w_i.b_n.in \leftarrow w_i.b_n.in \cup \{\sigma_n\}$
31: $w_i.b_n.best \leftarrow \sigma_n$
32: $w_i.b_j.out \leftarrow w_i.b_j.out \cup \{\sigma_n\}$
33: **return** $w_i.b_n$

	r	c	d^{TT}	τ^{TT}
$\mathcal{T}_1$	2	1	11	12

	r^{min}	r^{max}	c^{min}	c^{max}	d^{ET}	τ^{ET}
$\mathcal{E}_1$	0	1	2	3	12	12
$\mathcal{E}_2$	1	1	3	3	11	12
$\mathcal{E}_3$	2	2	1	2	10	12
$\mathcal{E}_4$	3	3	1	1	9	12

Table 5.3: Parameters of TT and ET jobs used in the fixation graph anomaly instance.

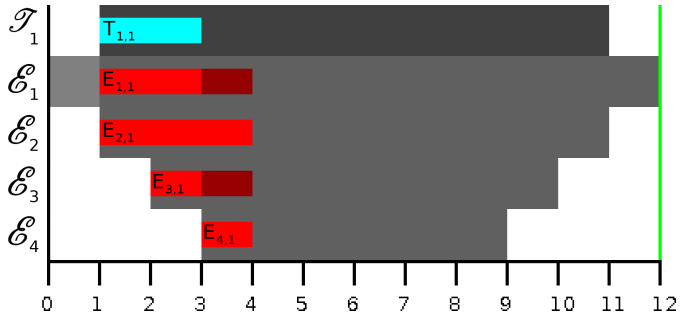


Figure 5.3: Loose Gantt chart of the fixation graph anomaly instance.

The fixation phase considers these two rules when creating a decision vertex. If neither of the conditions is met, no decision vertex is created, meaning that no TT jobs are fixed at this stage.

The fixation does not find every possible time, where a TT job may be fixed. We show this in an example specified in Table 5.3. In this example, the fixation graph generation algorithm runs under the EDF scheduling policy. Therefore priority values of tasks are undefined. The loose Gantt chart of the instance can be seen in Figure 5.3 and the resulting fixation graph in Figure 5.4. Here the TT job $T_{1,1}$ may be fixed at time $S_{1,1} = 4$, because no job will be executing at time 4 no matter the execution scenario. However, the fixation phase does not detect this possible start time. Due to this fact, the fixation graph generation algorithm finds no solution despite the fact that a solution exists.

As previously discussed, a decision vertex w_i contains next TT fixation time $w_i.t^f$. This value is computed by simulating a scenario, where no TT jobs are fixed by decision vertex w_i and ET jobs exhibit the worst case behaviour, i.e., all ET jobs release at their latest release time, and their execution time is the worst case execution time. Then time $w_i.t^f$ corresponds to the earliest time any job $E_{i,j} \in w_i.E^A$ has been executed and no ET jobs are released before $w_i.t^f$. If $w_i.E^A = \emptyset$ then $w_i.t^f = \infty$.

In the case of decision vertex w_4 in example 5.4.5, $w_4.t^f = E_{1,2}.r^{max} + E_{1,2}.c^{max} = 14$. Note that at this time both of the tasks $T_{2,1}$ and $T_{1,2}$ may be individually fixed at $w_4.t^f = 14$ without missing their deadlines, but both of them cannot be fixed without a deadline miss. Due to the way a viable Bratley vertex is defined, the decision vertex w_4 still attempts to not fix job $T_{2,1}$ earlier and the fixation phase has to later backtrack to vertex w_4.

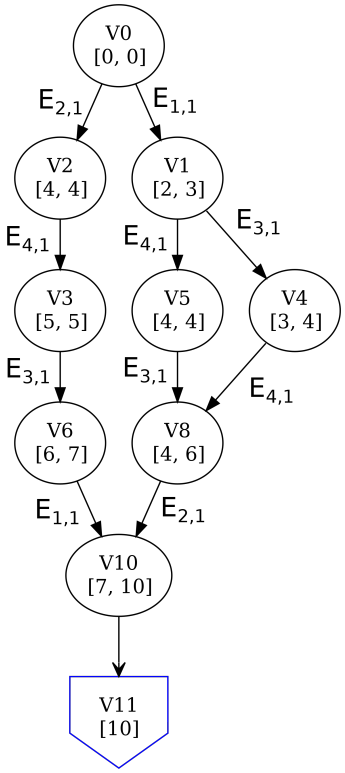


Figure 5.4: Fixation graph for the anomaly instance. Bratley vertices are not shown in this visualization.

5.4.9 A formal description of the fixation phase

Pseudocode of the fixation phase can be seen in Algorithm 15. For a set of vertices V_i the fixation phase returns a vertex v_j and integer ST. The ST variable determines how the fixation graph generation will continue. There are four possible values:

- 0 - use $\{v_j\}$ as input for the expansion phase
- 1 - use V_i as input for the expansion phase
- 2 - a solution has been found, the algorithm ends
- 3 - no solution has been found, the algorithm ends

When the fixation phase returns a vertex v_j, its parent is a decision vertex. This decision vertex has either just been created in the fixation phase or was already part of the fixation graph and the fixation phase backtracked to it.

5.4.10 Fixation graph generation algorithm

A pseudocode of the fixation graph generation algorithm can be seen in Algorithm 16. Function $GATHER\_START\_TIMES$ is filling a set of start times S. An element of the set is accessed as $S_{i,j}$ for job $T_{i,j}$.

Algorithm 15 Fixation phase

Input: a set of regular vertices V
Output: a regular vertex v and integer ST

1: **function** FIXATION_PHASE(V)
2: **if** $\forall v_i \in V \nexists \sigma_i \in v_i.in \ st. \ v_i.l > \sigma_i.T.d$ **then** ▷ If no deadline miss
3: **if** $\forall v_i \in V \, v_i.E^A = \emptyset \ \wedge \ v_i.T^A = \emptyset$ **then** ▷ If all jobs are completed
4: **return** $(null, 2)$
5: $l_{max} \leftarrow max\{v_i.l \mid v_i \in V\}$
6: **if** $(|V| = 1 \ \wedge \ V[0].e = V[0].l) \ \vee \ (\forall v_i \in V \nexists E_{i,j} \in v_j.E^A \ st. \ E_{i,j}.r^{min} < l_{max})$ **then** ▷ If can create a decision vertex
7: $w_n \leftarrow$ new decision vertex with $w_n.t = l_{max}$
8: **for** $v_i \in V$ **do**
9: $\sigma_n \leftarrow$ new edge with $\sigma_n.s = v_i, \sigma_n.d = w_n$
10: $v_i.out \leftarrow v_i.out \cup \{\sigma_n\}$
11: $w_n.in \leftarrow w_n.in \cup \{\sigma_n\}$
12: $w_n.b_m \leftarrow MOD\_BRATLEY\_NEXT(w_n)$
13: **if** $w_n.b_m \neq null$ **then**
14: $v_n \leftarrow GET\_REGULAR\_VERTEX(w_n.b_m)$
15: **return** $(v_n, 0)$
16: **else**
17: **return** $(null, 1)$
18: $w_i \leftarrow V[0].PD$
19: **while** $w_i \neq null$ **do**
20: $w_n.b_m \leftarrow MOD\_BRATLEY\_NEXT(w_n)$
21: **if** $v_n \neq null$ **then**
22: $v_n \leftarrow GET\_REGULAR\_VERTEX(w_n.b_m)$
23: **return** $(v_n, 0)$
24: $w_i \leftarrow w_i.PD$
25: **return** $(null, 3)$
26: **function** GET_REGULAR_VERTEX($w_n.b_m$)
27: $v_n \leftarrow$ new regular vertex with $v_i.e = v_i.l = w_n.b_m.t$
28: $\sigma_n \leftarrow$ new edge with $\sigma_n.s = w_n, \sigma_n.d = v_n, \sigma_n.\mathcal{T} = w_n.b_m.\mathcal{T}^f$
29: $w_n.out \leftarrow w_n.out \cup \{\sigma_n\}$
30: $v_n.in \leftarrow \{\sigma_n\}$
31: **return** v_n

Algorithm 16 Fixation graph generation algorithm

Input: a set of TT tasks $\mathcal{T}$, a set of ET tasks $\mathcal{E}$
Output: start times S or null

1: **function** FIXATION_GRAPH_GENERATION($\mathcal{T}, \mathcal{E}$)
2: $v_r \leftarrow$ new regular vertex with $v_r.e = v_r.l = 0$ and $v_r.in = v_r.out = \emptyset$
3: $V \leftarrow \{v_r\}$
4: $i \leftarrow 0$
5: **while** true **do**
6: $[v_f, ST] \leftarrow$ FIXATION_PHASE(V)
7: **if** $ST = 3$ **then** ▷ No solution found
8: **return** null
9: **if** $ST = 2$ **then** ▷ All jobs are finished with no deadline misses
10: **return** GATHER_START_TIMES(V)
11: **if** $ST = 1$ **then** ▷ No decision vertex created
12: $V^{ex} \leftarrow$ EXPANSION_PHASE(V)
13: **if** $ST = 0$ **then** ▷ Decision vertex created
14: $V^{ex} \leftarrow$ EXPANSION_PHASE($\{v_f\}$)
15: $V \leftarrow$ MERGE_PHASE(V^{ex})
16: **function** GATHER_START_TIMES(V)
17: $S \leftarrow$ empty start times
18: $w_i \leftarrow V[0].PD$
19: **while** $w_i \neq null$ **do**
20: $w_i.b_j \leftarrow$ last vertex returned by $MOD\_BRATLEY\_NEXT(w_i)$
21: **while** $w_i.b_j.best \neq null$ **do**
22: $\sigma_i \leftarrow w_i.b_j.best$
23: $T_{x,y} \leftarrow \sigma_i.T$
24: $w_i.b_j \leftarrow \sigma_i.s$
25: $S_{x,y} \leftarrow w_i.b_j.t$
26: $w_i \leftarrow w_i.PD$
27: **return** S

5.4.11 An exact algorithm

There are two issues that need to be addressed in order to make this algorithm exact. First, the algorithm needs to find all possible times, where a TT job may begin execution. Second, the modified Bratley algorithm needs to account even for non-optimal ways of fixing TT jobs. Although we have yet to come up with a scalable solution to the second problem, we have an idea of how to solve the first one.

The first problem may be solved by changing the fixation phase, so that it flags times, where an ET job is executing in some scenario. This information can be gathered when generating the fixation graph. For every edge σ_i where $\sigma_i.E.c^{max} > 1$ and both $\sigma_i.s$ and $\sigma_i.d$ are non-dead regular vertices, no TT jobs may be fixed in time range $[t^{EC}+1, t^{LC}+\sigma_i.E.c^{max}-1]$, where t^{EC} is the earliest time a job $\sigma_i.E$ becomes eligible during $EXPANSION\_PHASE(\sigma_i.s)$ and t^{LC} is the latest time a job $\sigma_i.E$ is eligible during $EXPANSION\_PHASE(\sigma_i.s)$. Note that for work-conserving policies, once a job stops being eligible in the expansion phase, it may not become eligible again. Therefore there is only one such time t^{EC} and t^{LC}.

Once there exists an unflagged time t^{NF} such that $t^{NF} \leq max\{v_j.l \mid v_j \in V_i\}$ then a TT job may be fixed at time t^{NF}. Using this approach, the fixation phase may need to work with vertices on multiple levels. In the fixation graph in Figure 5.4 on level $V_2 = \{v_3, v_4, v_5\}$ an unflagged time $t^{NF} = 4$ would be found. However, the newly created decision vertex would not have parent vertices v_3, v_4, v_5 but parent vertices v_2, v_4, v_5. Additionally, vertex v_3 would have to be removed from the graph. The fixation phase would also need to return multiple vertices instead of just one.

Chapter 6

ET+TT solution evaluation

In this section, we evaluate the algorithms described in Chapter 5. Once again, we are doing empiric measurements on randomly generated instances. The measured algorithms were implemented using Java 8. These are the relevant specifications of the computer the benchmarking was done on:

- CPU: 2600Mhz, 14 cores, 14 threads
- Memory: 516GB, DDR4

Unlike in Chapter 4, multiple tests ran at the same time on one CPU. Each test ran on a single core.

6.1 Instance generation

An algorithm that generates random instances was already presented in Chapter 4. This algorithm generates instances with only ET tasks. Therefore, we created a modified version of the algorithm, which generates random instances with only TT tasks. The algorithm is almost equivalent to generating ET tasks, removing the release jitter, execution time variation, and priority. It also takes the same arguments as the ET task generation algorithm, except it no longer takes arguments for release jitter and execution time variation as well as minimum and maximum priority. The result of the TT task generation algorithm is then combined with the result of the ET task generation algorithm to create an instance with both ET and TT tasks. For each instance the seed used for generating TT tasks always differed from the seed used for generating ET tasks.

6.2 Brute force evaluation (No jitter)

We generated a single dataset $\mathcal{D}_1^b$ which contains 5600 instances with both ET and TT tasks. The TT tasks were generated with arguments $1 \leq A^{\mathcal{E}} \leq 20$, $A^{\eta} = 200$, $A^{\tau} = 50$, $A^{U} = 0.4$, $A_s^U = 300$, $A_a^U = 0.1$, $A_d^P = A_r^P = 0.4$, $A^P = 0$. The ET tasks were generated with arguments $A^{\mathcal{E}} = 10$, $A^{\eta} = 200$, $A^{\tau} = 40$, $A^{U} = 0.4$, $A_s^U = 300$, $A_a^U = 0.1$, $A_j^P = A_c^P = 0.3$, $A_d^P = A_r^P = 0.4$, $A^P = 0$, $A_{min}^p = 1$, $A_{max}^p = 3$.

To summarize, each instance has 10 ET tasks with both release jitter and execution time variation and the number of TT tasks is variable between instances from 1 to 20 TT tasks. The hyperperiod of the instance is only 200. The number of possible TT task fixation combinations increases drastically with a higher hyperperiod. This means that the run time of the brute force algorithm would be exceedingly high for higher hyperperiods even for a very low number of tasks. To show how the number of tasks affects the run time we have chosen a very low hyperperiod.

6.2.1 Results

The resulting run times for the brute force algorithm with an overlap check and fixation without jitter on dataset $\mathcal{D}_1^b$ can be seen in Figure 6.1. The distribution of instance schedulability can be seen in 6.2. The scheduling was performed under the CP policy. The algorithm was terminated if its run time exceeded 1 hour.

As can be seen in the figures, the brute force algorithm often ends in a timeout even for 10 TT tasks, but sometimes finishes within the 1-hour time limit even for 20 TT tasks. Due to the fact that the hyperperiod and utilization was fixed for all tasks, instances with a few TT tasks had TT jobs with large execution times. On the other hand, instances with many TT tasks had jobs with small execution times.

6.3 Brute force run time and fixation graph schedulability evaluation (fixation with jitter)

We generated a single dataset $\mathcal{D}_2^b$ which contains 11200 instances with both ET and TT tasks. The TT tasks were generated with arguments $1 \leq A^{\mathcal{E}} \leq 5$, $A^{\eta} = 100$, $A^{\tau} = 25$, $A^{U} = 0.35$, $A_s^U = 300$, $A_a^U = 0.1$, $A_d^P = A_r^P = 0.3$, $A^P = 0$. The ET tasks were generated with arguments $A^{\mathcal{E}} = 3$, $A^{\eta} = 100$, $A^{\tau} = 20$, $A^U = 0.35$, $A_s^U = 300$, $A_a^U = 0.1$, $A_j^P = A_c^P = 0.5$, $A_d^P = A_r^P = 0.3$, $A^P = 0.5$, $A_{min}^p = 1$, $A_{max}^p = 3$.

To summarize, each instance has 3 ET tasks with both release jitter and execution time variation and the number of TT tasks is variable between instances from 1 to 5 TT tasks. Brute force fixation with jitter is more time demanding than fixation without jitter because the combinatorial explosion increases with the number of jobs instead of the number of tasks. Due to this fact, the dataset $\mathcal{D}_2^b$ contains arguably simpler instances than dataset $\mathcal{D}_1^b$.

Both the brute force algorithm with overlap check and the fixation graph generation algorithm were run on dataset $\mathcal{D}_2^b$. The brute force algorithm was fixing TT tasks with jitter.

6.3.1 Results

The resulting run times on dataset $\mathcal{D}_2^b$ for the brute force algorithm can be seen in Figure 6.3 and for the fixation graph generation algorithm in Figure 6.5. The distribution of instances schedulability for the brute force algorithm can be seen in Figure 6.4 and for the fixation graph generation algorithm in Figure 6.6. The scheduling was done under the EDF-FP policy. The brute force algorithm was terminated if its run time exceeded

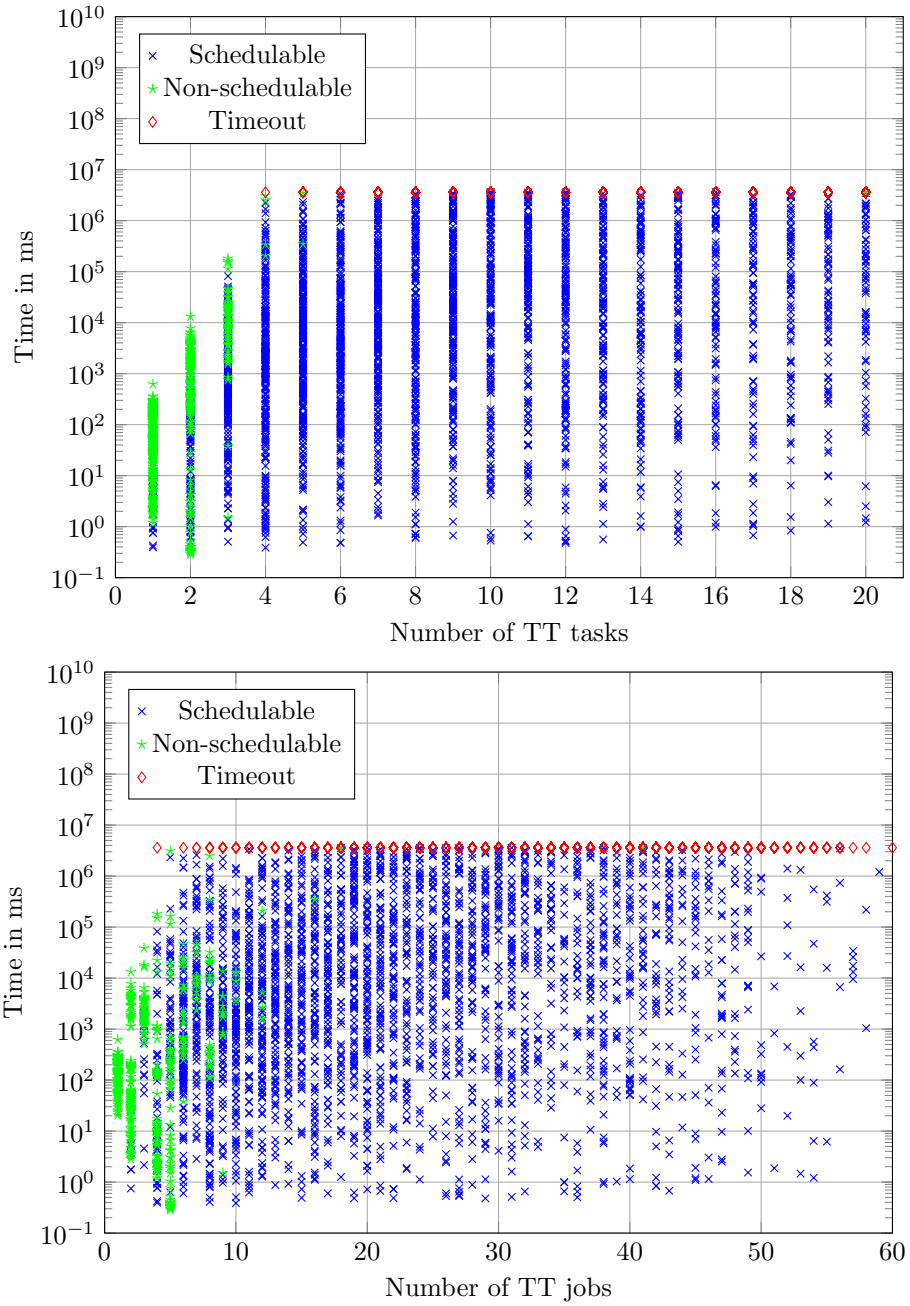


Figure 6.1: Run times of the ET+TT brute force algorithm on dataset $\mathcal{D}_1^b$. The top chart shows run times based on the number of tasks. The bottom chart shows run times based on the number of jobs.

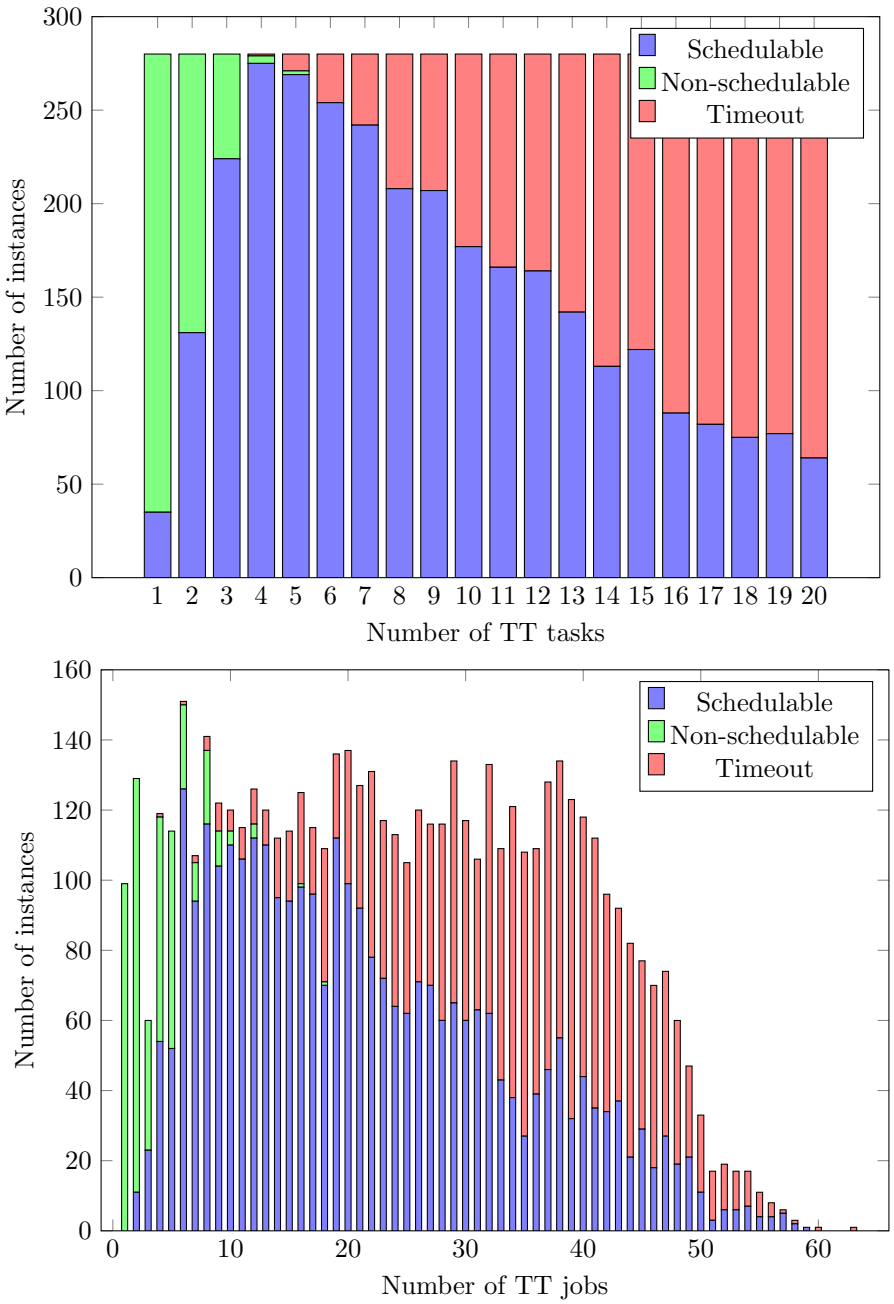


Figure 6.2: The number of schedulable, non-schedulable, and timeout instances for dataset $\mathcal{D}_1^b$. The top chart shows instances based on the number of tasks. The bottom chart shows instances based on the number of jobs.

	All instances	BF Timeout only	BF Schedulable only	BF Non-Schedulable only
BF Timeout	4159	4159	0	0
BF Schedulable	4494	0	4494	0
BF Non-Schedulable	2547	0	0	2547
FG Schedulable	6644	2250	4394	0
FG Non-Schedulable	4556	1909	100	2547

Table 6.1: Results of the Brute force (BF) and fixation graph (FG) algorithms schedulability.

1 hour. Additionally, the schedulability of the instances between the two algorithms can be seen in Table 6.1. The fixation graph generation algorithm found a solution in 97.8 % of instances where a solution exists.

6.4 Fixation graph run time evaluation (fixation with jitter)

We generated a single dataset $\mathcal{D}_3^b$ which contains 10080 instances with both ET and TT tasks. The TT tasks were generated with arguments $1 \leq A^{\mathcal{E}} \leq 60$, $A^{\eta} = 10000000$, $A^{\tau} = 1000000$, $A^{U} = 0.25$, $A_s^U = 300$, $A_a^U = 0.1$, $A_d^P = A_r^P = 0.2$, $A^P = 0$. The ET tasks were generated with arguments $A^{\mathcal{E}} = 20$, $A^{\eta} = 10000000$, $A^{\tau} = 1000000$, $A^{U} = 0.25$, $A_s^U = 300$, $A_a^U = 0.1$, $A_j^P = A_c^P = 0.3$, $A_d^P = A_r^P = 0.2$, $A^P = 0$, $A_{min}^p = 1$, $A_{max}^p = 3$.

To summarize, each instance has 20 ET tasks with both release jitter and execution time variation and the number of TT tasks is variable between instances from 1 to 60 TT tasks. The hyperperiod is also exceedingly large to show that the heuristic algorithm can handle large hyperperiods, unlike the brute force algorithm.

The evaluated implementation of the fixation graph generation algorithm does not keep dead vertices in memory. Unlike all previous measurements, the maximum available memory the application can use is 16GB, instead of the usual 4GB.

6.4.1 Results

The resulting run times for the fixation graph generation algorithm on dataset $\mathcal{D}_3^b$ can be seen in Figure 6.7. The fixation graph generation algorithm was terminated if its run time exceeded 1 hour. In case the algorithm exceeded the maximum allowed memory requirements, it was terminated. These instances are labeled as *Memout*.

The fixation graph generation algorithm has much higher memory requirements than the schedule graph generation algorithm. This is due to the fact that the schedule graph generation algorithm needs to keep only two levels of the schedule graph in memory, while the fixation graph generation algorithm is keeping all levels of the fixation graph.

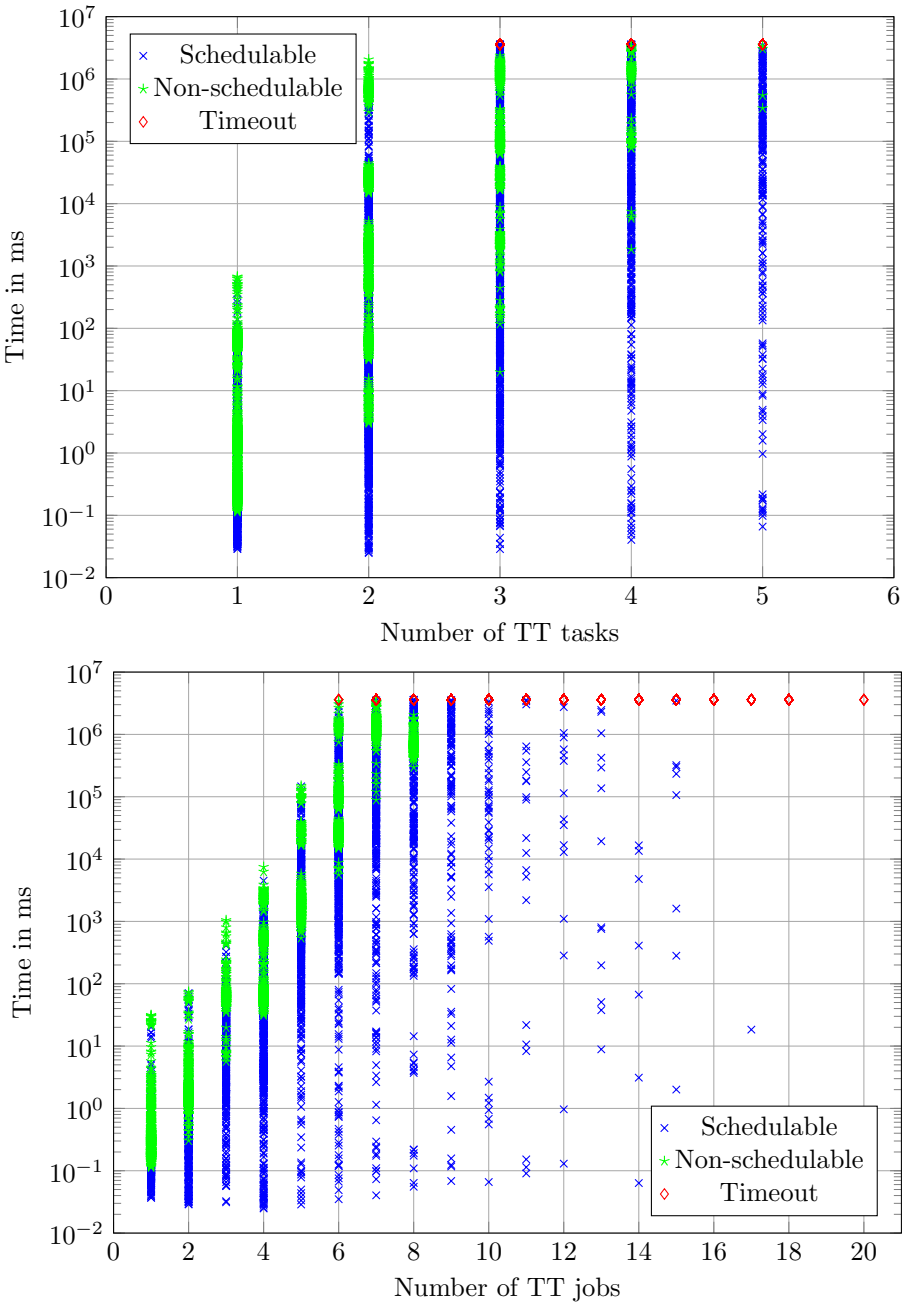


Figure 6.3: Run times of the ET+TT brute force algorithm on dataset $\mathcal{D}_2^b$. The top chart shows run times based on the number of tasks. The bottom chart shows run times based on the number of jobs.

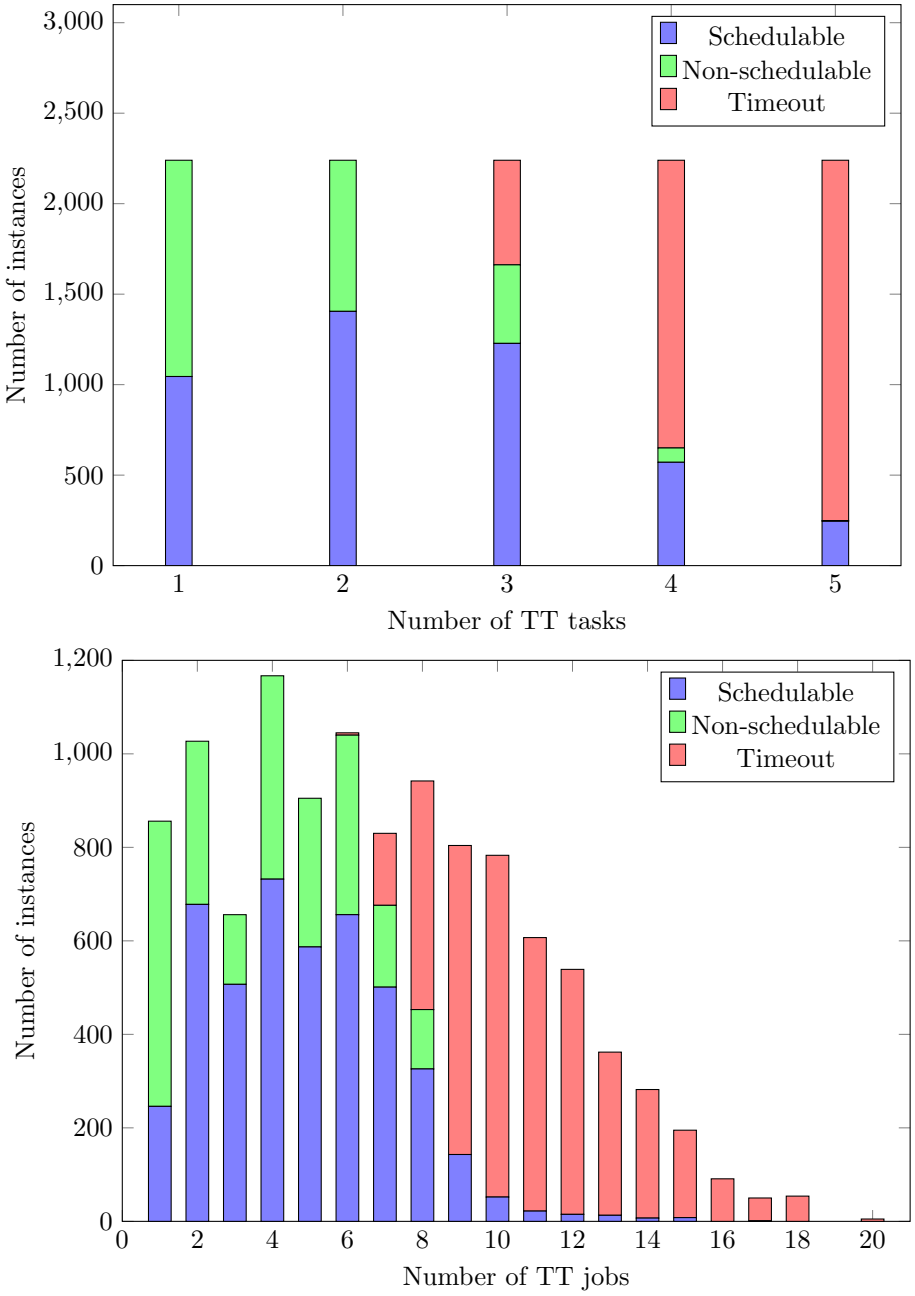


Figure 6.4: The number of schedulable, non-schedulable, and timeout instances for ET+TT brute force algorithm on dataset $\mathcal{D}_2^b$. The top chart shows instances based on the number of tasks. The bottom chart shows instances based on the number of jobs.

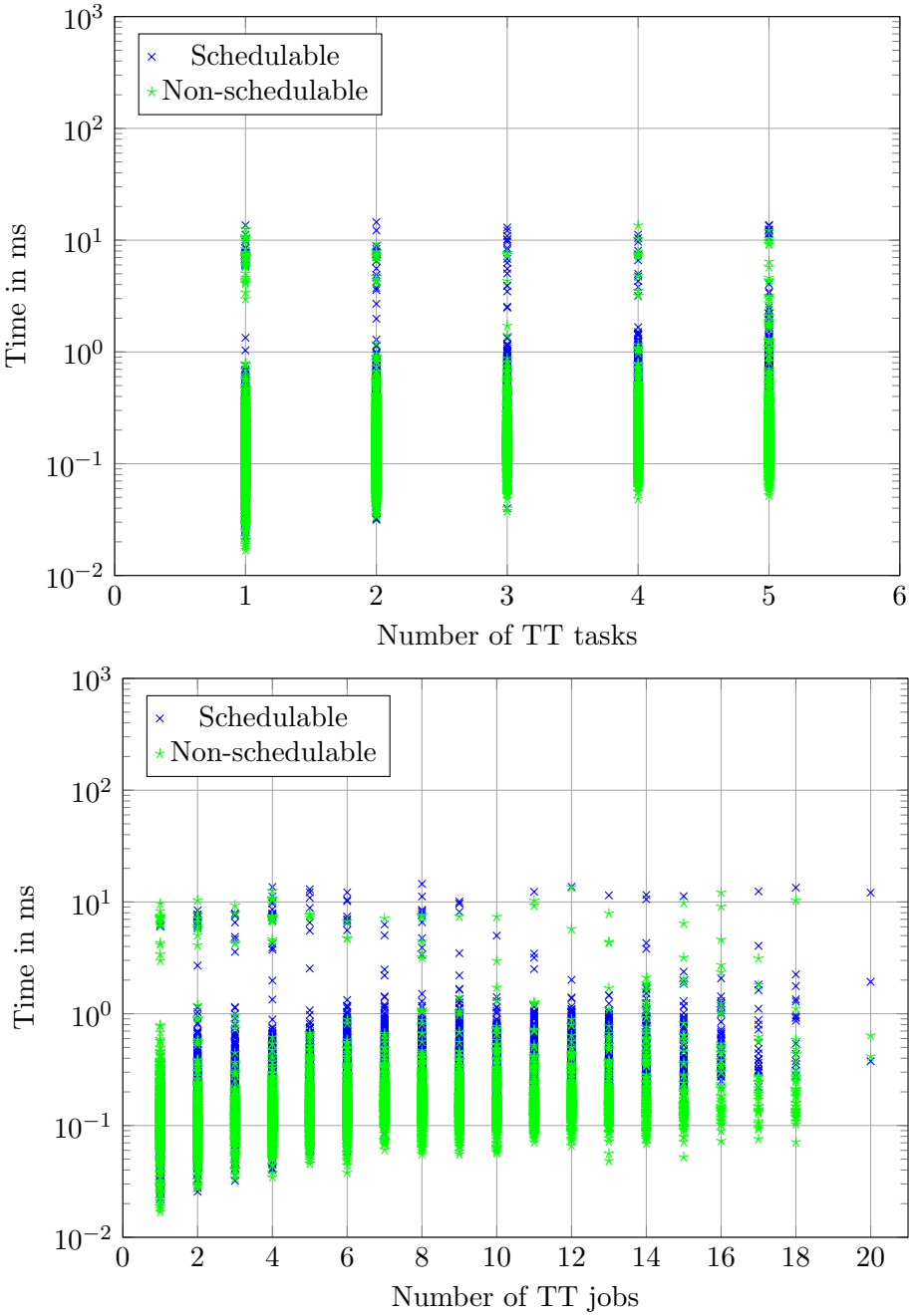


Figure 6.5: Run times of the fixation graph generation algorithm on dataset $\mathcal{D}_2^b$. The top chart shows run times based on the number of tasks. The bottom chart shows run times based on the number of jobs.

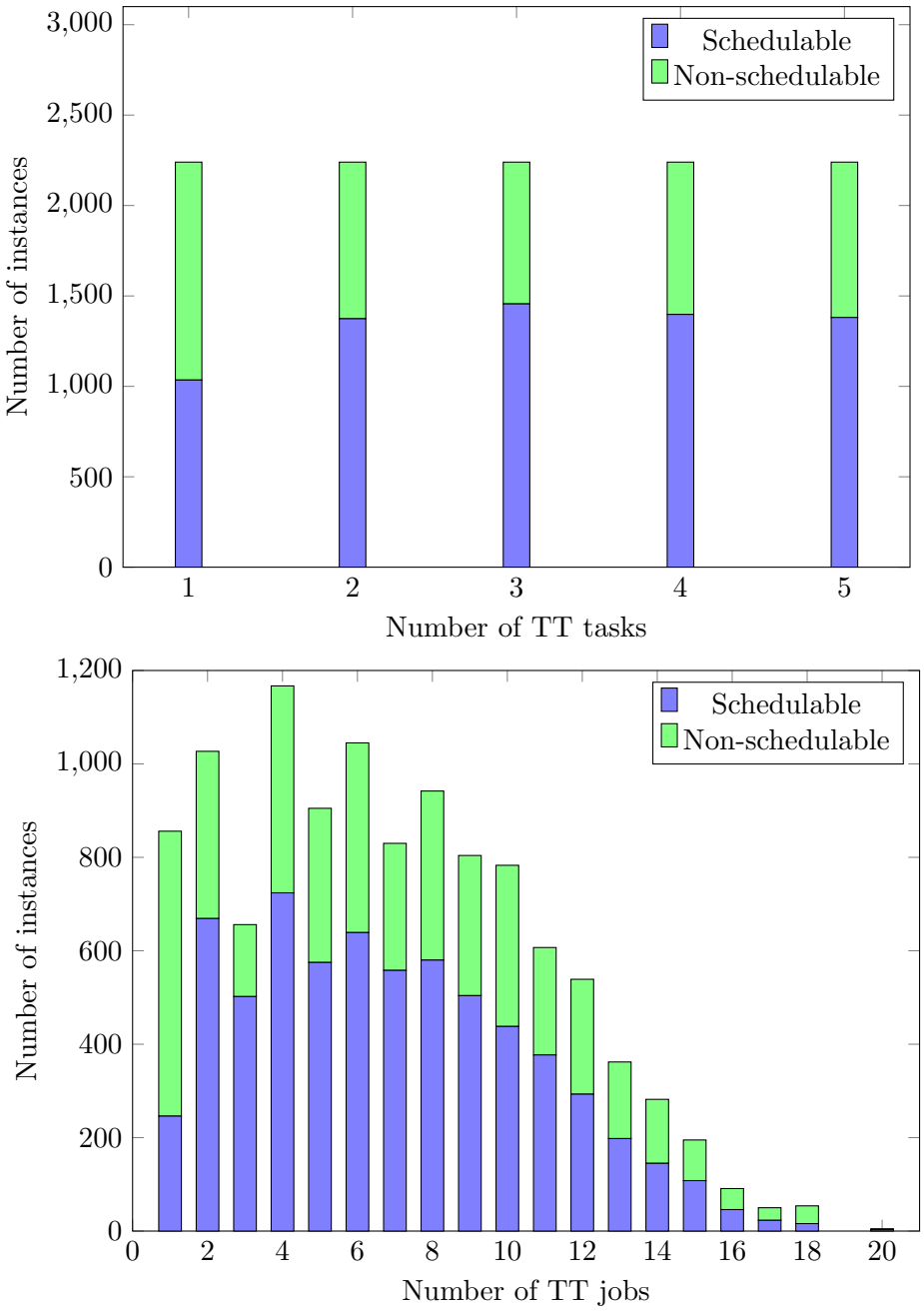


Figure 6.6: The number of schedulable, non-schedulable, and timeout instances for the fixation graph generation algorithm on dataset $\mathcal{D}_2^b$. The top chart shows instances based on the number of tasks. The bottom chart shows instances based on the number of jobs.

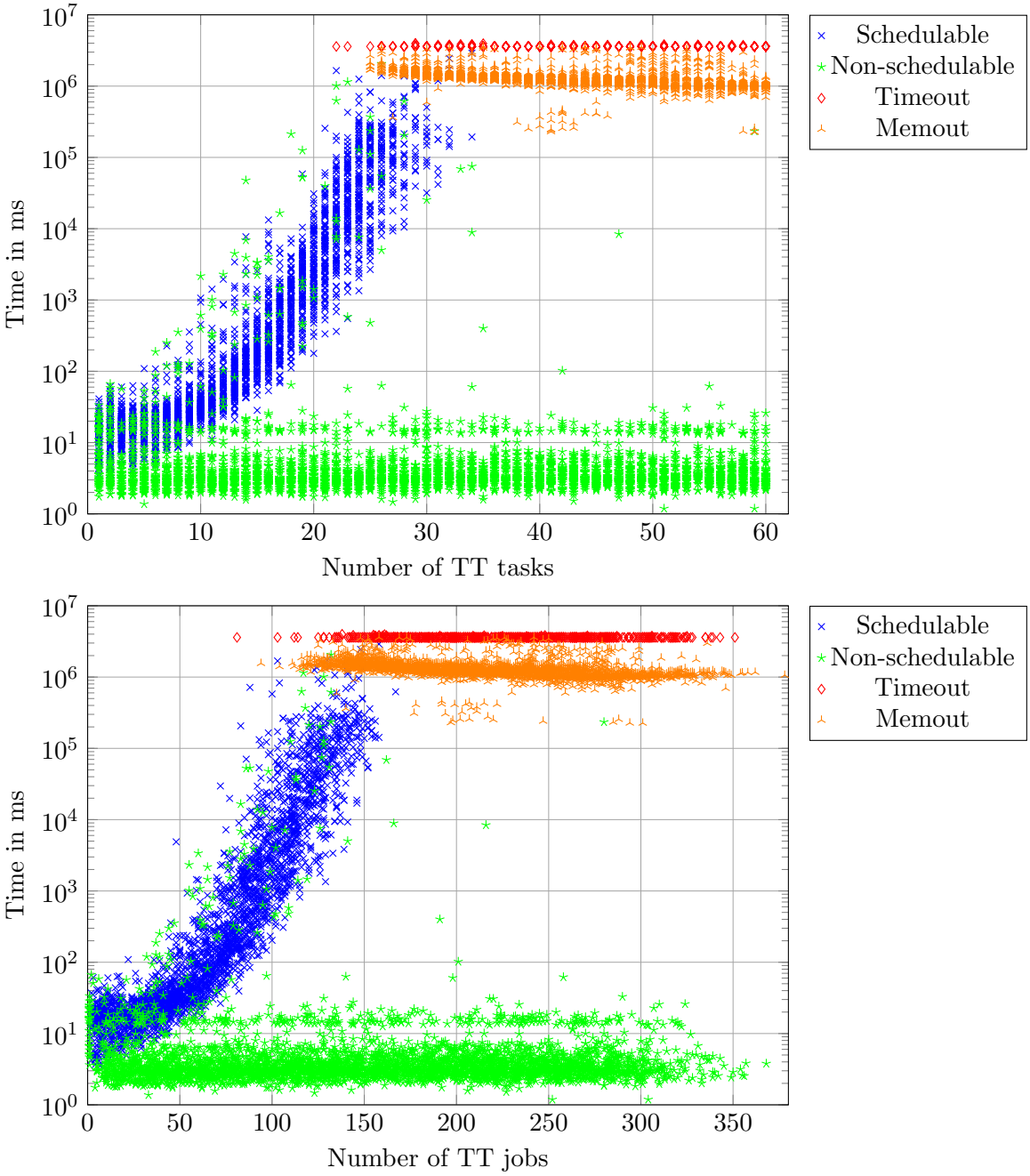


Figure 6.7: Run times of the fixation graph generation algorithm on dataset $\mathcal{D}_3^b$. The top chart shows run times based on the number of tasks. The bottom chart shows run times based on the number of jobs.

Chapter 7

Conclusion

This thesis formulated a problem of scheduling non-preemptive event-triggered and time-triggered tasks on a uniprocessor. We have described a brute force algorithm that implements an ET schedulability test for any scheduling policy. The policies we used were EDF-FP, P-RM, CP, and CW. Then we described an ET schedulability test that uses a schedule graph, first for work-conserving and then non-work-conserving policies.

We compared our approach to that of [12] in terms of run time and our implementation was faster by an order of magnitude. Additionally, our approach is exact even for non-work-conserving policies. We also measured the schedulability and run times of the schedule graph generation algorithm on different policies. The EDF-FP policy was the fastest but the worst in terms of schedulability. The P-RM and CP policies were around twice as slow as the EDF-FP policy and provided a slightly higher schedulability. The CW policy was around 4 times slower than the EDF-FP policy but had the highest schedulability rate. Additionally, the schedulability of CW policy was unaffected by an increasing amount of release jitter and execution time variation, unlike all other policies.

The second part of the thesis describes and evaluates algorithms for finding a valid set of start times for TT tasks. We described an exact brute force algorithm and a heuristic algorithm that may return false negatives. This heuristic algorithm is a modified version of the schedule graph generation algorithm.

We evaluated the run times of the brute force algorithm with and without fixation jitter. In the case of fixation without jitter, the brute force algorithm ran for longer than 1 hour for instances with 10 TT tasks in 40% of instances and for instances with 20 TT tasks in 80% of instances. In the case of fixation with jitter, the brute force algorithm ran for longer than 1 hour for instances with 3 TT tasks in 30% of instances and for instances with 4 TT tasks in 70% of instances. The heuristic algorithm found a solution in 97.8% of instances, where a solution existed. Finally, the heuristic algorithm was generally able to solve instances of 20 TT tasks and 20 ET tasks in a matter of seconds.

Bibliography

[1] Amos Albert and Robert Bosch. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. 2004.

[2] L. Almeida, P. Pedreiras, and J.A.G. Fonseca. The FTT-CAN protocol: why and how. *IEEE Transactions on Industrial Electronics*, 49(6):1189–1201, 2002.

[3] Sanjoy Baruah and Alan Burns. Sustainable Scheduling Analysis. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 159–168, 2006.

[4] Jacek Błażewicz, Klaus H Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. *Scheduling computer and manufacturing processes*. springer science & Business media, 2001.

[5] Josep Díaz, Jordi Petit, and Maria Serna. A Survey of Graph Layout Problems. *ACM Comput. Surv.*, 34(3):313–356, sep 2002.

[6] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In P.L. Hammer, E.L. Johnson, and B.H. Korte, editors, *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 287–326. Elsevier, 1979.

[7] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 207–216, 2000.

[8] Yuichi Itami, Tasuku Ishigooka, and Takanori 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*, pages 45–54, 2008.

[9] Gabriel Leen and D Heffernan. TTCAN: A new time-triggered controller area network. *Microprocessors and Microsystems*, 26:77–94, 03 2002.

[10] Marc van Kreveld Mark de Berg, Otfried Cheong and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, Heidelberg, 2008.

[11] Ayman Murshed, Roman Obermaisser, Hamidreza Ahmadian, and Ala Khalifeh. Scheduling and allocation of time-triggered and event-triggered services for multi-core processors with networks-on-a-chip. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1424–1431, 2015.

[12] Mitra Nasri and Bjorn B. Brandenburg. An Exact and Sustainable Analysis of Non-preemptive Scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23, 2017.

[13] Mitra Nasri and Mehdi Kargahi. Precautious-RM: A predictable non-preemptive scheduling algorithm for harmonic tasks. *Real-Time Systems*, 50, 06 2014.

[14] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:23, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks Under Global Scheduling. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] 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)*, pages 67–75, 2000.

[17] Traian Pop, Petru Eles, and Zebo Peng. Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems. *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, pages 187–192, 2002.

[18] Ricardo M. Vaz, Kleber N. Hodel, Max M.D. Santos, Benedito A. Arruda, Marcio Lobo Netto, and João F. Justo. An efficient formulation for optimization of FlexRay frame scheduling. *Vehicular Communications*, 24:100234, 2020.

[19] Shixi Wen, Ge Guo, and Wing Shing Wong. Hybrid Event-Time-Triggered Networked Control Systems: Scheduling-Event-Control Co-design. *Information Sciences*, 305, 02 2015.

Appendix A

A discrepancy in schedule graph generation algorithm proposed by M. Nasri and B. Brandenburg

To demonstrate an error in the scheduling of non-work-conserving policies in [12], we present an instance that yields different results for our approach and the approach of [12]. This instance contains 4 ET tasks whose properties can be seen in Table A.1. The loose Gantt chart of the instance can be seen in Figure A.1. The instance will be scheduled under the P-RM policy.

The schedule graph generated using our method can be seen in Figure A.2 and the schedule graph generated using the method from [12] can be seen in Figure A.3.

The schedule graph generated in Figure A.3 yields a deadline miss while the schedule graph generated in Figure A.2 does not. We believe that the issue with the approach in [12] is that eligibility of each job $E_{i,j}$ is checked only in time $t_E = max(v_n.e, E_{i,j}.r^{min})$ as according to Definition 5 from [12]. This means that once a job stops being eligible, it cannot become eligible again.

A.1 Accounting for differences between the approaches

As previously mentioned, we will be using the P-RM policy. Throughout the entire schedule graph generation process, the only critical job may be job $E_{1,1}$ as it is the only job with priority $p = 0$. In the formulation of the P-RM policy from [12], the critical job is the job with the lowest priority value, which in this case is also $E_{1,1}$. This means that

	r^{min}	r^{max}	c^{min}	c^{max}	d^{ET}	τ^{ET}	p
$\mathcal{E}_1$	10	10	2	2	12	16	0
$\mathcal{E}_2$	0	0	1	8	8	16	1
$\mathcal{E}_3$	1	1	2	2	14	16	3
$\mathcal{E}_4$	3	3	4	4	16	16	2

Table A.1: Parameters of the demonstration instance.

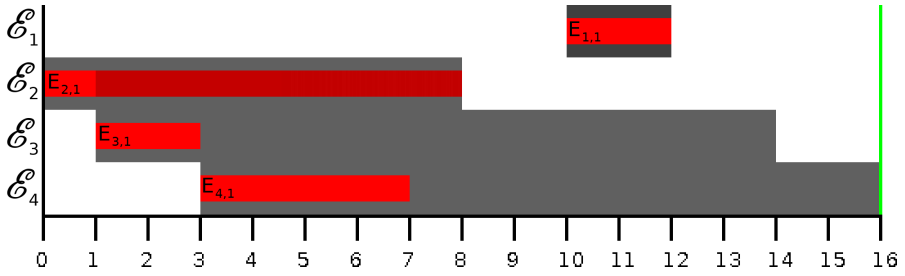


Figure A.1: A loose Gantt chart of the demonstration instance.

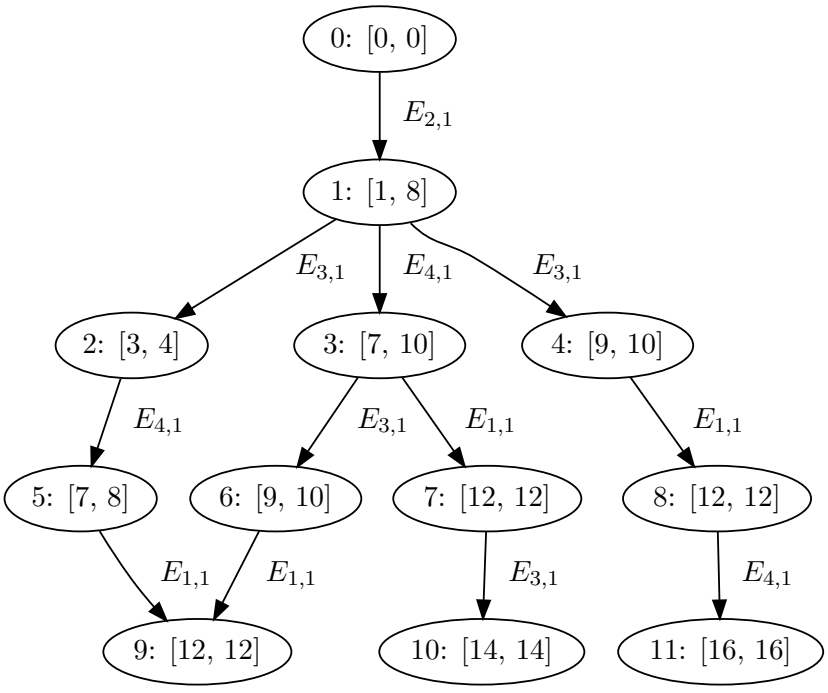


Figure A.2: Schedule graph generated using our method. The algorithm does not yield a deadline miss.

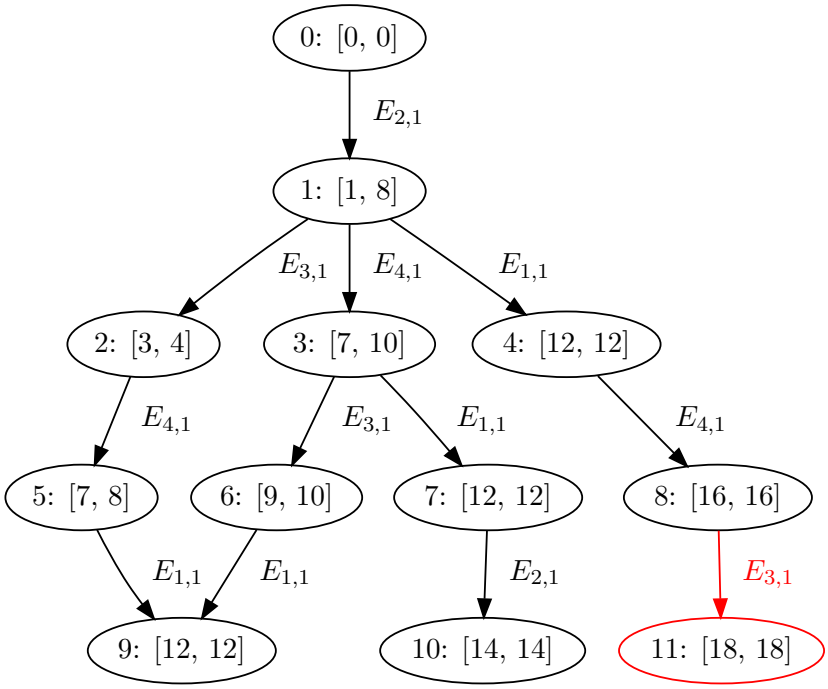


Figure A.3: Schedule graph generated using the method described in [12]. The algorithm yields a deadline miss.

$v.t^c = E_{1,1}.d - E_{1,1}.c^{max} = 10$ and $v.E^c = E_{1,1}$ for all vertices v where $E_{1,1}$ is unfinished.

Additionally, in the formulation [12] job priority is strictly determined by p value, i.e., by the Fixed Priority policy. To make sure that the job priority is the same as in our case, we gave each job a unique priority value. This means that our approach will also schedule jobs under the Fixed Priority policy.

A.2 Differences in schedule graph generation

Let us now explain both approaches of the schedule graph generation process on instance described in Table A.1. Both approaches begin by scheduling job $E_{2,1}$ as it is the only certainly released job at time $t = 0$ and it also does not violate the critical time because $E_{2,1}.c^{max} + t \leq v_0.t^c$, i.e., $8 + 0 \leq 10$. In both cases, the expansion phase of V_0 results in a single new vertex v_1 with $v_1.e = 1$ and $v_1.l = 8$.

Where the two approaches differ is during the expansion of vertex v_1. Here, job $E_{3,1}$ is eligible at times $[1, 2]$ and job $E_{3,1}$ is eligible at times $[3, 6]$. Both approaches expand the vertex v_1 accordingly by creating new vertices v_2 and v_3. However, while our approach concludes that job $E_{3,1}$ is also eligible at times $[7, 8]$, the approach of [12] concludes that there are no eligible jobs in the time interval $[7, 9]$ and job $E_{1,1}$ is scheduled at its release time $t = E_{1,1}.r^{min} = E_{1,1}.r^{max} = 10$. As mentioned, this is due to the fact that according to [12], once a job was eligible during expansion of a single vertex, it cannot become eligible again. In other words, according to [12] no vertex can have two outgoing edges with the same job label. Therefore, job $E_{1,1}$ is scheduled instead, which results in a vertex v_4 with $v_4.e = E_{1,1}.r^{min} + E_{1,1}.c^{min} = 12$ and $v_4.l = E_{1,1}.r^{min} + E_{1,1}.c^{max} = 12$.

In our approach, vertex v_4 is the result of expanding v_1 with job $E_{3,1}$ in time range $[7, 8]$ which results in $v_4.e = 7 + E_{3,1}.c^{min} = 9$ and $v_4.l = 8 + E_{3,1}.c^{max} = 10$.

This discrepancy then results in vertices v_8 and v_{11} having different time intervals $[v_8.e, v_8.l]$ and $[v_{11}.e, v_{11}.l]$ which causes a deadline miss only for the approach of [12].

A.3 Conclusion

If the online scheduler were to execute the job $E_{2,1}$ at time $t = 0$ and then finished execution in the time interval $[7, 8]$, then according to the P-RM policy the only released job which does not cause a deadline miss for the highest priority job $E_{1,1}$ is job $E_{3,1}$ and therefore should be scheduled next. This corresponds to our approach, where the job $E_{3,1}$ is eligible in v_1 in the time interval $[7, 8]$.

Appendix B

Contents of the enclosed CD

The enclosed CD contains the source code of the algorithms described in this thesis. For installation and usage instructions please refer to the README.md file located in a folder called ettt-scheduling. The application is also available on GitHub (`https://github.com/redakez/ettt-scheduling`) where the source code is also available. The enclosed CD also contains the LaTeX source code of this document as well as the resulting PDF file.