Delft University of Technology
Master of Science Thesis in Computer Science

Partial-Order Reduction in Reachability-based Response-Time Analyses

Sayra Sabah Ranjha

Supervisors:
Dr. Mitra Nasri
Dr. Geoffrey Nelissen

TU Delft Delft University of Technology

Partial-Order Reduction in Reachability-based Response-Time Analyses

Master of Science Thesis in Computer Science

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Sayra Sabah Ranjha

Supervisors:
Dr. Mitra Nasri
Dr. Geoffrey Nelissen

7th of July 2021

Author
Sayra Sabah Ranjha
Title
Partial-Order Reduction in Reachability-based Response-Time Analyses
MSc Presentation Date

13th of July 2021

Graduation Committee

Prof. dr. Koen Langendoen (chairman)	Delft University of Technology
Dr. Mitra Nasri (supervisor)	Eindhoven University of Technology
Dr. Geoffrey Nelissen (supervisor)	Eindhoven University of Technology
Dr. Burcu Kulahcioglu Ozkan	Delft University of Technology

The work presented in this thesis has lead to a paper which is being submitted to the Real-Time Systems Journal of Springer.

Abstract

The temporal correctness of safety-critical systems is typically guaranteed via a response-time analysis, whose goal is to determine the *worst-case response time* (WCRT) of a set of input jobs when they are scheduled by a given scheduling policy on a computing resource. However, response-time analysis is a hard problem to solve, with most variations of the problem being NP-hard. Recently, Nasri et al. [18] have introduced an exact reachability-based response-time analysis that is based on exploring the space of possible decisions that a scheduler can take for a set of jobs/tasks. Their solution is at least three orders of magnitude faster than other exact response-time analyses and scales well.

Despite its current success in scalability, the schedule-abstraction-based analysis still faces one big fundamental limitation: *in the reachability graph that it builds, each edge can only include one single scheduling decision.* As a result, as soon as there are large uncertainties in the release time or execution time of the jobs in the input job set, the number of states generated by the schedule-abstraction graph grows exponentially because the analysis will try to explore all (valid) combinations of ordering between jobs.

We improve the scalability of the schedule-abstraction-based analysis by introducing *partial-order reduction* (POR) rules that allow combining multiple scheduling decisions on one edge and hence avoiding combinatorial exploration of all possible orderings between jobs in cases where there are large uncertainties. Our solution is an *exact schedulability analysis* and provides a *safe response-time analysis* that reduces the size of the graph while only introducing a small overestimation on the WCRT of the jobs.

Our key idea is to identify subsets of jobs for which the combinatorial exploration of all orderings is *irrelevant* to the schedulability of the job set. Exploring these combinations is irrelevant when all scenarios lead to a system state without encountering a deadline miss in any of those scenarios. Dispatching such jobs can be considered in a single step (that combines all those scheduling decisions), which further defers the state-space explosion.

We show that our solution is able to reduce the runtime of the analysis by five orders of magnitude on average, and the number of explored states by 98% on average in comparison to the original schedule-abstraction-based analysis of Nasri et al. [18] for randomly generated periodic task sets. This achievement comes with a negligible cost of an average 0.1% increase in the WCRT of the jobs. This shows that POR allows us to analyze even more task sets than the original and has the potential to scale even further.

Preface

Working on this thesis has been an amazing journey. I would like to thank Mitra Nasri and Geoffrey Nelissen for their incredible supervision, for poking holes through my theories whenever I had not done so myself, and above all for being such a joy to work with. I feel very proud of what we have achieved in this thesis, and I cannot wait to carry on with this research in the future.

Sayra Sabah Ranjha

Delft, The Netherlands
7th July 2021

Contents

Chapter 1

Introduction

Safety-critical real-time systems can be found in many industries, including avionics, railway, and automotive industries. The safety of these systems depends on both functional and temporal correctness. Guaranteeing temporal correctness is typically done via a response-time analysis, whose goal is to determine the *worst-case response time* (WCRT) of a set of input jobs when they are scheduled by a given scheduling policy on a computing resource. Typically, a real-time system is said to be temporally correct or *schedulable* if the WCRT of each job is smaller than its deadline. Namely, for a given scheduling policy, no execution scenario (under the set of valid scenarios that respect the assumptions of the system) must exist such that any of the jobs violates its timing constraints.

1.1 Related work and hardness results

The problem of response-time analysis is a hard problem to solve. Eisenbrand et al. [10] show that computing response times for periodic tasks scheduled by a fixed-priority (FP) scheduling policy[1] (which is a policy where all instances of a task are assigned the same priority) on a single-core platform is an NP-hard problem, and there is no polynomial-time algorithm for approximating response-times within any constant factor unless P = NP. They also establish that the response-time analysis problem for the earliest deadline first (EDF) policy (a policy where the task instance with the earliest deadline has the highest priority) is coNP-hard [9].

Ekberg [11] shows that despite there being a polynomial-time solution for the rate-monotonic scheduling policy (which is a FP policy where the priorities of the tasks are monotonic to their periods) when the utilization (i.e., the workload generated by the task set per unit of time) is bounded by $ln(2)$ (Liu and Layland's bound [17]), the FP-schedulability problem is NP-complete when the utilization is bounded by $c > ln(2)$. The author also proves that for arbitrary priorities, this problem is already NP-complete for utilizations bounded by $c > 0$. Table 1.1 contains a complete overview of the complexity of the FP-schedulability problem as provided by [11]. It shows that all variations of the problem are either NP-complete or NP-hard. In this table, p and d refer to the

[1] A scheduling policy determines in which order tasks are dispatched on a processor.

	Implicit deadlines ($d = p$)	**Constrained deadlines** ($d \leq p$)	**Arbitrary deadlines** (d, p unrelated)
Arbitrary utilization	Weakly NP-complete Pseudo-polynomial time algorithm exists	Weakly NP-complete Pseudo-polynomial time algorithm exists	Weakly NP-hard Exponential time algorithm exists *(Open)*
Utilization bounded by a constant $c < 1$	Weakly NP-complete for (i) $c > 0$ and arbitrary priorities, or (ii) $c > ln(2)$ and RM; In P for $c \leq ln(2)$ and RM	Weakly NP-complete for $c > 0$ Pseudo-polynomial time algorithm exists	Weakly NP-hard for $c > 0$ Pseudo-polynomial time algorithm exists *(Open)*

Table 1.1: **Overview of the complexity of the FP-schedulability problem taken from [11]. d and p refer to the relative deadline and period of the tasks.**

tasks' period and relative deadline, respectively. For example, $p = d$ refers to periodic tasks whose relative deadlines are equal to their periods.

Despite the hardness results, there have been several solutions to the problem of schedulability and response-time analysis. For example, *utilization-based schedulability tests* (i.e., schedulability tests that are based on a mathematical relation between the utilization of individual tasks) are polynomial-time but they are only *sufficient* (i.e., a task set that passes the test is certainly schedulable, but one that fails the test is not necessarily unschedulable) and thus pessimistic [4,17,19].

Known *exact* analyses fall into two general categories: (i) fixed-point iteration-based analyses [3,8,15] that have pseudo-polynomial time complexity and (ii) *reachability-based analysis* [14,18,20–22,24,25]. Fixed-point iteration-based analyses are typically faster than reachability-based analyses, however, the exactness of those analyses is limited to special cases, e.g., preemptive periodic or sporadic (event-triggered) tasks scheduled by fixed-priority scheduling [3] or earliest-deadline first (EDF) [5], or non-preemptive sporadic tasks scheduled by fixed-priority [8] or EDF policy [15].

Reachability-based analyses, on the other hand, are often exact in more general cases, e.g., preemptive [14] or non-preemptive [25] periodic and sporadic tasks scheduled by global fixed-priority scheduling policies (these are FP policies for multiprocessor systems where tasks can migrate between processors). However, the problem with reachability-based analyses is their poor scalability, as they generally support limited period values and tasks [14,24,25].

1.2 Problem definition and research questions

Recently, Nasri et al. [18] have introduced a reachability-based response-time analysis that is based on exploring the space of possible decisions that a scheduler can take for a set of jobs. They searched this space by building a graph called the *schedule-abstraction graph* (SAG). To defer the state-space explosion, Nasri et al. [18] have introduced two main techniques to reduce the state space: (i) powerful interval-based abstractions to aggregate similar system states (schedules), and (ii) state-merging rules to combine system states whose future is the same or can be explored together. These techniques allowed their solution to be at least 3000 times (three orders of magnitude) faster than other exact response-time analyses that are based on generic formal verification tools such

as UPPAAL [25] and to scale to very large system sizes (e.g., to 32 cores and 30 parallel tasks in [21]).

Limitation of current schedule-abstraction-based analyses. Despite the current success in scalability, the schedule-abstraction-based analysis still faces one big fundamental limitation: *each edge can only include one single scheduling decision* (i.e., dispatching of one job) [18,20–22]. As a result, as soon as there are large uncertainties in the release time or execution time of the jobs in the input job set, the number of states generated by the schedule-abstraction graph grows exponentially because the analysis will try to explore all (valid) combinations of ordering between jobs (more detailed explanation and examples are provided in Section 5.2).

Our goal. The goal of our work is to improve the scalability of reachability-based response-time analyses using *partial-order reduction* (POR). POR is a technique frequently used to defer state-space explosion in the domain of concurrent software system verification [1,7,13]. In the context of concurrent system verification, POR relies on the fact that the order of concurrent operations often does not matter because all orders lead to the same system state, which means that all but one order can be discarded from the state-space.

As a concrete problem, in this work we will focus on the response-time analysis problem for a set of non-preemptive periodic tasks (and arbitrary job sets) with constrained deadline ($d \leq p$) scheduled by a *job-level fixed-priority* (JLFP) scheduling policy (which is a more general class of scheduling policies and includes both the FP and EDF policies) on a single-core platform. As shown by Table 1.1, this problem is NP-hard. A recent survey of Akesson et al. [2] on industrial real-time systems shows that more than 80% of real-time systems have periodic (time-triggered) activities and about 40% of them execute the tasks on single-core platforms. Their findings shows that this problem is not only relevant now but also in the next ten years as as indicated by over 30% of industrial practitioners that filled-in the survey.

Challenges and research questions. The existing POR approaches cannot be directly applied to schedulability analyses in real-time systems because the rules for these approaches are specific to the domain of concurrent systems. This poses a number of challenges that we need to address to create a POR technique that is applicable to schedule-abstraction-based analyses.

First of all, the concurrent system specific POR rules assume that the order of independent operations (operations that do not interfere with each other, e.g., read operations) always result in the same system state, whereas this is not true for schedulability analyses because different job execution orders do not necessarily lead to the same system state. Hence, we need to define under what conditions it is still *safe* to ignore the ordering between jobs and aggregate the possible resulting states into one. This leads us to our first research question:

RQ 1. When does the execution order of jobs not contribute to a violation of timing properties (such as a deadline miss)?

In the context of concurrent system verification, POR simply discards redundant operation orders [1,7,13], while we cannot simply do this in a schedulability analysis because *different orders may lead to system states that are equivalent in terms of executed jobs and schedulability, but not in terms of response-time bounds*. Hence, the second challenge is to find a way to incorporate these orders

without explicitly exploring them because discarding this information entirely will lead to inexact analyses. This motivates our second research question:

RQ 2. How to maintain the exactness of schedule-abstraction-based analyses by including the information of the possible job execution orders *without exploring each order explicitly*?

Given the answer to the above research questions, we have a set of POR rules that allows us to (i) ignore the ordering between jobs and aggregate the possible resulting states into one, (ii) without explicitly exploring every possible ordering, (iii) while maintaining an exact analysis. However, we still need to determine how a schedule-abstraction-based analysis should manage a POR, which includes (i) in what phase of the analysis the POR is performed, (ii) how to represent a POR state, and (iii) what to do when it is *not* safe to ignore the ordering between jobs. This prompts our third research question:

RQ 3. How to incorporate our proposed partial-order reduction rules in existing schedule-abstraction-based analyses?

Finally, we want to find out whether applying POR on reachability-based response-time analyses can improve their scalability similar to the rate POR improves the scalability of concurrent system verification. As the schedule-abstraction-based analysis by Nasri et al. [18,20,21] is currently the most scalable exact response-time analysis for various scheduling problems, we aim to extend it with POR to determine to what degree the analysis's scalability can be improved even further.

1.3 Contributions

To address these research questions, we extend the schedule-abstraction-based analysis by introducing partial-order reduction (POR) rules that allow combining multiple scheduling decisions on one edge and hence avoiding combinatorial exploration of all possible orderings between jobs in cases where there are large uncertainties without jeopardizing the soundness of the analysis and without making it more pessimistic. This further reduces the size of the graph, while maintaining the exactness of the original analysis in terms of schedulability[2] and only introducing a small overestimation on the WCRT.

Our key idea is to identify subsets of jobs for which the combinatorial exploration of all orderings is *irrelevant* to the schedulability of the job set. Exploring these combinations is irrelevant when all scenarios lead to a system state without encountering a deadline miss in any of those scenarios. Dispatching such jobs can be considered in a single step (that combines all those scheduling decisions), which further defers the state-space explosion.

We perform a thorough empirical evaluation comparing the original schedule-abstraction-based analysis with our POR extension in terms of runtime, state-space, and scalability under many different system configurations. An example illustrating the difference between the old and new analysis is shown in Figure 5.1 and will be elaborated upon in Section 5.2.

[2] A job set is said to be *schedulable* if no job misses its deadline under any schedule that the underlying scheduling policy can generate for those jobs at runtime.

For randomly generated synthetic task sets, our solution reduces the runtime of the analysis by *five orders of magnitude* on average, and the *number of explored states by 98%* on average in comparison to the original schedule-abstraction-based analysis of Nasri et al. [18]. This achievement comes with a negligible cost of an average 0.1% increase in the WCRT of the jobs[3]. Furthermore, our solution was able to scale to 70 tasks and $6 \cdot 10^6$ jobs per hyperperiod (which is the least-common multiple of the period of the tasks in the task set), while the original analysis reached the timeout of four hours well before $5 \cdot 10^4$ jobs per hyperperiod. The results of the empirical evaluation will be discussed in more detail in Section 7.

1.4 Organization

Chapter 2 introduces the background knowledge and definition required to understand the thesis *for the readers who might not be familiar with the terminologies and concepts of real-time systems.* **Chapter 3** sets up our system model, workload model, notations, and assumptions used in this work. **Chapter 4** provides an overview of the state-of-the-art of exact schedulability analyses and partial-order reduction. In **Chapter 5**, we first provide details of the original schedule-abstraction-based response-time analysis of Nasri et al. [18] which is the basis of our work. Then we motivate the goal of the thesis by discussing the limitations of the original schedule-abstraction-based analysis [18], and formally define the problem that we aim to solve in the thesis. **Chapter 6** presents the partial-order reduction technique used to extend the schedule-abstraction-based analysis. We empirically evaluate our new partial-order reduction-based analysis in **Chapter 7** and compare it with the original analysis. Finally, we conclude the thesis in **Chapter 8** and propose suggestions for future work.

[3] A job is an instance of a task. A periodic task generates jobs periodically.

Chapter 2

Prerequisites and terminology

The aim of this chapter is to provide required background knowledge on real-time systems for the readers who might not be familiar with the terminology and the concepts of real-time systems. We have taken most of the definitions from the book of Giorgio Buttazzo on real-time systems [6]. Readers who are familiar with these concepts may continue from Chapter 3.

First, in Section 2.1, we describe some properties by which tasks can be characterized, such as the activation model, execution model, and type of deadline. Section 2.2 explains how scheduling algorithms can be categorized in different ways, e.g., the time the scheduler is activated (Section 2.2.1), whether it is work-conserving or not (Section 2.2.2), and the method used to assign priorities to tasks (Section 2.2.3). We discuss a number of well-known scheduling policies in Section 2.2.4. We conclude the chapter with a description of different categories of schedulability tests (Section 2.3).

2.1 Task properties

A task is a computation executed by a processor. Tasks can be characterized by their activation model (Section 2.1.1), execution model (Section 2.1.2), and deadline (Section 2.1.3). We will also explain in more detail how periodic tasks and their jobs can be modeled (Section 2.1.4).

2.1.1 Task activation models

Tasks can be distinguished by their activation model, which describes the arrival pattern of tasks, i.e., when instances of a task arrive in the system. The different release patterns are periodic, sporadic, and aperiodic. Figure 2.1 illustrates the differences between each release pattern.

Periodic. The jobs of a *periodic* task arrive at a certain frequency. The time between two consecutive arrivals is the period of a task. If a task has period p, then the time between consecutive job arrivals is exactly p. Task 1 in Figure 2.1 shows a periodic task with period 10. Note that in this example, Task 1 does not have a release jitter.

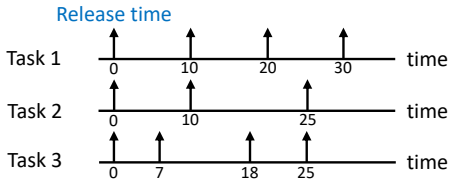


Figure 2.1: **An example showing different task activation models. Task 1 is a periodic task with period 10, task 2 is a sporadic task (with a minimum inter-arrival time of 10), and task 3 is an aperiodic task.**

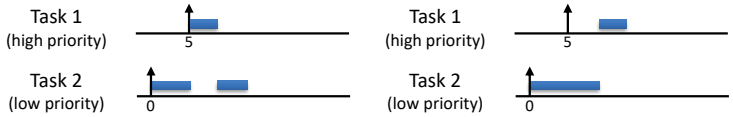


Figure 2.2: **An example showing different execution models. The left schedule shows a preemptive task set, and the right schedule shows a non-preemptive task set.**

Sporadic. The jobs of a *sporadic* task have a *minimum inter-arrival time*, which is also referred to as *period*. In contrast to periodic tasks, the time between consecutive job arrivals for a sporadic task with period p is *at least* p instead of exactly p. Task 2 in Figure 2.1 is a sporadic task with a minimum inter-arrival time of 10.

Aperiodic. The jobs of an *aperiodic* task have random and unbounded arrival times. Task 3 in Figure 2.1 is an aperiodic task.

2.1.2 Execution models

Tasks can be characterized by their execution model, which can either be preemptive or non-preemptive and describes whether the execution of a task can be interrupted/preempted.

Preemptive execution. Preemptive execution models allow the execution of a job to be preempted by the scheduler (or the operating system). This happens typically when there is a higher-priority job that has been released (added to the ready queue). If a job is preempted, it is placed back in the ready queue until the next time that the scheduler *resume*s the task. The left schedule in Figure 2.2 shows a set of preemptive tasks. Observe that Task 2 is preempted by a higher-priority task, i.e., Task 1, and resumes its execution as soon as Task 1 is finished.

Non-preemptive execution. Under non-preemptive execution, when a job is *dispatched* (starts its execution, or equivalently, when the processor is allocated to the job), it is not forced to release the processor (e.g., by the scheduler or by the operating system). Hence, it may not yield the processor until it completes its execution. Consequently, a running non-preemptive job cannot be preempted by a higher-priority job. The right schedule in Figure 2.2 shows a set of non-preemptive tasks. Observe that Task 2 keeps running until completion even though a higher-priority task is released during its execution.

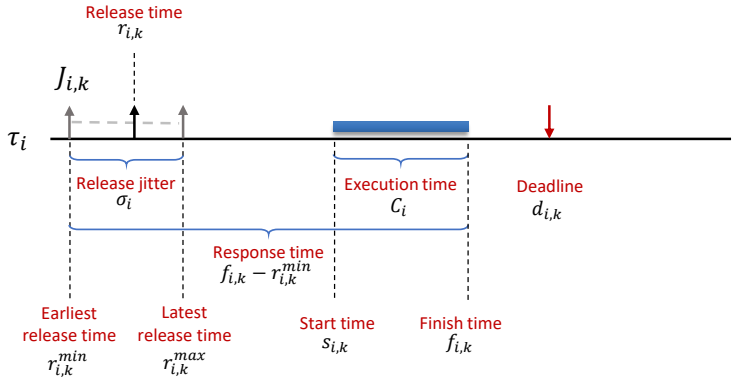


Figure 2.3: **An example showing the properties of the k^{th} job $J_{i,k}$ of a periodic task τ_i.**

2.1.3 Type of deadline

Tasks can be characterized by the type of deadline they have. The deadline type of a task indicates the consequences of missing a deadline for that task.

Hard deadline. If a task with a hard deadline suffers a deadline miss, it may have a catastrophic effect on the system.

Soft deadline. If a task with a soft deadline suffers a deadline miss, the system will still function correctly and the output of the task still has some value, but it reduces the quality of service.

Firm deadline. A firm deadline is similar to a soft deadline, but the output of a firm task missing its deadline is of no use anymore, hence, the task can be discarded.

2.1.4 Modeling periodic tasks

A periodic task is modeled by a 4-tuple $\tau_i = (T_i, D_i, [C_i^{min}, C_i^{max}], \sigma_i)$, where T_i denotes the period, D_i the relative deadline, C_i^{min} the best-case execution time (BCET), C_i^{max} the worst-case execution time (WCET), and σ_i the maximum release jitter of τ_i. We refer to the uncertainty of execution time as execution time variation. Execution time variation can be caused by e.g., processor caches, input dependencies, out-of-order-execution, or program path diversity.

A periodic task τ_i releases an infinite set of instances, called *jobs*, during its life time. The k^{th} job $J_{i,k}$ of a periodic task τ_i is released at $r_{i,k} \in [r_{i,k}^{min}, r_{i,k}^{max}]$, where $r_{i,k}^{min} = (k-1)T_i$ and $r_{i,k}^{max} = (k-1)T_i + \sigma_i$. We refer to $r_{i,k}^{min}$ and $r_{i,k}^{max}$ as the *earliest release time* and *latest release time* of $J_{i,k}$, respectively. This uncertainty of the release time is referred to as release jitter. Some causes of release jitter are interrupt latency and timer inaccuracy. The different properties of $J_{i,k}$ discussed in this section are shown in Figure 2.3. Section 3.3 describes how to create a job set from a set of periodic tasks.

Deadline. The absolute deadline $d_{i,k}$ of the $J_{i,k}$ is relative to its earliest release time $r_{i,k}^{min}$, i.e., $d_{i,k} = r_{i,k}^{min} + D_i$.

Start and finish time. The start time $s_{i,k}$ of $J_{i,k}$ is the time at which the job starts executing on the processor. The finish time $f_{i,k}$ is the time at which

$J_{i,k}$ completes its execution. If the finish time of $J_{i,k}$ is larger than its deadline, i.e., $f_{i,k} > d_{i,k}$, then $J_{i,k}$ is said to suffer a deadline miss.

Response time. The response time of $J_{i,k}$ is the length of the interval between its earliest release time $r_{i,k}^{min}$ and its finish time $f_{i,k}$, i.e., the response time of $J_{i,k}$ is $f_{i,k} - r_{i,k}^{min}$. The best-case response time (BCRT) of τ_i is the *minimum* response time among all jobs of τ_i, and the worst-case response time (WCRT) of τ_i is the *maximum* response time among all jobs of τ_i.

Slack. The slack of a job $J_{i,k}$ is the difference between the finish time and deadline of a job, i.e., $d_{i,k} - f_{i,k}$. Negative slack indicates a deadline miss.

Tardiness. The tardiness of a job is the difference between the finish time and deadline of the job, i.e., $\max\{0, f_{i,k} - d_{i,k}\}$. Tardiness can never be a negative value.

Hyperperiod. A set of periodic tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ has a *hyperperiod* H, which is defined as the least common multiple of the periods in the task set, i.e., $H = lcm(T_1, \ldots, T_n)$. The hyperperiod of a task set is the minimum amount of time after which the arrival pattern of the task set repeats.

Utilization. Periodic tasks have utilization. Utilization is an indication of the processor load caused by a task or a task set. The utilization U_i of task τ_i is the fraction of processor time spent to execute τ_i and is defined as $U_i = C_i^{max}/T_i$. The total utilization U of a system is the fraction of processor time spent to execute all tasks in the task set, i.e., it is the sum of the utilizations of all tasks.

2.2 Scheduling algorithms

A *scheduling algorithm* or *scheduling policy* determines in which order tasks are dispatched on a processor. The decisions made by the scheduling algorithm result in a particular assignment of tasks the processor, also referred to as a *schedule*. Scheduling algorithms can be categorized in different ways, such as the time the scheduler is activated (Section 2.2.1), whether it is work-conserving or not (Section 2.2.2), and the method used to assign priorities to tasks (Section 2.2.3). We will conclude this section by discussing a number of well-known scheduling policies (Section 2.2.4).

2.2.1 Categorization by the time the scheduler is activated

Scheduling algorithms can be categorized by the time that scheduling decisions are taken, namely online or offline.

Online scheduling. Online scheduling policies take scheduling decisions at runtime, i.e., the scheduler is called every time a task is releases or completes.

Offline scheduling. Offline scheduling policies take all scheduling decisions beforehand, so before any task has been dispatched. These scheduling decisions are then stored in a table, which is then used by the dispatcher at runtime to dispatch tasks according to the predetermined schedule.

2.2.2 Categorization by work-conservation

Scheduling algorithms can be categorized based on whether they are work-conserving or non-work-conserving.

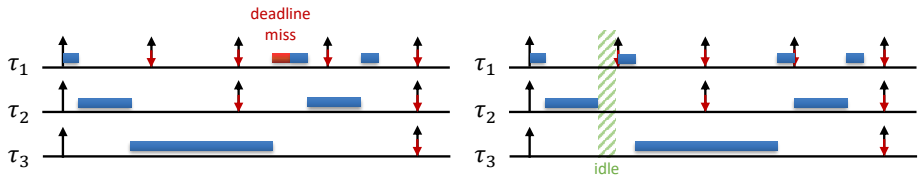


Figure 2.4: **An example showing how a non-work-conserving scheduling policy can insert an idle interval to prevent a deadline miss.**

Work-conserving. Work-conserving scheduling algorithms do not leave the processor idle as long as there is a ready job in the system. The schedule on the left in Figure 2.4 shows there is a deadline miss for the second job of τ_1 when the tasks are scheduled using a work-conserving policy.

Non-work-conserving. Non-work-conserving scheduling algorithms may schedule an idle interval, even if there is a ready job in the system. The schedule on the right in Figure 2.4 shows that an idle time is scheduled after the first job of τ_2 even though the first job of τ_3 is ready. As a result, there is no deadline miss for any of the jobs when they are scheduled using a non-work-conserving policy.

2.2.3 Categorization by priority assignment

Scheduling algorithms can be categorized to the following three categories based on how they prioritize jobs over each other: (i) task-level fixed-priority, (ii) job-level fixed-priority, and (iii) job-level dynamic priority algorithms. The rest of this section introduces these policies.

Task-level fixed-priority (FP). Tasks are assigned priorities and each job of a task has the same priority. An example of a task-level fixed priority assignment is rate-monotonic (RM), which assigns priorities to tasks based on their period, where a smaller period corresponds to a higher priority.

Job-level fixed-priority (JLFP). A JLFP scheduler is a scheduler where jobs are assigned priorities based on its parameters at its release time. This means that jobs of the same task can have different priorities. An example of a job-level fixed priority assignment is the earliest deadline first (EDF) policy. EDF assigns priority in order of absolute deadline, where the job with the earliest deadline has the highest priority.

Job-level dynamic-priority. Jobs are assigned priorities based on parameters that can change over time. An example is the shortest remaining execution time first policy, which assigns priorities to jobs based on their remaining execution time, where the job with the shortest execution time remaining has the highest priority.

2.2.4 Well-known scheduling policies

In this section we introduce the following well-known scheduling policies: (i) rate monotonic (RM), (ii) deadline monotonic (DM), (iii) earliest deadline first (EDF), (iv) shortest job first (SJF), and (v) first in first out (FIFO). Furthermore, we describe two different categories of scheduling policies for multiprocessor scheduling, namely global and partitioned scheduling.

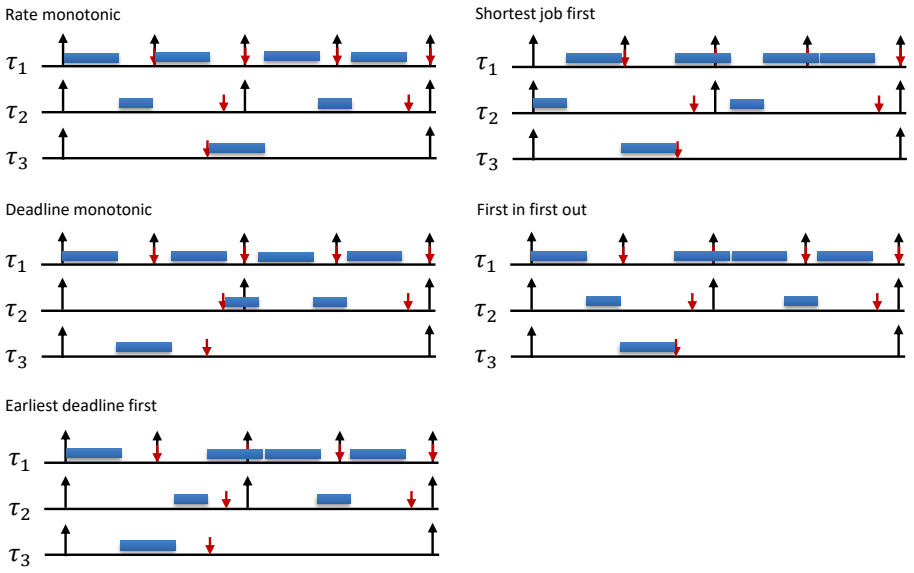


Figure 2.5: **An example showing how scheduling policies can generate different schedules for the same task set. Pictured are non-preemptive versions of rate monotonic, deadline monotonic, earliest deadline first, shortest job first, and first in first out.** $T_1 = 5$, $C_1 = 3$, **and** $D_1 = 5$. $T_2 = 10$, $C_2 = 2$, **and** $D_2 = 9$. $T_3 = 20$, $C_3 = 3$, **and** $D_3 = 8$.

Rate monotonic (RM) The rate monotonic (RM) scheduling policy assigns priorities to tasks based on their period, where a smaller period corresponds to a higher priority. Hence, RM is a fixed-priority scheduling policy. The schedule on the left side of the top row in Figure 2.5 shows how a task set is scheduled under a non-preemptive RM policy. τ_1 has the shortest period and thus the highest priority, followed by τ_2 and finally τ_3.

Deadline monotonic (DM) The deadline monotonic (DM) scheduling policy assigns priorities to tasks based on their relative deadline, where a smaller relative deadline corresponds to a higher priority. Hence, DM is a fixed-priority scheduling policy. The schedule on the left side of the middle row in Figure 2.5 shows how a task set is scheduled under a non-preemptive DM policy. In this example, τ_1 has the highest, τ_3 the medium, and τ_2 the lowest priority.

Earliest deadline first (EDF) The earliest deadline first (EDF) scheduling policy assigns priority to jobs in order of absolute deadline, where the job with the earliest deadline has the highest priority. Since jobs of the same task can have different priorities, EDF is a job-level fixed-priority scheduling policy. The schedule on the left side of the bottom row in Figure 2.5 shows how a task set is scheduled under a non-preemptive EDF policy.

Shortest job first (SJF) The shortest job first (SJF) scheduling policy assigns priorities to tasks based on their execution time, where a smaller execution time corresponds to a higher priority. As a result, SJF is a fixed-priority scheduling policy. The schedule on the right side of the middle row in Figure 2.5 shows how a task set is scheduled under a non-preemptive SJF policy. τ_2 has the smallest execution time and thus the highest priority, followed by τ_1 and τ_3.

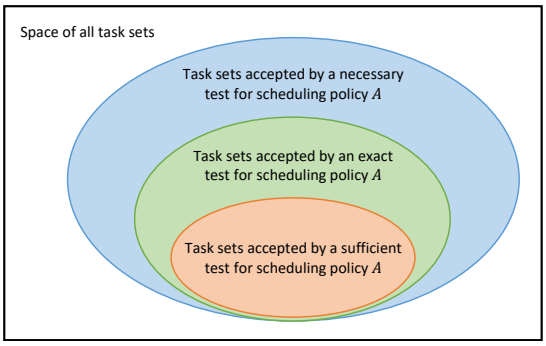


Figure 2.6: **An example showing how necessary, sufficient, and exact schedulability tests relate to each other in the space of all task sets.**

First in first out (FIFO) The first in first out (FIFO) scheduling policy schedules jobs in order of their arrival, i.e., the job that arrives first is executed first. The schedule on the right side of the middle row in Figure 2.5 shows how a task set is scheduled under a FIFO policy.

Multiprocessor scheduling policies

Online scheduling policies for multiprocessor systems can be categorized into two broad categories depending on whether the migration is allowed or not:

Global scheduling Under a global scheduling policy, tasks may migrate between processors. The system manages a single ready queue and the processor that a task runs on is determined at runtime. The concept of global scheduling can be applied to the scheduling policies mentioned previously, e.g., to create a global FP or global EDF scheduling policy.

Partitioned scheduling Under a partitioned scheduling policy, each task is assigned to a particular processor offline, and the processor a task runs on cannot be changed at runtime. Each processor manages its own ready queue. Hence, the partitioned scheduling problem reduces to deciding on an assignment of tasks to processors followed by uniprocessor scheduling. The concept of partitioned scheduling can be applied to the scheduling policies mentioned previously, e.g., to create a partitioned EDF scheduling policy.

2.3 Schedulability tests

A schedulability test determines whether a set of tasks is schedulable under a given scheduling policy A using a given system model. Schedulability tests can be categorized into the following categories based on what the output tells us about the schedulability of a job set: (i) necessary, (ii) sufficient, and (iii) exact. Figure 2.6 shows how these categories relate to each other.

Necessary test A schedulability test is a *necessary* test for scheduling policy A if all task sets that fail the test (i.e., are deemed unschedulable by the test) are certainly unschedulable by A.

Sufficient test A schedulability test is a *sufficient* test for scheduling policy A if all task sets that pass the test (i.e., are deemed schedulable by the test) are certainly schedulable by A.

Exact test A schedulability test is an *exact* test for scheduling policy A if it is both a necessary and sufficient test for A. Namely, all task sets that fail the test are certainly unschedulable, and all task sets that pass the test are certainly schedulable by A.

Chapter 3

System model and assumptions

3.1 Job and system model

We consider the problem of scheduling a finite set of non-preemptive jobs $\mathcal{J}$ on a uniprocessor platform. A job $J_i = ([r_i^{min}, r_i^{max}], [C_i^{min}, C_i^{max}], d_i, p_i)$ is characterized by its earliest release time r_i^{min}, latest release time r_i^{max}, best-case execution time (BCET) C_i^{min}, worst-case execution time (WCET) C_i^{max}, absolute deadline d_i, and priority p_i. We assume a discrete time model, meaning the job timing parameters are integer multiples of the system clock.

Job J_i non-deterministically releases at a time instant $r_i \in [r_i^{min}, r_i^{max}]$ and executes for an *a priori* unknown amount of time $C_i \in [C_i^{min}, C_i^{max}]$. Because of this uncertainty, we say that job J_i is *possibly released* at time t if $r_i^{min} \leq t < r_i^{max}$ and *certainly released* at time t if $t \geq r_i^{max}$. Furthermore, we say a job is *ready* at time t if it is released but did not start executing before t.

Because we assume that every job J_i is non-preemptive, a job J_i that starts executing *at* time t finishes its execution *by* time $t + C_i$, continuously occupying the processor during the interval $[t, t + C_i)$. We denote the finish time of J_i by f_i. Once J_i finishes its execution by f_i, the processor becomes available again and the next job may start. The response time of J_i is the length of the interval between its earliest release time r_i^{min} and its finish time f_i, i.e, the response time of J_i is $f_i - r_i^{min}$.

As the deadline of a job is absolute, we assume that it is not affected by the uncertainty on the release time (referred to as release jitter in the rest of this work). We assume that if $p_i < p_j$, job J_i has a higher priority than job J_j, i.e., a lower value of p_i indicates a higher priority. Additionally, priority ties are broken by task ID and then job ID, so we assume that the "$<$" operator implicitly uses this tie-breaking rule.

We use $\langle\ \rangle$ to refer to an ordered set (or a sequence) and $\{\ \}$ to refer to a non-ordered set. Neither of the two contains repeated items. We use $\min_\infty\{X\}$ and $\max_0\{X\}$ over a set of positive integers X to indicate that the minimum of an empty set is equal to ∞ and the maximum of an empty set is equal to 0.

3.2 Scheduler model

We consider all non-preemptive job-level fixed-priority (JLFP) scheduling algorithms. Despite the original schedule-abstraction graph from Nasri et al. [18] supporting both work-conserving and non-work-conserving scheduling policies, we only focus on work-conserving policies in this thesis, i.e., policies that do not leave the processor idle as long as there is a ready job in the system. Like the original analysis, we solely focus on schedulers that are priority-driven and deterministic, i.e., schedulers that only schedule a job if it is the highest-priority ready job in the system and always produce the same schedule for a given execution scenario, where an execution scenario is defined as follows (according to [18,20]).

Definition 1. (from [18]) An execution scenario $\gamma = (C, R)$ for a set of jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_m\}$ is a sequence of execution times $C = \langle C_1, C_2, \ldots, C_m \rangle$ and release times $R = \langle r_1, r_2, \ldots, r_m \rangle$ such that, $\forall J_i \in \mathcal{J}$, $C_i \in [C_i^{min}, C_i^{max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$

We consider a set of jobs $\mathcal{J}$ to be *schedulable* under a given scheduling policy A if there exists no execution scenario of $\mathcal{J}$ that results in a deadline miss when scheduled by A.

3.3 Creating job sets from periodic task sets

Forming the job set. A given periodic task set $\tau = \{\tau_1, \ldots, \tau_n\}$, where each task is introduced as a 4-tuple $\tau_i = (T_i, D_i, [C_i^{min}, C_i^{max}], \sigma_i)$, can be converted to the job set described in Section 3.1 as follows. First, compute the hyperperiod H of τ, where $H = lcm(T_1, \ldots, T_n)$. Then, for each task τ_i, we compute the number of jobs that that task releases within one hyperperiod, i.e., $n_i = \lceil \frac{H}{T_i} \rceil$. Finally, we generate a job set constructed from all jobs of all tasks in one hyperperiod:

$$\mathcal{J} = \{J_{i,k} \mid \forall i, 1 \leq i \leq n, \forall k, 1 \leq k \leq n_i\}. \tag{3.1}$$

Assigning parameters of a job. The k^{th} job of τ_i is characterized as follows: $J_{i,k} = ([r_{i,k}^{min}, r_{i,k}^{max}], [C_i^{min}, C_i^{max}], d_{i,k}, p_{i,k})$, where $r_{i,k}^{min} = (k-1)T_i$, $r_{i,k}^{max} = (k-1)T_i + \sigma_i$ are the release interval of the job, and $d_{i,k} = r_{i,k}^{min} + D_i$ is the absolute deadline of the job.

Assigning priority of a job. The priority $p_{i,k}$ of a job $J_{i,k}$ is determined by the scheduling policy. For example, rate-monotonic (RM) priorities can be obtained by ordering all tasks by their period in ascending order, and assigning each task a priority corresponding to their position in that order. Then each job of a task will inherit priority of the task (recall that RM is a task-level fixed-priority scheduling policy).

For the EDF policy, the priority of each job is equal to the value of the absolute deadline of that job; the earlier (equivalently, the smaller) the absolute deadline, the higher the priority. Recall that EDF is a job-level fixed-priority policy (but a task-level dynamic priority policy).

Chapter 4

Related work

The purpose of this chapter is to provide an overview of the state-of-the-art of exact schedulability analyses and partial-order reduction. Section 4.1 introduces different types of exact schedulability analyses. We distinguish between fixed-point iteration-based analyses (Section 4.1.1) and reachability-based analysis (Section 4.1.2), and describe how they work and what their limitations are. Section 4.2 introduces the concept of partial-order reduction and presents different variations of partial-order reduction techniques, namely static (Section 4.2.1) and dynamic (Section 4.2.2) partial-order reduction. Finally, Section 4.2.3 explains why existing partial-order reduction approaches are not directly applicable to schedulability analyses.

4.1 Exact schedulability analyses

Despite the hardness results introduced in Section 1.1, there have been several exact solutions to the problem of schedulability and response-time analysis.

4.1.1 Fixed-point iteration-based analyses

Davis et al. [8] propose an exact schedulability analysis that provides exact WCRT for non-preemptive *sporadic* tasks scheduled by FP. The analysis computes the WCRT of tasks by solving recurrence relations by means of fixed-point iteration. In each step, the amount of the new interference that the task might get from higher-priority tasks is added to the current estimation of the WCRT until no new interference can be included (reaching a convergence in the fixed-point iteration).

Although this analysis is relatively fast (for an analysis solving an NP-hard problem), and is exact for *sporadic tasks*, it is pessimistic when applied on periodic tasks as shown by Nasri et al. [18], since the analysis assumes a worst-case release pattern for the tasks which may never happen in a periodic system. Namely, it includes more (and worst) scenarios than those that can happen in a periodic task set. As a result, when applied on periodic tasks, it is only a *sufficient* analysis.

4.1.2 Reachability-based analysis

Reachability-based analyses determine the schedulability of task sets by exploring system states and checking whether there is a reachable state where some task suffers a deadline miss.

Linear hybrid automata-based analysis

Sun et al. [24] provide an exact schedulability analysis for preemptive sporadic tasks scheduled by global fixed-priority scheduling on multiprocessor systems. The analysis uses linear hybrid automata to model the system, and then checks for the reachability of a state where some task suffers a deadline miss.

Though exact, this analysis is not scalable. The evaluation results show that it can handle at most only seven tasks and four cores. Furthermore, it is only shown to be exact for sporadic tasks and not for periodic tasks. Additionally, the analysis by Sun et al. [24] considers preemptive tasks, while our work considers non-preemptive tasks.

Timed automata-based analysis

An exact schedulability analysis based on timed automata (TA) is introduced by Guan et al. [14]. This analysis considers preemptive periodic and sporadic tasks scheduled under global fixed-priority scheduling on multiprocessor systems. Timed automata are used to model the system, the scheduler, and the tasks. Then the authors have used the UPPAAL model checker to check if a state containing a deadline miss is reachable in the model. Similar to the analysis by Sun et al. [24], the TA-based analysis by Guan et al. [14] has limited scalability because it is only able to handle integer task periods between 8 and 20. Furthermore, this analysis also only considers preemptive tasks.

Yalcinkaya et al. [25] introduce an exact TA-based schedulability test for non-preemptive self-suspending periodic and sporadic tasks. Like the analysis by Guan et al. [14], the system is modeled by timed automata and UPPAAL is utilized as a model checker. With the ability to scale up to 60 tasks on 2 cores, 30 tasks on 4 cores, and 15 tasks on 8 cores, it scales better than the TA-based analysis by Guan et al. [14]. However, it still suffers from scalability issues as the runtime increases rapidly with the utilization, number of tasks, and number of cores as it has been shown in [21].

Schedule-abstraction-based analysis

Recently, Nasri et al. [18] have introduced a reachability-based response-time analysis that is based on exploring the space of possible decisions that a scheduler can take for a set of jobs. They searched this space by building a graph called the *schedule-abstraction graph* (SAG). Each vertex in the graph represents the state of the platform/resource after the execution of a set of jobs. An edge between two vertices v_1 and v_2 represents a scheduling decision. Since the SAG has been designed for non-preemptive jobs [18], the scheduling decision is to determine "a next job that can possibly be dispatched" on the platform/resource after the state v_1. Dispatching of this job will change the platform/resource state from v_1 to v_2. While building the graph, the response time of each job that is being dispatched on an edge is tracked and hence when the graph is fully

constructed, the method outputs the smallest and largest response time of each job in all scenarios (edges) that involve that job.

To defer the state-space explosion, Nasri et al. [18] have introduced two main techniques to reduce the state space: (i) powerful interval-based abstractions to aggregate similar system states (schedules), and (ii) state-merging rules to combine system states whose future is the same or can be explored together (details of these steps will be explained in Section 5.1). These techniques allowed their solution to be at least 3000 times (i.e., three orders of magnitude) faster than other exact response-time analyses that are based on generic formal verification tools such as UPPAAL [25] and to scale to very large system sizes (e.g., to 32 cores and 30 parallel tasks in [21]).

However, despite their current success in scalability, the schedule-abstraction based analysis still faces one big fundamental limitation: *each edge can only include one single scheduling decision* (i.e., dispatching of one job) [18,20–22].

As a result, as soon as there are large uncertainties in the release time or execution time of the jobs in the input job set, the number of states generated by the schedule-abstraction graph grows exponentially because the analysis will try to explore all (valid) combinations of ordering between jobs. As we will show in our work, such a combinatorial exploration can be avoided in many cases by introducing *partial-order reduction* (POR) rules that combine a set of jobs on one single edge without jeopardizing the soundness of the analysis and without making it more pessimistic.

4.2 Partial-order reduction

Partial-order reduction is a technique frequently used in the verification of concurrent software systems by model checking. The state-space that needs to be explored to verify a concurrent system suffers from the problem of state-space explosion, as the number of states that need to be considered grows exponentially with the number of concurrent processes.

The problem is that concurrent operations on shared variables are often *independent*, i.e., they do not interfere with each other. For example, read operations on a shared variable are independent. The order in which such independent operations are performed does not matter as every order leads to the same system state. However, model checkers explore all possible combinations of such operations, leading to an explosion of the explored state-space. Partial-order reduction reduces the state-space by ignoring such redundant orders of concurrent operations.

In the rest of this section, we first introduce two categories of partial-order reduction techniques used for concurrent system verification, i.e., static (Section 4.2.1) and dynamic (Section 4.2.2) partial-order reduction. Next, we summarize why these techniques cannot directly be used in the context of the response-time analysis problem.

4.2.1 Static partial-order reduction

Clarke et al. [7] describe a method of static partial-order reduction. Static analysis is used to determine what shared memory locations can be accessed

by concurrent processes. Based on this information, the POR removes state transitions that do not add additional information to the state-space.

Static POR requires a concurrent program to perform static analysis on. Applying this concept to schedulability analyses would require a schedulability analysis that explicitly simulates the execution of jobs as a concurrent program that accesses some shared resources. Even if we would have such a schedulability analysis, the POR rules do not respect the fact that different job execution orders do not necessarily lead to the same system state. Hence, the technique of static POR is not applicable to schedule-abstraction based analyses.

4.2.2 Dynamic partial-order reduction

Flanagan et al. [13] introduce a dynamic POR that determines shared memory accesses at runtime, and expands the execution trace by backtracking until all alternative paths in the state-space are explored. This dynamic POR is extended by Abdulla et al. [1] to make an optimal solution in terms of the number of paths that are explored.

The difference between static and dynamic POR is that the former statically identifies shared memory accesses, while the latter does so by executing the program. Therefore, dynamic POR is not directly applicable to schedule-abstraction based analyses for the same reasons as static POR.

4.2.3 Applicability to schedulability analyses

The POR approaches discussed above cannot be directly applied to schedulability analyses in real-time systems because the rules for existing POR approaches are specific to the domain of concurrent systems.

To add POR to the schedule-abstraction based analyses, we need to define: (i) *when the execution order of jobs does not contribute to a violation of timing properties (such as a deadline miss)*, and (ii) how to incorporate POR into the system state and the analysis.

The addition of POR to schedule-abstraction based analyses poses a number of challenges: (i) In the context of concurrent system verification, POR simply discards redundant operation orders, while we cannot simply do this in a schedulability analysis because different job orders may lead to system states that are equivalent in terms of executed jobs and schedulability, but not in terms of response-time bounds. Discarding this information entirely will lead to inexact analyses, so we need to ensure that the POR technique we define does not entirely discard such orders. (ii) Naively solving this problem by keeping such job execution orders would limit the effect that POR can have on the state-space, because different job orders rarely result in system states with equivalent response-time bounds. Hence, we need to find a to incorporate these job execution orders instead of discarding them like the conventional POR approaches *without* explicitly exploring them.

Chapter 5

Motivation and problem definition

The goal of this work is to improve the scalability of the SAG by introducing the concept of partial-order reduction (POR) to it. This chapter provides a motivation for this goal and a definition of the problem we will solve. First, Section 5.1 describes how the SAG works and then Section 5.2 explains why the SAG is inefficient in its current form and provides an idea of how POR can solve this inefficiency. Finally, Section 5.3 formally defines the problem we aim to solve.

5.1 Schedule-abstraction graph

This section introduces the reachability-based response-time analysis of Nasri et al. [18]. The analysis searches the space of possible decisions a scheduler can take for a set of jobs $\mathcal{J}$ by building a graph called the *schedule-abstraction graph* (SAG). This SAG is a *directed acyclic graph* (DAG) denoted by $G = (V, E)$ and defined as follows:

Edges. E is the set of edges, where an edge $e_k = (v_i, v_j, J_l)$ between system states (vertices) v_i and v_j represents a scheduling decision. Since the SAG is designed for non-preemptive jobs, the scheduling decision is to determine "a next job that can possibly be dispatched" on the platform after state v_i. Hence, each edge e_k is labeled with the job J_l that is dispatched after state v_i and evolves that state to a new system state v_j.

Paths. A path P starting from vertex v_1 and ending with vertex v_i is a possible sequence of scheduling decisions to go from initial system state v_1 to v_i. We denote $\mathcal{J}^P$ as the *set of jobs* dispatched on a path P. We only consider paths that start from v_1.

Vertices. V is the set of vertices or system states, where each vertex represents the state of the platform after the execution of a set of jobs. Each $v_i \in V$ has a label $[A_1^{min}(v_i), A_1^{max}(v_i)]$ that represents the availability interval of the processor in state v_i in the form of an *uncertainty interval* (defined in [21]). Here, $A_1^{min}(v_i)$ and $A_1^{max}(v_i)$ are the *earliest* and *latest finish time* of the set of jobs that are on any path from v_1 to v_i. Equivalently, the processor is not available to dispatch a new job before $A_1^{min}(v_i)$, it can become available at any time

during the interval $[A_1^{min}(v_i), A_1^{max}(v_i)]$, and it is certainly available from time $A_1^{max}(v_i)$ onward. This interval captures the non-determinism in the release time and execution time of the jobs that have been dispatched on the platform before reaching this state.

We say that the processor becomes *possibly available* after system state v_i at time $A_1^{min}(v_i)$ and *certainly available* at time $A_1^{max}(v_i)$.

Construction of the graph. Nasri et al. [18,20–22] have constructed the schedule-abstraction graph following a breadth-first approach. Namely, after the initial state v_1 is added, all possible scheduling decisions that can be taken after v_1 are found (in a phase called *expansion phase*). For each of these decisions, one edge and a new system state is added to the state v_1. Then, one-by-one, each of the new states is expanded separately in a breadth-first manner.

To defer the state space explosion, the schedule-abstraction graph also includes a *merge phase*, where newly created states (from the expansion phase) are assessed and if certain conditions hold, they are merged together. In the following sections, we explain the expansion and merge phases with more details.

5.1.1 Expansion phase

In the expansion phase, the shortest path (or one of them in case of multiple shortest paths) P in the graph (connecting v_1 to v_p) is expanded. The expansion phase finds all jobs J_j that can be a *direct successor* of state v_p.

A job J_j is said to be a direct successor of a state v_p if there exists an execution scenario in which job J_j is dispatched *after* state v_p and *before* any other job. For all these jobs, a vertex is created and added to the graph by connecting it to v_p, the leaf vertex of P, with an edge. A job J_j can be dispatched next after path P ending with state v_p if its earliest start time after v_p, EST_j, is not later than its latest start time after v_p, LST_j, i.e., if

$$EST_j \leq LST_j \tag{5.1}$$

Definition 2. (From [21]) A job $J_j \in \mathcal{J} \setminus \mathcal{J}^P$ is a *direct successor* for path P ending in vertex v_p if and only if (5.1) holds.

A job cannot start executing before its earliest release or before the earliest time the processor becomes available. Hence, the earliest time that J_j can start executing after state v_p is

$$EST_j = \max\{A_1^{min}(v_p), r_j^{min}\} \tag{5.2}$$

The latest start time of J_j, i.e., LST_j, is influenced by two properties of the underlying scheduling policy, namely, the scheduler (i) applies a job-level fixed-priority policy when deciding the *next* job to be dispatched, and (ii) it is work-conserving. More formally, a job J_j can be a direct successor of v_p if it can start executing before a *higher-priority job is certainly released* (from (i)).

Nasri et al. [21] define the upper bound on the certain release of a higher-priority job as follows

$$t_{high} \triangleq \min_{\infty}\{r_x^{max} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P \wedge p_x < p_j\}. \tag{5.3}$$

From (5.3), we derive the last time instant that J_j can start executing before a higher-priority job is certainly released as $t_{high} - 1$.

The second upper bound on the latest start time of J_j is t_{wc} under work-conserving policies. It is the time at which the processor is certainly available and a not-yet-scheduled job is certainly released (because if these two conditions hold, then the scheduler will certainly dispatch *a job* on the platform). Nasri et al. [21] define t_{wc} as follows:

$$t_{wc} \triangleq \max\{A_1^{max}(v_p), \min_{\infty}\{r_x^{max} \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P\}\} \tag{5.4}$$

Combining the two upper bounds from (5.3) and (5.4), the latest time instant at which J_j can start such that J_j is dispatched after state v_p and before *any other job* is

$$LST_j = \min\{t_{wc}, t_{high} - 1\} \tag{5.5}$$

When expanding the shortest path P ending with vertex v_p, a new vertex v_q is created for each direct successor J_j of P by adding J_j at the end of P. This v_q represents the state of the platform after executing J_j after v_p. The *earliest finish time* (EFT_j) of J_j after v_p (or equivalently, the earliest availability time of v_q, i.e., $A_1^{min}(v_q)$) is obtained as follows [21]:

$$EFT_j = EST_j + C_j^{min} \tag{5.6}$$

The *latest finish time* (LFT_j) of J_j after v_p (or equivalently, the earliest availability time of v_q, i.e., $A_1^{max}(v_q)$) is obtained as follows [21]:

$$LFT_j = LST_j + C_j^{max} \tag{5.7}$$

The EFT_j and LFT_j provide a lower bound on the BCRT and an upper bound on the WCRT of J_j, respectively. After building the graph (i.e., after exploring all possible scheduling scenarios for J_j), the smallest lower bound will be reported as the BCRT and the largest upper bound will be reported as the WCRT of job J_j.

5.1.2 Merge phase

After creating a vertex v_q by expanding v_p, we enter the merge phase. In the merge phase, system states whose future is the same or can be explored together are merged into a single state to defer the state-space explosion. The future of system states is the same if their sets of dispatched jobs are equal and if their processor availability intervals overlap (intersects).

More formally, for the newly created state v_q with path Q, we check if there exist any states in the graph that can be merged with v_q. Nasri et al. [21] defined the criteria by which two states v_q and v_r can be safely merged without introducing pessimism in the analysis as follows:

Definition 3. (From [21]) Two system states v_q and v_r reachable from v_1 via paths Q and R, respectively, can be merged if $\mathcal{J}^Q = \mathcal{J}^R$ and $[A_1^{min}(v_q), A_1^{max}(v_q)] \cap [A_1^{min}(v_r), A_1^{max}(v_r)] \neq \emptyset$.

To merge v_q and v_r, we first update the label of v_q as follows:

$$[A_1^{min}(v_q), A_1^{max}(v_q)] \leftarrow [A_1^{min}(v_r), A_1^{max}(v_r)] \cup [A_1^{min}(v_q), A_1^{max}(v_q)] \tag{5.8}$$

Then, all edges pointing to v_r are pointed to v_q. Now, path R ends with v_q instead of v_r and v_r is removed from the graph. This means that the future of

both paths Q and R can be explored by only expanding a single state (i.e., the updated v_q).

Note that due to the condition $\mathcal{J}^Q = \mathcal{J}^R$ in Definition 3, the only states that can be merged are those that are reachable from v_1 with *the same set of jobs*. As explained earlier, Nasri et al. [18,20,21] build the graph with a breadth-first manner. Hence, to find states that could possibly be merged with a newly created state, we only need to look at the states that are reachable from v_1 with the same number of jobs and then check the conditions of Definition 3. This has a linear complexity w.r.t. the number of vertices that are on the front of the graph (the ones that are being expanded in one iteration).

The algorithm that builds the graph alternates between the expansion and merge phases until there exist no more incomplete paths in the graph, or it encounters a deadline miss (the algorithm can be adjusted to stop as soon as it finds a deadline miss, or to continue anyway until it processes all input jobs).

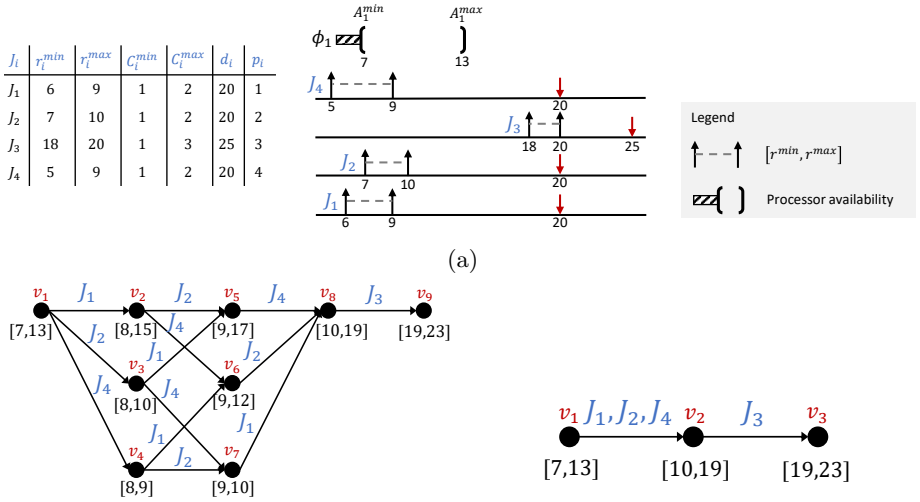
5.2 Motivation and basic idea

The current schedulability analysis using the SAG from Nasri et al. [18] does not scale well when there are large uncertainties in the release time or execution time because there are too many possible job execution orderings the analysis needs to consider, as shown in [18] and our own experiments in Section 7. Many of these job execution orderings are not relevant to explore as they analyze all the different execution orderings of the same set of jobs, of which none may ever lead to a deadline miss. If we could somehow avoid all these possible scenarios from being explored, we could get a performance gain in the analysis.

Example 1. Consider the job set in Figure 5.1a. Due to the release jitter, either J_1, J_2, or J_4 can start first (can be released first), after which any of the two remaining jobs can be dispatched. For example, if J_4 is released at time 5 and J_1 and J_2 are released at time instants 9 and 7, respectively, the first job that will be dispatched by the scheduler is J_4 and then job J_2 and finally job J_1. Alternatively, we could see a scenario in which J_1 is dispatched first (i.e., when it is released at time 6 but all other jobs are released after time 6). The SAG captures all these scenarios (and possible orderings between the jobs in the graph shown in Figure 5.1b.

As we can see, the corresponding SAG as constructed by the algorithm in [18] and shown in Figure 5.1b, has nine nodes and 13 edges. Yet, none of the possible job execution orderings of J_1, J_2, and J_4 considered in the graph may lead to a deadline miss (for any of these jobs or the future jobs, e.g., J_3). Ideally, we would want to skip over these three jobs without exploring every individual job execution ordering.

The partial-order reduction technique proposed in this work allows us to identify job orderings that are not relevant to explore, and treat them as a single scheduling decision in the SAG, i.e., depict them as a single edge. Figure 5.1c shows the schedule-abstraction graph constructed using our proposed POR technique, where J_1, J_2, and J_4 are put on a single edge, removing the need to enumerate all possible scenarios between these jobs and shrinking the resulting schedule graph. Note that the interval (i.e., label) recorded in v_8 in



(b) **Schedule-abstraction graph constructed by the original analysis from [18] using NP-FP.**

(c) **Schedule-abstraction graph constructed by partial-order reduction using NP-FP.**

Figure 5.1: **An example showing the difference between the schedule-abstraction graph constructed by the original analysis and the new partial-order reduction analysis.**

Figure 5.1b is the same as the interval recorded in v_2 in Figure 5.1c, even though the latter did not explore all different scenarios.

In this work, we extend the SAG algorithm with the idea of POR sketched above. In the expansion phase of the schedule-abstraction graph algorithm, instead of simply expanding the graph with a new vertex for each direct successor, we perform POR to see whether a set of future (i.e., not yet dispatched) jobs can be "reduced" to a single scheduling decision that encompasses all orderings of the original jobs without explicitly exploring all these orderings. If the set of jobs meets the conditions for a *safe* POR, then it is added to the graph by adding a single new edge and vertex. In the case that the conditions are not met, the graph is expanded with a new vertex for each direct successor as in the original SAG algorithm.

5.3 Problem definition

5.3.1 Ensuring an exact schedulability analysis

An important property of the original schedulability analysis by Nasri et al. [18] is that it is both an exact schedulability and response-time analysis. The analysis is an *exact schedulability analysis*, meaning that any job set deemed schedulable (resp. unschedulable) by the analysis is indeed schedulable (resp. unschedulable).

Definition 4. An analysis is an *exact schedulability analysis* if and only if there is no execution scenario (see Definition 1) in a job set that is deemed schedulable by the analysis that may result in a deadline miss. If a job set is

deemed unschedulable, then there is at least one execution scenario in that job set that results in a deadline miss.

Definition 5. An analysis is an *exact response-time analysis* if and only if (i) there is no execution scenario in a job set such that any job has a response-time smaller than the BCRT or larger than the WCRT returned by the analysis for that job, and (ii) there must be an execution scenario in the job set that results in the reported BCRT and WCRT for that job.

However, since the goal of POR is to eliminate the need to explore excessive job execution orderings, we need to make a trade-off between the exactness of the analysis and its scalability. In this work, *we allow deriving response time bounds that are not tight* (i.e., our lower bound might be smaller than the actual BCRT of the job and our upper bound might be larger than the actual WCRT of the job) in return for a reduced state-space. However, we make sure that the *analysis remains exact in terms of schedulability*. Namely, the analysis' verdict whether a job set is schedulable or not remains exact. Note that the response-time analysis remains safe, i.e., it always returns a lower and upper bound on the exact BCRT and WCRT, respectively. A safe response-time analysis only satisfies condition (i) in Definition 5.

Safe POR. Next, we need to determine under what conditions a POR of a set of jobs maintains the exactness of the schedulability analysis while also maintaining the safeness of the response-time analysis. Or, in other words, when a partial-order reduction can be considered *safe*. We denote the set of jobs to be considered for reduction in the POR, also called the *reduction set*, by $\mathcal{J}^M$.

Definition 6. A partial-order reduction of a set of jobs $\mathcal{J}^M$ is *safe* if and only if it maintains the exactness of the schedulability analysis (Definition 4) and the safeness of the response-time analysis (i.e., satisfies condition (i) in Definition 5).

5.3.2 Creating the candidate reduction set

The key to having an exact schedulability analysis with POR is that the reduction does not affect the analysis of the execution of jobs not contained in the reduction set.

Definition 7. Given a system state v_p, the *reduction set* $\mathcal{J}^M(v_p)$ is defined as the set of jobs such that there is no other job in $\mathcal{J} \setminus \mathcal{J}^M(v_p)$ that can start executing before all jobs in $\mathcal{J}^M$ finish their execution.

If for a state v_p, the reduction set $\mathcal{J}^M$ is not empty, then all the direct successors of v_p must be part of $\mathcal{J}^M$. We will discuss how to compute $\mathcal{J}^M$ in detail in Section 6.6.1. Having defined the reduction set $\mathcal{J}^M$ and the criteria that should hold for a safe partial-order reduction, we are ready to introduce our problem as follows:

Definition 8. Given a system state v_p, find the reduction set $\mathcal{J}^M(v_p)$ that satisfies the conditions of Definitions 6 and 7.

Chapter 6

Partial-order reduction

This section explains how the *partial-order reduction* (POR) is performed during the expansion phase of the SAG algorithm by Nasri et al. [18]. Figure 6.1 provides a visual high-level overview of the POR algorithm. First, the candidate *reduction set* is created (Section 5.3.2). The next step is to check whether any of the jobs in the reduction set may potentially suffer a deadline miss (Section 6.5). If so, the POR of this particular set of jobs is rejected (Section 6.5). Otherwise, the analysis checks whether there are any jobs that may *interfere* with the execution of the reduction set (Section 6.4). If no such jobs exist, the POR is accepted and the reduction set is added to the SAG as a single scheduling decision (Section 6.7). Section 6.6 explains how to ensure a POR remains *safe* when there are interfering jobs for a reduction set. Finally, Section 6.7 summarizes how to construct the SAG using POR.

6.1 Earliest finish time of a reduction set

When adding a single job J_i to a path P ending in vertex v_p, the original SAG analysis creates a new vertex v_q by computing the EFT and LFT of J_i after v_p. Likewise, to add a reduction set $\mathcal{J}^M$ as a single vertex to path P ending in vertex v_p, we need to compute the EFT and LFT of any job included in $\mathcal{J}^M$ after v_p in order to create v_q. This v_q corresponds to the final vertex the original schedule-abstraction graph algorithm would create after scheduling all jobs in $\mathcal{J}^M$. For instance, in the example of Section 5.2 shown in Figure 5.1c, we generate a single node v_2 that corresponds to the abstract system state after the execution of the set of jobs $\mathcal{J}^M = \{J_1, J_2, J_4\}$ according to any potential execution scenario.

It is important that the EFT we compute for $\mathcal{J}^M$ is exact (i.e., the EFT is a lower bound on the finish time of the set of jobs $\mathcal{J}^M$, and there is an execution scenario where all jobs in $\mathcal{J}^M$ have finished exactly at EFT), as otherwise it would change the execution of jobs that are scheduled after the jobs in the reduction set after applying the POR.

Definition 9. The EFT of a reduction set $\mathcal{J}^M(v_p)$ is *exact* iff the set of jobs $\mathcal{J}^M(v_p)$ cannot complete its execution earlier than EFT and there exists an execution scenario such that all jobs in $\mathcal{J}^M(v_p)$ have completed exactly at EFT.

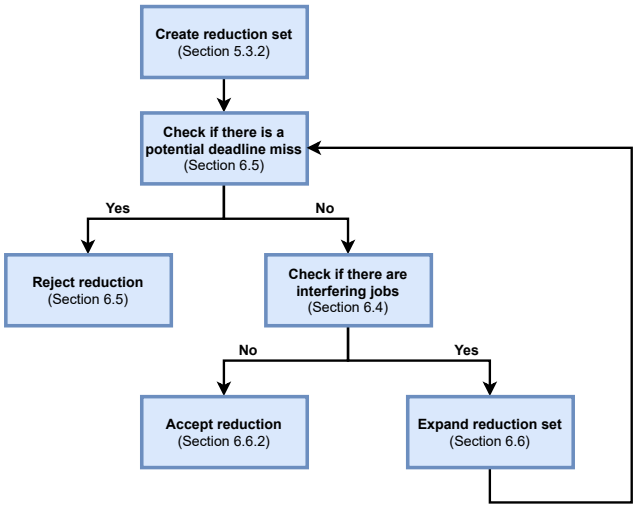


Figure 6.1: **High-level overview of our partial-order reduction algorithm.**

Consider the example in Figure 6.2. The left scenario shows the correct EFT for $\mathcal{J}^M = \{J_1, J_2\}$, namely 17 (obtained by scheduling J_1 and J_2 as early and for as short as possible). As a result, J_3 is always the next job and will finish before its deadline. Now imagine that we end up with an under-estimated EFT of 14 by assuming that the jobs in $\mathcal{J}^M$ start from the earliest time the processor becomes available, as in the right scenario. In this case, the analysis would falsely think that, in addition to the scenario where J_3 executes first, there is an additional execution scenario where J_4 can execute before J_3, in which J_3 may suffer a deadline miss.

The EFT of a set of jobs $\mathcal{J}^M$ can be determined by scheduling all jobs as early as possible and assuming that they have their smallest execution time (BCET). While this does not necessarily result in the EFT for all individual jobs, it will be the EFT of at least one of them (and hence represents the EFT of the entire set). Algorithm 1 shows how to obtain the EFT of a reduction set, i.e., $\overline{EFT}(\mathcal{J}^M, v_p)$.

Algorithm 1: Earliest finish time of $\mathcal{J}^M$ after state v_p

Input : Job set $\mathcal{J}^M$, system state v_p
Output: Earliest finish time $\overline{EFT}(\mathcal{J}^M, v_p)$

```
1 Sort J^M by r^min ascending and break ties by p descending;
2 EFT ← A_1^min(v_p);
3 for each job J_x ∈ J^M do
4 |   EFT ← max{EFT, r_x^min} + C_x^min;
5 end
6 return EFT ;
```

The following lemma shows that the $\overline{EFT}(\mathcal{J}^M, v_p)$ calculated by Algorithm 1 is a tight lower bound for the earliest time at which the processor can possibly

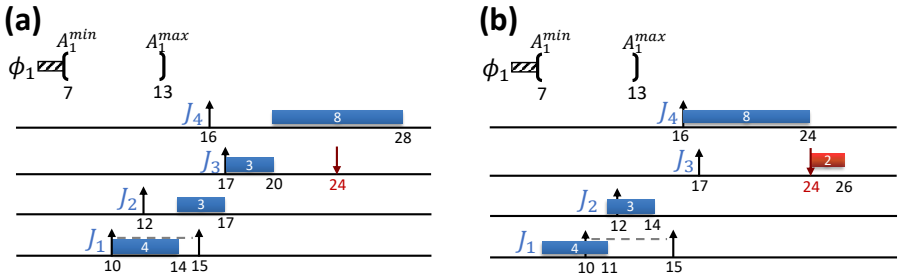


Figure 6.2: **An example showing a scenario with a correct EFT (a) and a scenario with an under-approximated EFT (b). Under-estimating EFT can result in the inclusion of impossible job ordering which will never happen in reality. Here,** $\mathcal{J}^M = \{J_1, J_2\}$**, and** $\mathcal{J} = \{J_1, J_2, J_3, J_4\}$ **with priority assignment** $p_1 < p_2 < p_3 < p_4$**. Furthermore, the execution time of the jobs are as follows:** $C_1 \in [4, 4]$**,** $C_2 \in [3, 3]$**,** $C_3 \in [2, 3]$**, and** $C_4 \in [8, 10]$

become available after dispatching the jobs in set $\mathcal{J}^M$ (in any possible order that the jobs in $\mathcal{J}^M$ can have in their execution scenarios) when they are dispatched after state v_p.

Lemma 1. *The set of jobs* $\mathcal{J}^M$ *scheduled after state* v_p *cannot complete its execution earlier than* $\overline{EFT}(\mathcal{J}^M, v_p)$ *as returned at line 6 of Algorithm 1.*

Proof. At line 1 of Algorithm 1, the jobs in $\mathcal{J}^M$ are sorted in ascending order of r^{min}, and any ties are broken by highest priority. Let J_k denote the k^{th} job in the ordered set $\mathcal{J}^M$, and let $J_k^M = \{J_1, J_2, \ldots, J_k\}$ denote the re-ordered (re-indexed) set of jobs in $\mathcal{J}^M$. And finally, let $\overline{EFT}_k$ be the value computed at line 4 of Algorithm 1 after the k^{th} iteration of the for-loop.

We prove by induction that $\overline{EFT}_k$ is a lower bound on the finish time of all jobs in J_k^M assuming that no job $J_j \notin J_k^M$ executes between the jobs in J_k^M. The base case considers the first job J_1. Job J_1 is the job with the earliest r^{min} in $\mathcal{J}^M$, and in case of a tie, the highest-priority one. We prove that line 4 of Algorithm 1 computes a lower bound on the EFT of J_1. Since a job cannot start executing before it is released and the processor is available, $\max\{A_1^{min}, r_1^{min}\}$ is a lower bound on the start time of J_1. Therefore, J_1 cannot finish before $\max\{A_1^{min}, r_1^{min}\} + C_1^{min}$ as computed at line 4 since C_1^{min} is the minimum execution time of J_1.

In the induction step, $\overline{EFT}_{k-1}$ is a lower bound on the finish time of the jobs in $\{J_1, \ldots, J_{k-1}\}$ assuming no other job executes between them. We show that $\overline{EFT}_k$ as computed at line 4 of Algorithm 1 is the EFT of all jobs $\{J_1, \ldots, J_k\}$. We divide the proof into two cases depending on whether J_k starts its execution before or after the completion of the jobs in $J_{k-1}^M = \{J_1, \ldots, J_{k-1}\}$.

Case (i): Assume that J_k does not start its execution before the jobs in J_{k-1}^M have finished, namely, it does not start before $\overline{EFT}_{k-1}$. We know that J_k cannot start before r_k^{min} since J_k cannot start before it is released. Thus, $\max\{\overline{EFT}_{k-1}, r_k^{min}\}$ is a lower bound on the start time of J_k. Since C_k^{min} is the minimum execution time of J_k, if J_k starts executing at $\max\{\overline{EFT}_{k-1}, r_k^{min}\}$ then it cannot finish before $\overline{EFT}_k = \max\{\overline{EFT}_{k-1}, r_k^{min} + C_k^{min}\}$ as computed at line 4. Furthermore, since J_k starts executing after all jobs in J_{k-1}^M completed

their own execution, the finish time $\overline{EFT}_k$ of J_k is also an upper bound on the finish time of all the other jobs in $J_k^M = J_{k-1}^M \cup \{J_k\}$.

Case (ii): Assume that J_k starts executing before $\overline{EFT}_{k-1}$. Let s_k be the start time of J_k. By assumption, $s_k < \overline{EFT}_{k-1}$. Since J_k cannot start before it is released, $r_k^{min} \leq s_k$. We show that $\overline{EFT}_k$ computed as $\overline{EFT}_k = \overline{EFT}_{k-1} + C_k^{min}$ by Algorithm 1 is a lower bound on the finish time of $J_k^M = J_{k-1}^M \cup \{J_k\}$. Since all jobs in J_{k-1}^M have their release before that of J_k by line 1 of Algorithm 1, the processor executes at least $\overline{EFT}_{k-1} - s_k$ time units of workload from the jobs in J_{k-1}^M *after* s_k. Thus, when including J_k, the processor must execute at least $\overline{EFT}_{k-1} - s_k + C_k^{min}$ time units of workload of J_k^M after s_k. Therefore, the jobs in J_k^M cannot complete before $s_k + \overline{EFT}_{k-1} - s_k + C_k^{min} = \overline{EFT}_{k-1} + C_k^{min}$, which is $\overline{EFT}_k$ as computed at line 4 of Algorithm 1.

Hence, if we apply the inductive step to all jobs in $\mathcal{J}^M$, then line 6 of Algorithm 1 returns a lower bound on the earliest time the processor can finish scheduling all jobs in $\mathcal{J}^M$. □

Lemma 2. *There exists an execution scenario such that all jobs in $\mathcal{J}^M$ have completed exactly at $\overline{EFT}(\mathcal{J}^M, v_p)$ as returned by line 6 of Algorithm 1.*

Proof. If each $J_i \in \mathcal{J}^M$ releases at r_i^{min} and executes for exactly C_i^{min} time units and the processor becomes available at A_1^{min}, then, the execution of $\mathcal{J}^M$ will complete exactly at $\overline{EFT}(\mathcal{J}^M, v_p)$ as returned by line 6 of Algorithm 1 since this algorithm practically simulates the schedule of the jobs in $\mathcal{J}^M$ under the given execution scenario. Hence, there exists an execution such that all jobs in $\mathcal{J}^M$ have completed exactly at $\overline{EFT}(\mathcal{J}^M, v_p)$. □

Corollary 1. *The $\overline{EFT}(\mathcal{J}^M, v_p)$ as returned by line 6 of Algorithm 1 is the exact earliest finish time of $\mathcal{J}^M$.*

Proof. The set of jobs $\mathcal{J}^M$ cannot complete its execution before $\overline{EFT}(\mathcal{J}^M, v_p)$ (Lemma 1), and there exists an execution scenario such that the jobs in $\mathcal{J}^M$ complete at $\overline{EFT}(\mathcal{J}^M, v_p)$ (Lemma 2). Hence, $\overline{EFT}(\mathcal{J}^M, v_p)$ is exact. □

6.2 Latest finish time of a reduction set

Similarly, the latest finish time we compute for $\mathcal{J}^M$ should also be exact (i.e., the LFT is an upper bound on the finish time of the set of jobs $\mathcal{J}^M$, and there is an execution scenario where all jobs in $\mathcal{J}^M$ have finished exactly at LFT), as otherwise it could also change the execution of jobs that are scheduled after it.

Definition 10. The LFT of a reduction set $\mathcal{J}^M(v_p)$ is *exact* iff the set of jobs $\mathcal{J}^M(v_p)$ cannot complete its execution later than LFT and there exists an execution scenario such that all jobs in $\mathcal{J}^M(v_p)$ have completed exactly at LFT.

Consider the example in Figure 6.3. The left scenario shows the correct LFT for $\mathcal{J}^M = \{J_1, J_2\}$, namely 18, which is obtained by scheduling J_1 and J_2 as late and for as long as possible. Then, J_3 can start and finish before its deadline at time 23. Again, imagine that we compute an imprecise LFT of 20, by assuming that the jobs in $\mathcal{J}^M$ start from the time that the latest job is certainly released, as in the right scenario of Figure 6.3. Now J_3 starts later and misses its deadline.

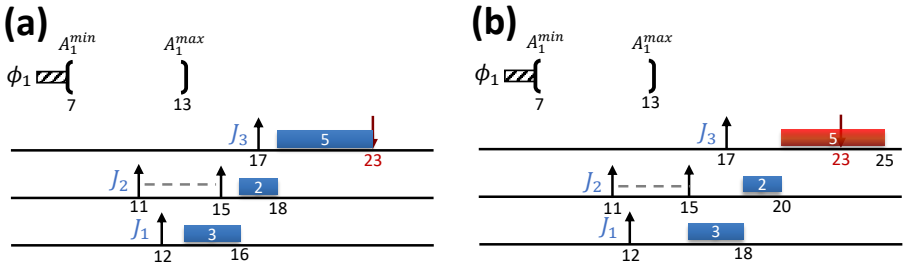


Figure 6.3: **An example showing that over-approximated LFT may result in the inclusion of impossible scenarios. (a) a correct LFT and (b) over-approximated LFT, for the reduction set $\mathcal{J}^M = \{J_1, J_2\}$ and a job set $\mathcal{J} = \{J_1, J_2, J_3\}$ with priority assignment $p_1 < p_2 < p_3$. The execution time of the jobs are as follows $C_1 \in [2, 3]$, $C_2 \in [2, 2]$, and $C_3 \in [1, 5]$.**

Hence, the LFT should be exact as otherwise we could wrongly conclude that a job set is unschedulable.

The latest finish time of a set of jobs $\mathcal{J}^M$ can be determined by scheduling all jobs as late for as long as possible. This property is formally proven in Lemma 3.

Algorithm 2: Latest finish time of $\mathcal{J}^M$ after state v_p

Input : Job set $\mathcal{J}^M$, system state v_p
Output: Latest finish time $\overline{LFT}(\mathcal{J}^M, v_p)$

1 Sort $\mathcal{J}^M$ by r^{max} ascending and break ties by p descending;
2 $\overline{LFT} \leftarrow A_1^{max}(v_p)$;
3 **for** *each job* $J_x \in \mathcal{J}^M$ **do**
4 $\quad \overline{LFT} \leftarrow \max\{\overline{LFT}, r_x^{max}\} + C_x^{max}$;
5 **end**
6 **return** $\overline{LFT}$;

Lemma 3. *The set of jobs $\mathcal{J}^M$ scheduled after state v_p cannot complete their execution later than $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 6 of Algorithm 2.*

Proof. At line 1 of Algorithm 2, the jobs in $\mathcal{J}^M$ are sorted in ascending order of r^{max}, and any ties are broken by highest priority. Let J_k denote the k^{th} job in the ordered set and let $J_k^M = \{J_1, \ldots, J_k\}$ denote the re-indexed set of jobs in $\mathcal{J}^M$ sorted by their r^{max} (in Line 1 of the algorithm). Finally, let $\overline{LFT}_k$ be the value computed at line 4 of Algorithm 2 after the k^{th} iteration of the for-loop.

We prove by induction that $\overline{LFT}_k$ is an upper bound on the finish time of all jobs in J_k^M assuming that no job $J_j \notin J_k^M$ executes between the jobs in J_k^M. The base case considers the first job J_1. J_1 is the job with the earliest r^{max} in $\mathcal{J}^M$, and in case of a tie, the highest priority one. We prove that line 4 of Algorithm 2 computes an upper bound on the LFT of J_1. Because of the work-conserving property of the scheduler, a job cannot start later than the time by which it is certainly released and the processor is certainly available. Hence, $\max\{A_1^{max}, r_1^{max}\}$ is an upper bound on the start time of J_1. If J_1 starts at

$\max\{A_1^{max}, r_1^{max}\}$ it cannot finish after $\max\{A_1^{max}, r_1^{max}\} + C_1^{max}$ as computed at line 4 of Algorithm 2 since C_1^{max} is the maximum execution time of J_1.

In the induction step, $\overline{LFT}_{k-1}$ is an upper bound on the finish time of the jobs in $\{J_1, \ldots, J_{k-1}\}$ assuming no other job executes between them. We show that $\overline{LFT}_k$ as computed at line 4 of Algorithm 2 is the LFT of all jobs $\{J_1, \ldots, J_k\}$. We divide the proof into two cases depending on whether J_k starts its execution before or after the completion of the jobs in $J_{k-1}^M = \{J_1, \ldots, J_{k-1}\}$.

Case (i): If J_k does not start its execution before the jobs in J_{k-1}^M have finished, then, we know that $\overline{LFT}_{k-1}$ is the latest time at which the processor becomes available to other jobs including J_k. We also know that J_k will be released at the latest by time r_k^{max}. Hence, an upper bound on the start time of J_k is $\max\{\overline{LFT}_{k-1}, r_k^{max}\}$ because at that time, a work-conserving scheduler must dispatch a job (in this case, J_k).

Since C_k^{max} is the maximum execution time of J_k, if J_k starts executing at $\max\{\overline{LFT}_{k-1}, r_k^{max}\}$ then it cannot finish after $\overline{LFT}_k = \max\{\overline{LFT}_{k-1}, r_k^{max} + C_k^{max}\}$ as computed at line 4. Furthermore, since J_k starts executing after all jobs in J_{k-1}^M completed their own execution, the finish time $\overline{LFT}_k$ of J_k is also an upper bound on the finish time of all the other jobs in $J_k^M = J_{k-1}^M \cup \{J_k\}$.

Case (ii): If J_k starts executing before $\overline{LFT}_{k-1}$, then let s_k be the start time of J_k. By assumption, $s_k < \overline{LFT}_{k-1}$. We show that $\overline{LFT}_k$ computed as $\overline{LFT}_k = \overline{LFT}_{k-1} + C_k^{max}$ by Algorithm 2 is an upper bound on the finish time of $J_k^M = J_{k-1}^M \cup \{J_k\}$.

Since all jobs in J_{k-1}^M have their latest release (r^{max}) before that of J_k by line 1 of Algorithm 2, the processor executes at most $\overline{LFT}_{k-1} - s_k$ time units of workload from the jobs in J_{k-1}^M after s_k. Thus, the processor must execute at most $\overline{LFT}_{k-1} - s_k + C_k^{max}$ time units of workload of J_k^M after s_k. Therefore, the jobs in J_k^M cannot complete after $s_k + (\overline{LFT}_{k-1} - s_k + C_k^{max}) = \overline{LFT}_{k-1} + C_k^{max}$, which is $\overline{LFT}_k$ as computed at line 4 of Algorithm 2.

Hence, if we apply the inductive step to all jobs in $\mathcal{J}^M$, then line 6 of Algorithm 2 returns an upper bound on the latest time the processor can finish scheduling all jobs in $\mathcal{J}^M$. □

Lemma 4. *There exists an execution scenario such that all jobs in $\mathcal{J}^M$ have completed exactly at $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 6 of Algorithm 2.*

Proof. If each $J_i \in \mathcal{J}^M$ releases at r_i^{max} and executes for exactly C_i^{max} time units and the processor becomes available at A_1^{max}, then, the execution of $\mathcal{J}^M$ will complete exactly at $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 6 of Algorithm 2. Hence, there exists an execution such that all jobs in $\mathcal{J}^M$ have completed exactly at $\overline{LFT}(\mathcal{J}^M, v_p)$. □

Corollary 2. *The $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 6 of Algorithm 2 is the exact latest finish time of $\mathcal{J}^M$.*

Proof. The set of jobs $\mathcal{J}^M$ cannot complete its execution after $\overline{LFT}(\mathcal{J}^M, v_p)$ (Lemma 3), and there exists an execution scenario such that the jobs in $\mathcal{J}^M$ complete at $\overline{LFT}(\mathcal{J}^M, v_p)$ (Lemma 4). Hence, $\overline{LFT}(\mathcal{J}^M, v_p)$ is exact. □

Example 2. Consider the schedules in Figure 6.4. The left shows the schedule that is constructed to compute $\overline{EFT}(\mathcal{J}^M, v_p)$, and the right one shows

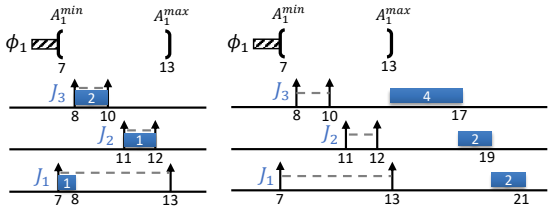


Figure 6.4: **An example showing how $\overline{EFT}(\mathcal{J}^M, v_p)$ and $\overline{LFT}(\mathcal{J}^M, v_p)$ are computed respectively. $\mathcal{J}^M = \{J_1, J_2, J_3\}$ with priority assignment $p_1 < p_2 < p_3$. In this example, C_1 and $C_2 \in [1, 2]$, and $C_3 \in [2, 4]$.**

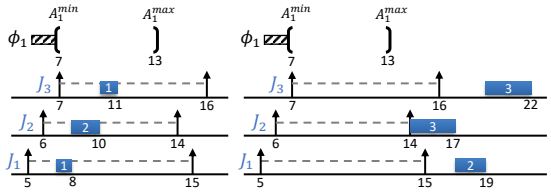


Figure 6.5: **An example showing how $\overline{EFT}(\mathcal{J}^M, v_p)$ and $\overline{LFT}(\mathcal{J}^M, v_p)$ are computed respectively. $\mathcal{J}^M = \{J_1, J_2, J_3\}$ with priority assignment $p_1 < p_2 < p_3$. In this example, $C_1 \in [1, 2]$, $C_2 \in [2, 3]$, and $C_3 \in [1, 3]$.**

the schedule that is constructed to compute $\overline{LFT}(\mathcal{J}^M, v_p)$. Note that for $\overline{EFT}(\mathcal{J}^M, v_p)$, the jobs are scheduled in order of r^{min} and their execution time is set to C^{min}, while for $\overline{LFT}(\mathcal{J}^M, v_p)$, this is r^{max} and C^{max} respectively. Because all r^{max} are before or at A_1^{max}, the jobs are scheduled starting from A_1^{max} as this is the latest time the processor becomes available.

Next, consider the schedules in Figure 6.5. Again, the left and right schedules are respectively for $\overline{EFT}(\mathcal{J}^M, v_p)$ and $\overline{LFT}(\mathcal{J}^M, v_p)$. Now, all r^{min} are before A_1^{min}, so for $\overline{EFT}(\mathcal{J}^M, v_p)$ the jobs are scheduled starting from A_1^{min} as the processor cannot possibly be available before this time. Because all r^{max} after A_1^{max}, the jobs are scheduled at their r^{max} to get the latest finish time of $\mathcal{J}^M$.

6.3 Earliest and latest finish times for each job in a reduction set

Since the analysis is a response-time analysis we also need to compute the EFT and LFT of the individual jobs in $\mathcal{J}^M$. As stated before, the response-time analysis will not be exact as the partial-order reduction algorithm does not explore all possible orderings of jobs in $\mathcal{J}^M$. Yet it remains safe, i.e., the analysis always returns a lower bound on the EFT and an upper bound on the LFT of each job.

6.3.1 Earliest finish time of $J_i \in \mathcal{J}^M$

The earliest finish time of job $J_i \in \mathcal{J}^M$ is lower bounded by

$$\widehat{EFT}_i(\mathcal{J}^M, v_p) = \max\{A_1^{min}(v_p), r_i^{min}\} + C_i^{min} \tag{6.1}$$

Lemma 5. *$J_i \in \mathcal{J}^M$ scheduled after state v_p cannot complete its execution earlier than $\widehat{EFT}_i(\mathcal{J}^M, v_p)$ as defined in* (6.1)*.*

Proof. J_i cannot start before r_i^{min} as a job cannot start before it is released, and J_i cannot start before A_1^{min} as A_1^{min} is the earliest time at which all jobs before J_i finish. Hence, $\max\{A_1^{min}, r_i^{min}\}$ is a lower bound on the start time of J_i. If J_i starts at $\max\{A_1^{min}, r_i^{min}\}$ it cannot finish before $\max\{A_1^{min}, r_i^{min}\} + C_i^{min}$ as C_i^{min} is the minimum execution time of J_i. □

6.3.2 Latest start and finish time of $J_i \in \mathcal{J}^M$

In order to compute an upper bound on the LFT of J_i, we need an upper bound on the start time of J_i. As shown by Davis et al. [8], the following execution scenario results in a late start time for job J_i: a lower priority job starts its execution *just before* J_i is released and hence blocks J_i's execution and, subsequently, all higher-priority jobs interfere with J_i. Since we consider a JLFP scheduling policy, the latest time at which a lower-priority job J_l can have a chance to be dispatched before J_i is when it starts its execution at $r_i^{max} - 1$ (and J_i is released at r_i^{max}) because after $r_i^{max} - 1$, job J_l will no longer be a high-priority job in the system as long as J_i has not been dispatched.

While the above-mentioned scenario is very pessimistic and often does not happen in practice, assuming that scenario allows us to easily compute an upper bound on the start time of J_i using a fixed-point iteration equation. Note that it is possible to compute a tighter upper bound on J_i's start time. However, it would require exploring more scenarios and thus more overhead, which we are trying to minimize. Following the above discussion, we compute an upper bound $s_i(v_p)$ on the start time of J_i in system state v_p using the following recursive equations:

$$s_i^{(0)} = \max\{A_1^{max}, r_i^{max} - 1 + \max_{\forall J_j \in \mathcal{J}^M} \{C_j^{max} \mid p_i < p_j\}\} \tag{6.2}$$

$$s_i^{(k)} = s_i^{(0)} + \sum_{\{J_j | J_j \in \mathcal{J}^M \wedge r_j^{min} \leq s_i^{(k-1)} \wedge p_j < p_i\}} C_j^{max} \tag{6.3}$$

The upper bound $s_i(v_p)$ is then equal to $s_i^{(k)}$ when $s_i^{(k)} = s_i^{(k-1)}$.

Lemma 6. *The fixed-point iteration in* (6.3) *converges.*

Proof. $s_i^{(k)}$ increases or remains constant since only non-negative terms are added because $\forall J_j$, $C_j^{max} \geq 0$. Furthermore, $s_i^{(k)}$ increases when there exists $J_j \in \mathcal{J}^M$ such that J_j has a higher priority than J_i and $s_i^{(k-2)} < r_j^{min} \leq s_i^{(k-1)}$. If such a J_j does not exist, there is no more job that has a higher priority than J_i and is possibly released at $s_i^{(k-1)}$ and $s_i^{(k)} = s_i^{(k-1)}$. Therefore, as long as it does not converge, (6.3) must add at least one more job from $\mathcal{J}^M(v_p)$ at every iteration. Since the number of jobs in $\mathcal{J}^M$ is finite, the number of iterations by (6.3) is upper bounded by $|\mathcal{J}^M(v_p)|$. This proves the lemma. □

Lemma 7. *In system state v_p, $J_i \in \mathcal{J}^M$ starts executing no later than $s_i(v_p)$.*

Proof. The proof is by contradiction. Assume that the processor starts executing $J_i \in \mathcal{J}^M$ later than $s_i(v_p)$. Then, there should either be a larger *blocking by a lower-priority job* or a *larger interference by higher-priority jobs* than accounted for by $s_i(v_p)$. We divide the proof into two cases depending on whether there is a larger blocking or interference for J_i. We show that there cannot be a larger blocking or interference than what is already included in $s_i(v_p)$.

Case (i): There is a larger blocking for J_i. This means either the processor becomes possibly available later than A_1^{min}, or there exists a job with a lower priority that can execute for longer than what is used in (6.2). The former contradicts the assumption that A_1^{min} is the exact earliest time at which the processor becomes possibly available. The latter contradicts the assumptions that a lower-priority job cannot execute for longer than C_j^{max} before J_i starts, and that (6.2) uses the lower-priority job with the largest C^{max}.

Case (ii): There is a larger interference for J_i. (6.3) terminates when there is no more higher-priority job that possibly releases at $s_i^{(k-1)}$. If J_i can start later than $s_i(v_p)$, the jobs currently included in $s_i(v_p)$ can execute longer than their C^{max}, or there must be a higher-priority job that is released before J_i starts but is not included in $s_i(v_p)$. The former is not possible as C^{max} is the maximum execution time of a job. The latter means that there is still a J_j with $p_j < p_i$ and $r_j^{min} < s_i(v_p)$. But this contradicts the assumption that (6.3) has terminated.

Hence, when (6.3) terminates, the resulting $s_i^{(k)}$ is the latest time instant the processor may start scheduling $J_i \in \mathcal{J}^M$ after v_p. □

A second upper bound on the start time of J_i is also given by $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$ (as we will show in Lemma 8). We need this second upper bound because in some cases, $s_i^{(k)}$ can become larger than $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$ and we would prefer a smaller (less pessimistic) upper bound on the LFT of a job in $\mathcal{J}^M$.

Lemma 8. *The processor starts executing $J_i \in \mathcal{J}^M$ after v_p no later than $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$.*

Proof. By contradiction. Assume that the processor starts executing $J_i \in \mathcal{J}^M$ later than $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$, say at $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max} + x$, where $x > 0$. Since, according to the definition of $\mathcal{J}^M$, no job $J_j \notin \mathcal{J}^M$ can start executing before all jobs in $\mathcal{J}^M$ finish their own execution, it means that that the processor is busy executing jobs in $\mathcal{J}^M \setminus J_i$ until $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max} + x$. Consequently, at least one job in $\mathcal{J}^M \setminus J_i$ finishes at $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max} + x$. And then, J_i starts at $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max} + x$. Now if J_i executes for its worst-case execution time C_i^{max}, then it finishes at $\overline{LFT}(\mathcal{J}^M, v_p) + x$. This contradicts Corollary 1 that states that $\overline{LFT}(\mathcal{J}^M, v_p)$ is the latest time instant the processor may be busy executing jobs in $\mathcal{J}^M$. □

Combining these two upper bounds, we define $\widehat{LST}_i(\mathcal{J}^M, v_p)$ as an upper bound on the latest start time of $J_i \in \mathcal{J}^M$.

$$\widehat{LST}_i(\mathcal{J}^M, v_p) = \min\{s_i(v_p), \overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}\} \tag{6.4}$$

Example 3. The top left schedule in Figure 6.6 shows how the first upper bound on the latest start time of J_2, $s_2(v_p)$, is computed by (6.2) and (6.3). J_3 gives the largest blocking compared to A_1^{max}, thus it is scheduled at $r_2^{max} - 1$.

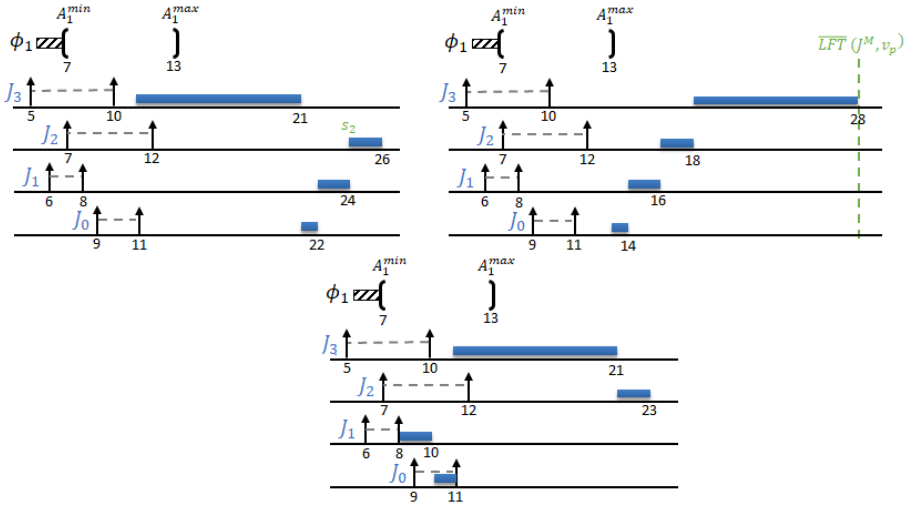


Figure 6.6: **An example showing the scheduling scenario for $s_2(v_p)$ as computed in (6.2) and (6.3) (top left), the scenario for computing $\overline{LFT}(\mathcal{J}^M, v_p)$ (top right), and the valid scheduling scenario that results in the true latest start time for J_2 after v_p (bottom). $\mathcal{J}^M = \{J_0, J_1, J_2, J_3\}$ with priority assignment $p_0 < p_1 < p_2 < p_3$. In this example, $C_0 \in [1, 1]$, $C_1, C_2 \in [1, 2]$, and $C_3 \in [7, 10]$.**

Next, all higher priority jobs that can interfere are scheduled, namely J_0 and J_1. After scheduling these jobs, there is nothing left that can push J_2 any further, so $s_2 = 24$. Note that this schedule would actually be impossible, as for J_3 to start at time 11, J_0 and J_1 would have to be finished already as they have a higher priority than J_3 and are both certainly released by time 11.

The top right schedule depicts the second upper bound given by $\overline{LFT}(\mathcal{J}^M, v_p) - C_2^{max} = 28 - 2 = 26$. This is larger than $s_2(v_p)$, showing that (6.2) and (6.3) can give a tighter upper bound than $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$.

The true latest start time of J_2 is shown in the bottom schedule, where indeed J_0 and J_1 execute before J_3 so that J_3 can push J_2 as far as possible, resulting in a start time at time 21. This shows that neither $s_2(v_p)$ nor $\overline{LFT}(\mathcal{J}^M, v_p) - C_2^{max}$ is a tight upper bound on the latest start time of J_2.

Example 4. Figure 6.7 shows a set of jobs where $s_2 = 25$ (left) is larger than $\overline{LFT}(\mathcal{J}^M, v_p) - C_2^{max} = 21$ (right). This shows that in some cases, $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$ can give a tighter upper bound on the latest start time of J_i than $s_i(v_p)$.

Since the latest start time of $J_i \in \mathcal{J}^M$ is upper bounded by $\widehat{LST}_i(\mathcal{J}^M, v_p)$, the latest finish time of $J_i \in \mathcal{J}^M$ is upper bounded by

$$\widehat{LFT}_i(\mathcal{J}^M, v_p) = \widehat{LST}_i(\mathcal{J}^M, v_p) + C_i^{max} \tag{6.5}$$

Lemma 9. *$J_i \in \mathcal{J}^M$ scheduled after state v_p cannot complete its execution later than $\widehat{LFT}_i(\mathcal{J}^M, v_p)$ as defined in (6.5).*

Proof. As $\widehat{LST}_i(\mathcal{J}^M, v_p)$ is the latest start time of J_i (Lemma 7), J_i cannot start later than $\widehat{LST}_i(\mathcal{J}^M, v_p)$. If J_i starts at $\widehat{LST}_i(\mathcal{J}^M, v_p)$, it cannot finish

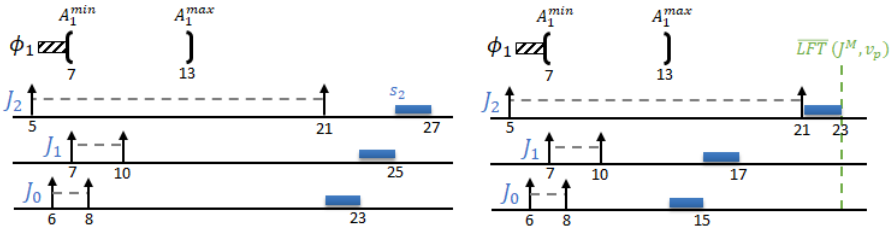


Figure 6.7: **An example where $s_2(v_p)$ is larger than $\overline{LFT}(\mathcal{J}^M, v_p) - C_2^{max}$. $\mathcal{J}^M = \{J_0, J_1, J_2\}$ with priority assignment $p_0 < p_1 < p_2$. The execution time range for all jobs is $[1, 2]$.**

later than $\widehat{LST}_i(\mathcal{J}^M, v_p) + C_i^{max}$ because C_i^{max} is the maximum execution time of J_i. □

6.4 Interfering jobs

In this section, we discuss the properties that will allow us to construct a safe reduction set $\mathcal{J}^M$, where no execution scenario that may lead to a deadline miss is overlooked (missed). To understand the importance of this requirement, let's look at the following example. Consider the scheduling scenario in Figure 6.11. Reducing J_1, J_2, and J_4 to a single edge in the schedule-abstraction graph would remove the depicted scheduling scenario where J_3 executes before J_4 and causes a deadline miss for J_4. Keeping this scenario in the graph is thus important, as otherwise the analysis would not see the deadline miss for J_4 and incorrectly conclude that this job set is schedulable. Therefore, the partial order reduction of this particular set of jobs is unsafe.

As shown in the example above, one case where aggregating scheduling decisions for a set of jobs $\mathcal{J}^S \subseteq \mathcal{J} \setminus \mathcal{J}^P$ may affect the analysis of the execution of other jobs is when some other job $J_j \notin \mathcal{J}^S$ is able to execute between two jobs in $\mathcal{J}^S$. We call such a job J_x that can execute between two arbitrary jobs in $\mathcal{J}^S$ but is itself not in $\mathcal{J}^S$ an *interfering* job.

Definition 11. A job $J_x \notin \mathcal{J}^S$ is an *interfering* job for $\mathcal{J}^S$ iff J_x can execute before any of the jobs in $\mathcal{J}^S$.

The conditions that should hold for a job J_x to be able to interfere with $\mathcal{J}^S$ relate to the *priority-driven* and *work-conserving* properties of the scheduling algorithm.

In the rest of this section, we assume that $\mathcal{J}^S$ is a set of candidate jobs that is being considered when constructing the final reduction set $\mathcal{J}^M$.

6.4.1 Work-conserving interference condition

A job J_x can execute between two jobs in $\mathcal{J}^S$ if J_x is released before the end of an idle interval between the execution of two jobs in $\mathcal{J}^S$. Figure 6.8 shows an example of a job J_x (here, it is J_3) that is released before the end of an idle interval between two jobs in $\mathcal{J}^S = \{J_1, J_2\}$. In this example, we see that even though J_3 has a lower priority than J_1 and J_2, it is able to execute between these jobs because the scheduling algorithm is work-conserving and there can

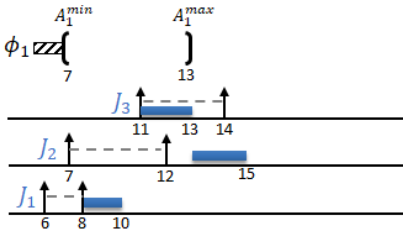


Figure 6.8: **An example where J_3 interferes with the partial-order reduction of $\mathcal{J}^S = \{J_1, J_2\}$ because J_3 releases before the end of an idle time in $\mathcal{J}^S$. $\mathcal{J} = \{J_1, J_2, J_3\}$ with priority assignment $p_1 < p_2 < p_3$. The execution time range for J_1, J_2 and J_3 is $[1, 2]$.**

be an idle interval between them (i.e., when J_1 is released at 6 and J_2 is released at 12).

Computing idle intervals in $\mathcal{J}^S$

In order to find out whether a job J_x is released before the end of an idle interval between the execution of jobs in $\mathcal{J}^S$, we need to determine the possible idle intervals between the execution of jobs in $\mathcal{J}^S$. To do so, we use the property stated in the following Lemma:

Lemma 10. *The latest idle interval before the execution of $J_i \in \mathcal{J}^S$ cannot end later than r_i^{max}.*

Proof. By contradiction. Suppose that there could be an idle interval before the execution of $J_i \in \mathcal{J}^S$ that ends at $r_i^{max} + 1$. Then it means that the processor remains idle until $r_i^{max} + 1$ even though J_i is certainly released. This contradicts the assumption that the scheduling policy is work-conserving, as a work-conserving scheduling policy never keeps the processor idle when there is a ready job. □

To determine whether there is an idle interval ending by r_i^{max}, we schedule as little workload as possible before r_i^{max} and see whether that minimum workload is enough to keep the processor busy until r_i^{max}. A minimal workload before r_i^{max} can be obtained by only scheduling jobs that are certainly released before r_i^{max} as early as possible and considering that they run for their BCET, while scheduling all other jobs *after* r_i^{max}.

As a first step, we define the set of jobs in $\mathcal{J}^S$ that are certainly released before r_i^{max} in Definition 12. We then characterize the largest possible idle interval that may end in r_i^{max} in Lemma 11.

Definition 12. Given a system state v_p and a candidate reduction set $\mathcal{J}^S$, the set of jobs $\mathcal{C}(t, v_p) \subseteq \mathcal{J}^S$ that are *certainly released before time t* is defined as

$$\mathcal{C}(t, v_p) = \{J_j \in \mathcal{J}^S \mid r_j^{max} < t\} \tag{6.6}$$

Lemma 11. *Let J_i be a job in $\mathcal{J}^S$ that is released at r_i^{max}. If there is an idle interval just before the execution of J_i starts, then $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ is a lower bound on the start time of that idle interval.*

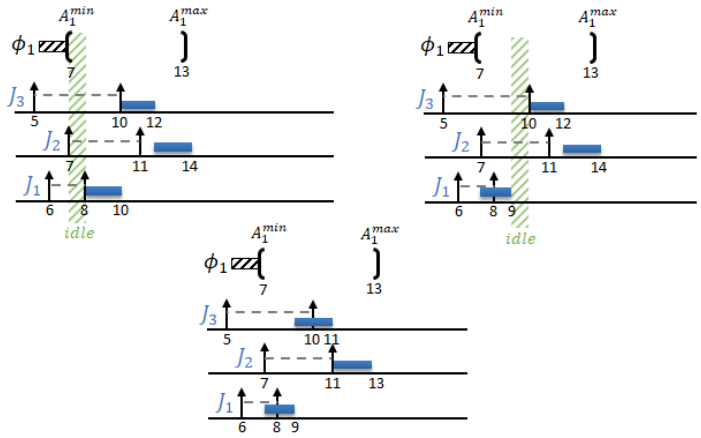


Figure 6.9: **An example showing how to find all idle intervals ending with jobs in $\mathcal{J}^S = \{J_1, J_2, J_3\}$. The priority assignment is $p_1 < p_2 < p_3$. The execution time range for each job is $[2, 2]$.**

Proof. Since by Definition 12, all jobs in $\mathcal{C}(r_i^{max}, v_p)$ must have been released strictly before r_i^{max}, if there is an idle interval before J_i starts to execute, then all the jobs in $\mathcal{C}(r_i^{max}, v_p)$ must have completed their execution by r_i^{max} (by the work-conserving property).

Now, by contradiction, suppose that there is an idle interval that starts before $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$. This means that the jobs in $\mathcal{C}(r_i^{max}, v_p)$ can finish before $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$. This contradicts the fact that $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ is the earliest finish time of any job in $\mathcal{C}(r_i^{max}, v_p)$ as proven in Corollary 1. □

Example 5. Figure 6.9 shows how to find all idle intervals ending with jobs in $\mathcal{J}^S = \{J_1, J_2, J_3\}$. The top left schedule shows how the idle interval ending with J_1 is found. Since there is no job that is certainly released before J_1, $\overline{EFT}(\mathcal{C}(r_1^{max}, v_p), v_p) = A_1^{min} = 7$ resulting in an idle interval $[7, 8)$ for J_1.

The top right schedule shows the idle interval ending with J_2. J_2 is scheduled as early and as short as possible, so $\overline{EFT}(\mathcal{C}(r_2^{max}, v_p), v_p) = 9$, resulting in an idle interval $[9, 10)$ for J_2. Finally, the bottom schedule shows how to find the idle interval ending with J_3 by scheduling J_1 and J_2 as early and as short as possible. However, $\overline{EFT}(\mathcal{C}(r_2^{max}, v_p), v_p) = 11 = r_3^{max}$, making the idle interval $[11, 11)$ and therefore empty. Thus, there is no idle interval possible that ends at r_3^{max}.

If $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ is strictly smaller than r_i^{max}, then an idle interval exists that starts at $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ and ends by r_i^{max}. If otherwise, $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p) = r_i^{max}$, then there will be no idle interval immediately before r_i^{max}. To determine whether it is possible for some job J_x to interfere with $\mathcal{J}^S$ because of the work-conserving property of the scheduling algorithm, we are only interested in the nonempty idle intervals between the execution of jobs in $\mathcal{J}^S$. If there is an empty (i.e., no) idle interval between two jobs in $\mathcal{J}^S$, then the work-conserving property of the scheduler could not be the reason for some job J_x to execute between these two jobs. We denote the set of jobs in $\mathcal{J}^S$ with a nonempty idle interval ending with their release by $\mathcal{J}^\delta$.

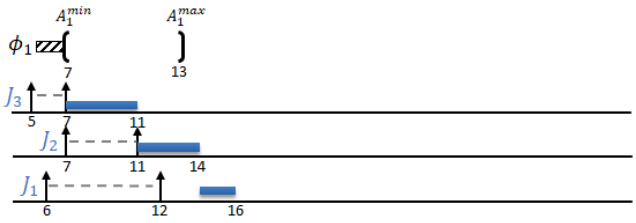


Figure 6.10: **An example showing how to find all idle intervals ending with jobs in $\mathcal{J}^S = \{J_1, J_2, J_3\}$ with priority assignment $p_1 < p_2 < p_3$. The execution time range for J_1 is $[2, 2]$, for J_2 it is $[3, 3]$ and for J_3 it is $[4, 4]$.**

Definition 13. The set of jobs defining the end of a nonempty idle interval $\mathcal{J}^\delta$ is defined as

$$\mathcal{J}^\delta = \{J_i \mid J_i \in \mathcal{J}^S \ \wedge \ \overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p) < r_i^{max}\} \tag{6.7}$$

If for a given $\mathcal{J}^S$, the set $\mathcal{J}^\delta$ is empty, then there cannot be any idle intervals between the jobs in $\mathcal{J}^S$. The following lemma proves this claim.

Lemma 12. *If $\mathcal{J}^\delta = \emptyset$, there exists no execution scenario such that there is an idle interval between the execution of two jobs in $\mathcal{J}^S(v_p)$ scheduled after v_p.*

Proof. If $\mathcal{J}^\delta = \emptyset$ it means that for each $J_i \in \mathcal{J}^S$, $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p) \geq r_i^{max}$. Since $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ is a lower bound on the start time of the idle interval ending with r_i^{max} (Lemma 11) , the idle interval ending with r_i^{max} is empty for each r_i^{max}. Hence, there is no idle interval between any two arbitrary jobs in $\mathcal{J}^S$. □

Example 6. Consider the schedule in Figure 6.10 that shows how to find all idle intervals between jobs in $\mathcal{J}^S = \{J_1, J_2, J_3\}$. Note that in this example we only need one schedule to find all idle intervals. The schedule shows how the idle interval ending with J_1 is found. Since there is no job that is certainly released before J_1, $\overline{EFT}(\mathcal{C}(r_1^{max}, v_p), v_p) = A_1^{min} = 7$ resulting in an empty idle interval $[7, 7)$ for J_1. Next, the schedule also shows the idle interval ending with J_2. J_1 is scheduled as early and as short as possible, so $\overline{EFT}(\mathcal{C}(r_1^{max}, v_p), v_p) = 11$, resulting in an empty idle interval $[11, 11)$ for J_2. Finally, the schedule shows how to find the idle interval ending with J_3 by scheduling J_1 and J_2 as early and as short as possible. $\overline{EFT}(\mathcal{C}(r_2^{max}, v_p), v_p) = 14$, resulting in an empty idle interval $[14, 12)$ for J_2. As all idle intervals are empty, there is no time instant the processor may be idle between the execution of any two jobs in $\mathcal{J}^S$.

When determining whether some job $J_x \notin \mathcal{J}^S$ can potentially interfere with $\mathcal{J}^S$ by executing in an idle interval between two jobs in $\mathcal{J}^S$, we only care whether J_x is released before the end of *some* idle intervals between the jobs of $\mathcal{J}^S$. Namely, we do not need to know which idle interval in particular. Hence, we only need to determine the latest time at which an idle interval ends. We denote that time instant by δ_M.

$$\delta_M(v_p) = \max\{r_j^{max} \mid J_j \in \mathcal{J}^\delta\} \tag{6.8}$$

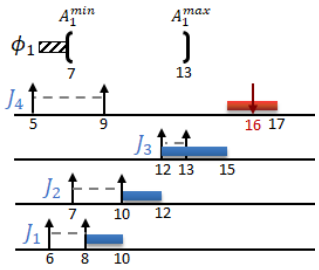


Figure 6.11: **An example where J_3 interferes with the partial-order reduction of $\mathcal{J}^S = \{J_1, J_2, J_4\}$ because J_3 releases before a lower-priority job in $\mathcal{J}^S$. $\mathcal{J} = \{J_1, J_2, J_3, J_4\}$ with priority assignment $p_1 < p_2 < p_3 < p_4$. The execution time range for J_1, J_2, and J_4 is $[1, 2]$ and for J_3 it is $[1, 3]$.**

Using $\delta_M(v_p)$, we can formulate the condition that should hold for a job $J_x \notin \mathcal{J}^S$ to be able to interfere with the jobs in $\mathcal{J}^S$ due to the work-conserving property of the the scheduling algorithm.

Lemma 13. *If a job $J_x \notin \mathcal{J}^S$ can execute in an idle interval between two jobs in $\mathcal{J}^S$, then*

$$\mathcal{J}^\delta \neq \emptyset \wedge r_x^{min} < \delta_M(v_p) \tag{6.9}$$

Proof. If $\mathcal{J}^\delta \neq \emptyset$, $\delta_M(v_p)$ is the latest end of an idle interval ending before the execution of the last job in $\mathcal{J}^S$. J_x is released before $\delta_M(v_p)$, so J_x will be a ready job at some idle instant before $\delta_M(v_p)$. As the scheduler is work-conserving it will not leave the processor idle when there is a ready job, and hence will schedule J_x. Thus, J_x can execute between two jobs in $\mathcal{J}^S$ and interferes with $\mathcal{J}^S$. □

6.4.2 Priority-driven interference condition

A job $J_x \notin \mathcal{J}^S$ can execute between two arbitrary jobs in $\mathcal{J}^S$ if it has not been scheduled yet, it has a higher priority than a job $J_l \in \mathcal{J}^S$, and J_x is released before J_l started to execute. The case depicted in Figure 6.11 is an example of a higher-priority job (J_3 in this case) interfering with $\mathcal{J}^S = \{J_1, J_2, J_4\}$ because it is released before the time at which a lower-priority job in $\mathcal{J}^S$ (i.e., J_4) has started executing.

In order to define the conditions under which there is no interfering job for a candidate reduction set $\mathcal{J}^S$, we first define the set $\mathcal{J}^{high}$ containing the jobs that have a higher priority than at least one job in $\mathcal{J}^S$, formally

$$\mathcal{J}^{high} = \{J_x \mid J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P) \ \wedge \ \exists J_l \in \mathcal{J}^S, p_x < p_l\} \tag{6.10}$$

Lemma 14. *If there exists no $J_x \notin \mathcal{J}^S$ such that (6.9) holds* ***and*** *$\forall J_y \in \mathcal{J}^{high}$, $\forall J_l \in \mathcal{J}^S$ such that $p_y < p_l$ we have $r_y^{min} > \widehat{LST}_l(\mathcal{J}^S, v_p)$, then there exists no interfering job for $\mathcal{J}^S$.*

Proof. Under a non-preemptive work-conserving job-level fixed-priority policy, a job $J_x \notin \mathcal{J}^S$ can only start its execution before a job $J_i \in \mathcal{J}^S$ if and only if: **(i)** J_x has a higher priority than J_i and J_x is possibly released before J_i starts executing **or (ii)** if the processor is idle before the start of J_i, and J_x possibly releases before or during this idle time interval.

Lemma 13 proves that (6.9) must hold if J_x interferes with $\mathcal{J}^S$ by executing in an idle interval. Since by the lemma's assumption, (6.9) does not hold for any job $J_x \notin \mathcal{J}^S$, no job respects condition (ii). Therefore, if the claim does not hold, then there must be a set of jobs $\mathcal{J}^I$ not in $\mathcal{J}^S$ (i.e., $\mathcal{J}^I \cap \mathcal{J}^S = \emptyset$) that satisfies condition (i). That is, for all $J_x \in \mathcal{J}^I$, there exists $J_l \in \mathcal{J}^S$ such that $p_x < p_l$ and J_x is released before J_l starts executing.

According to Lemmas 7 and 8, $\widehat{LST}_l(\mathcal{J}^S, v_p)$ is an upper bound on the start time of an arbitrary job $J_l \in \mathcal{J}^S$ when no job interferes with $\mathcal{J}^S$. Hence, for the jobs in $\mathcal{J}^I$ to interfere with $\mathcal{J}^S$, there must be at least one job $J_y \in \mathcal{J}^I$ and a job $J_l \in \mathcal{J}^S$ such that $p_y < p_l$ and J_y is released before or at $\widehat{LST}_l(\mathcal{J}^S, v_p)$ (from condition (i)). This contradicts the lemma's assumption that $r_y^{min} > \widehat{LST}_l(\mathcal{J}^S, v_p)$. Therefore, the set $\mathcal{J}^I$ must be empty, thereby proving that no job can interfere with $\mathcal{J}^S$. □

6.5 Ensuring the absence of deadline misses

Recall that $\mathcal{J}^S$ is a set of candidate jobs that is being considered when constructing the final reduction set $\mathcal{J}^M$. If $\widehat{LST}_i(\mathcal{J}^S, v_p)$ is larger than the deadline of at least one of the jobs $J_i \in \mathcal{J}^S$, then we cannot reduce the execution of the jobs in $\mathcal{J}^S$ to a single execution scenario, because encountering a deadline miss based on the over-approximated $\widehat{LST}_i(\mathcal{J}^S, v_p)$ does not always imply an actual deadline miss.

Consider the schedule in Figure 6.12. J_1 appears to have a deadline miss since $\widehat{LST}_1(\mathcal{J}^S, v_p) = 15$ so that $\widehat{LFT}_1(\mathcal{J}^S, v_p) = 17 > 16 = d_1$. This execution scenario is possible, so indeed J_1 misses its deadline if we would explore all execution scenarios individually. However, upon finding a job $J_i \in \mathcal{J}^S$ such that $\widehat{LST}_i(\mathcal{J}^S, v_p) > d_i$, we cannot simply conclude that the deadline miss for J_i can actually happen. Consider the example in Figure 6.13. Here, J_2 seems to have a deadline miss because $\widehat{LST}_2(\mathcal{J}^S, v_p) = 16$, so $\widehat{LFT}_2(\mathcal{J}^S, v_p) = 19 > 17 = d_2$. However, this $\widehat{LST}_2(\mathcal{J}^S, v_p)$ is of course an upper bound on the latest start time of J_2. The right schedule in Figure 6.13 shows the actual latest start time of J_2, which is 13. So in reality, J_2 will not miss its deadline because the true LFT of J_2 is 16. Therefore, from considering $\widehat{LST}_2(\mathcal{J}^S, v_p)$ only, we would falsely conclude that J_2 misses its deadline even though it is not possible.

Hence, when creating the final reduction set $\mathcal{J}^M$ from $\mathcal{J}^S$, we need to be certain that even when considering the upper-bounded LFT, all $J_i \in \mathcal{J}^S$ certainly finish before their deadlines.

Lemma 15. *None of the jobs in $\mathcal{J}^S$ misses its deadline if*

$$\forall J_i \in \mathcal{J}^S, \widehat{LFT}_i(\mathcal{J}^S, v_p) \leq d_i \tag{6.11}$$

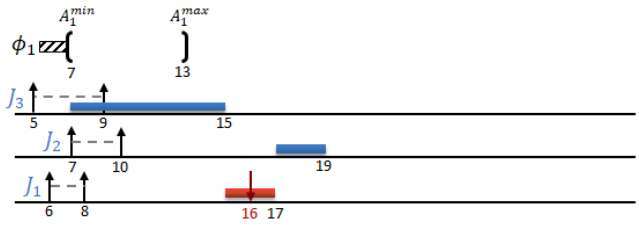


Figure 6.12: **An example where $\widehat{LST}_1(\mathcal{J}^S, v_p)$ correctly indicates a deadline miss for J_1 in $\mathcal{J}^S = \{J_1, J_2, J_3\}$ with priority assignment $p_1 < p_2 < p_3$. The execution time range for J_1, J_2 is $[1, 2]$ and for J_3 it is $[1, 8]$.**

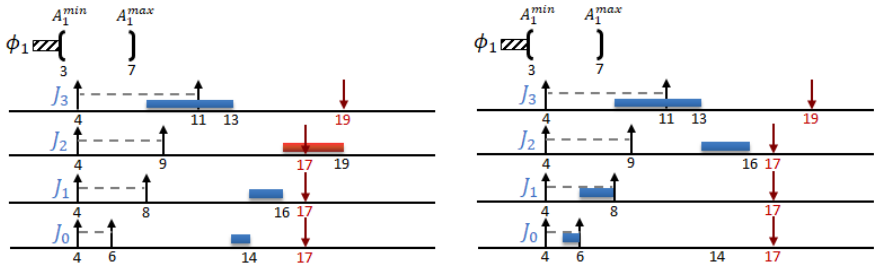


Figure 6.13: **An example where $\widehat{LST}_2(\mathcal{J}^S, v_p)$ falsely indicates a deadline miss for J_2 in $\mathcal{J}^S = \{J_0, J_1, J_2, J_3\}$ with priority assignment $p_0 < p_1 < p_2 < p_3$. In this example, $C_0 \in [1, 1]$, $C_1 \in [1, 2]$, $C_2 \in [2, 3]$, and $C_3 \in [2, 5]$.**

Proof. Lemma 9 shows that $\widehat{LFT}_i(\mathcal{J}^S, v_p)$ is an upper bound on the latest finish time of $J_i \in \mathcal{J}^S$. Hence, if it is smaller than the deadline of J_i, then it is certain that $J_i \in \mathcal{J}^S$ cannot miss its deadline. □

If a partial-order reduction is unsafe because of a potential deadline miss, unfortunately there is not much we can do other than fall back on the original algorithm to explore all possible scenarios. This is needed to ensure that all deadline misses are reported correctly.

6.6 Safe partial-order reduction

If a partial-order reduction is unsafe because there exists a J_x that *can interfere* with candidate reduction set $\mathcal{J}^S$ (see Definition 11), we can add J_x to $\mathcal{J}^S$ and check whether a reduction of this new $\mathcal{J}^{S'} = \mathcal{J}^S \cup \{J_x\}$ is safe. Note that after adding J_x to $\mathcal{J}^S$, we need to recompute all properties of this new $\mathcal{J}^{S'}$, i.e. $\widehat{LST}_i(\mathcal{J}^{S'}, v_p)$, $\widehat{EFT}_i(\mathcal{J}^{S'}, v_p)$, and $\widehat{LFT}_i(\mathcal{J}^{S'}, v_p)$, for all $J_i \in \mathcal{J}^{S'}$, $\overline{EFT}(\mathcal{J}^{S'}, v_p)$, $\overline{LFT}(\mathcal{J}^{S'}, v_p)$, and $\delta_M(v_p)$.

Let's return to the example in Figure 6.8. Here, J_3 is able to interfere with $\mathcal{J}^S = \{J_1, J_2\}$. If we add J_3 to $\mathcal{J}^S$, the partial-order reduction of $\mathcal{J}^{S'} = \{J_1, J_2, J_3\}$ would be safe. Thus, adding J_x to $\mathcal{J}^S$ sometimes allows us to convert an unsafe partial-order reduction to a safe one.

When there are multiple jobs interfering with $\mathcal{J}^S$, the question is in what order they will be added to $\mathcal{J}^S$ as different addition orders can have different

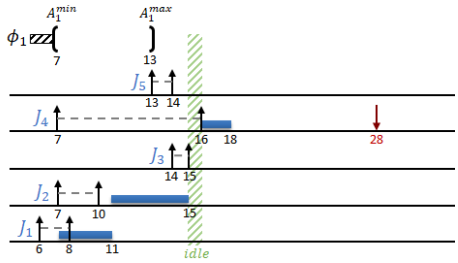


Figure 6.14: **An example where adding interfering jobs to $\mathcal{J}^S = \{J_1, J_2, J_4\}$ in order of descending priority results in a safe partial-order reduction. $\mathcal{J} = \{J_1, J_2, J_3, J_4, J_5\}$ with priority assignment $p_1 < p_2 < p_3 < p_4 < p_5$. In this example, C_1 and C_2 are in $[4, 4]$, $C_3 \in [1, 1]$, $C_4 \in [1, 2]$, and $C_5 \in [1, 8]$.**

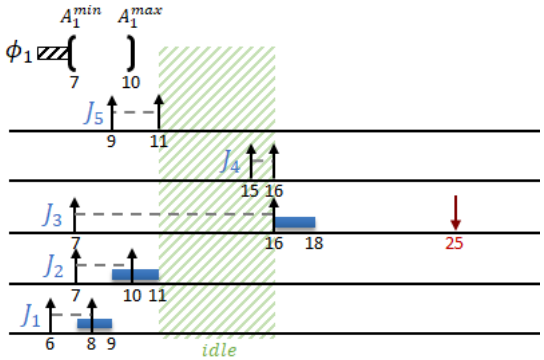


Figure 6.15: **An example where adding interfering jobs to $\mathcal{J}^S = \{J_1, J_2, J_4\}$ in order of ascending r^{min} results in a safe partial-order reduction. $\mathcal{J} = \{J_1, J_2, J_3, J_4, J_5\}$ with priority assignment $p_1 < p_2 < p_3 < p_4 < p_5$. In this example, C_1 and C_2 are in $[2, 2]$, $C_3 \in [1, 2]$, $C_4 \in [1, 10]$, and $C_5 \in [5, 5]$.**

results. Consider the example in Figure 6.14 where $\mathcal{J}^S = \{J_1, J_2, J_4\}$. There is an idle interval $[15, 16)$ and $\overline{LFT}(\mathcal{J}^S, v_p) = 23$. If we add J_3 to $\mathcal{J}^S$ first, there is no idle interval left in which J_5 can be inserted, and since $\overline{LFT}(\mathcal{J}^S, v_p)$ is now 24 there are no deadline misses, so this is a safe partial-order reduction. On the other hand, if we add J_5 to $\mathcal{J}^S$ first, then $\overline{LFT}(\mathcal{J}^S, v_p) = 31$ and $\widehat{LST}_4(\mathcal{J}^S, v_p) = 29$. As we can now no longer rule out a deadline miss for J_4, this partial-order reduction is not safe. So, in this case adding the higher priority job to $\mathcal{J}^S$ first is the better approach as this leads to a safe partial-order reduction.

Now, consider the example in Figure 6.15 where $\mathcal{J}^S = \{J_1, J_2, J_4\}$. There is an idle interval $[11, 16)$ and $\overline{LFT}(\mathcal{J}^S, v_p) = 18$. If we add J_5 to $\mathcal{J}^S$ first, there is no idle interval left in which J_5 can be inserted, and since $\overline{LFT}(\mathcal{J}^S, v_p)$ is now 23 there are no deadline misses, so this is a safe partial-order reduction. On the other hand, if we add J_3 to $\mathcal{J}^S$ first, then $\overline{LFT}(\mathcal{J}^S, v_p) = 28$ and $\widehat{LST}_4(\mathcal{J}^S, v_p) = 26$. As we can now no longer rule out a deadline miss for J_4, this partial-order reduction is not safe. So, in this case adding the earlier released job to $\mathcal{J}^S$ first is the better approach as this leads to a safe partial-order reduction.

This shows that different scenarios have different "optimal" orders of adding interfering jobs to $\mathcal{J}^S$, where optimal means that the addition of J_x to $\mathcal{J}^S$ leads to a successful POR. Note that the non-optimal addition orders we explored in the previous examples are actually scenarios that would have never been possible. For Figure 6.14, J_5 could never execute before J_4 because both J_4 and J_5 have been certainly released at the start of the idle interval at time 15. So, J_4 would execute first because of its higher priority. In Figure 6.15, J_4 can never insert itself between jobs in $\mathcal{J}^S$ because the processor will be certainly idle and J_5 will be certainly released before r_4^{min}, causing J_5 to start before J_4.

However, *we have no way of knowing whether these scenarios are actually impossible without exploring the different execution scenarios in $\mathcal{J}^S$ first.* This would technically mean to manually perform the original SAG analysis on the jobs in $\mathcal{J}^S$ which is obviously not efficient. Consequently, we choose to use a heuristic approach when adding jobs to $\mathcal{J}^S$. Our approach is not optimal and hence may fail to form a reduction set when it is possible to form one. Later in Chapter 7, we will evaluate the performance of the heuristic(s).

6.6.1 Algorithm for reduction set creation

Algorithm 3 shows how to create the final reduction set $\mathcal{J}^M$ for a system state v_p. In short, interfering jobs are added to the set of candidate jobs $\mathcal{J}^S$ until the POR is either accepted because of a lack of interfering jobs, or rejected because of a potential deadline miss.

In line 1 of Algorithm 3, $\mathcal{J}^S$ is initialized with the *direct successors* of v_p, i.e., successors that the original SAG analysis would naturally add immediately after state v_p according to Definition 2. In the while-loop, the algorithm first computes properties of $\mathcal{J}^S$ (line 3) and then it checks whether there is a potential deadline miss for $\mathcal{J}^S$ (line 5). If so, it returns the empty set (line 6) since the partial-order reduction of $\mathcal{J}^S$ is inherently unsafe and therefore rejected. Otherwise, the algorithm continues by searching for the interfering jobs.

Dealing with interfering jobs. Through lines 8-11, the algorithm forms the interfering job set $\mathcal{J}^I$. If this set $\mathcal{J}^I$ is empty (line 12), i.e., there are no jobs interfering with $\mathcal{J}^S$, the partial-order reduction can be considered safe and thus the algorithm returns $\mathcal{J}^S$ as the final reduction set (line 13). If this set is not empty (line 14-17), the algorithm tries to add one of the interfering jobs $J_x \in \mathcal{J}^I$ to the current $\mathcal{J}^S$ (line 15). That job is the one chosen according to the input criterion X.

Criteria to expand the reduction set. We use two greedy criteria (denoted by X) to select one job $J_x \in \mathcal{J}^I$ from the interfering set and add it to the reduction set $\mathcal{J}^S$. The first criterion, called *priority order*, selects the the highest-priority job in $\mathcal{J}^I$. The second heuristic, called *release order*, selects the job with the earliest r_x^{min}. After adding J_x to $\mathcal{J}^S$ (line 16) the while-loop repeats, either until the partial-order reduction is accepted (line 13) or rejected (line 6).

Complexity of Algorithm 3. Recall that $m = |\mathcal{J}|$ is the number of jobs in the whole job set. Determining the properties of $\mathcal{J}^S$ (line 3) can be done in $O(m \log m)$ (because of the sorting involved in computing $\widehat{EFT}_i(\mathcal{J}^S, v_p)$ and $\widehat{LFT}_i(\mathcal{J}^S, v_p)$). Despite the fact that m can be considerably large, in practice, due to the release time of the jobs that is typically widely spread throughout the

Algorithm 3: Algorithm for constructing a reduction set

Input : System state v_p, criterion X
Output: The reduction set $\mathcal{J}^M(v_p)$

$\mathcal{J}^S \leftarrow$ direct successors of v_p (Definition 2);
while *true* **do**
  Compute $\widehat{EFT}_i(\mathcal{J}^S, v_p)$ and $\widehat{LFT}_i(\mathcal{J}^S, v_p)$ $\forall J_i \in \mathcal{J}^S$ using (6.1) and (6.5);
  Compute $\delta_M(v_p)$ using (6.8) ;
  if $\exists J_i \in \mathcal{J}^S$ *s.t.* $\widehat{LFT}_i(\mathcal{J}^S, v_p) > d_i$ **then**
    return $\emptyset$;
  end
  $\mathcal{J}^I \leftarrow \emptyset$;
  while $\exists J_j \in \mathcal{J} \setminus (\mathcal{J}^P \cup \mathcal{J}^S)$ *s.t.* $(\mathcal{J}^\delta \neq \emptyset \wedge r_j^{min} < \delta_M(v_p)$ *or* $\exists J_i \in \mathcal{J}^S$ *s.t.* $p_j < p_i \wedge r_j^{min} \leq \widehat{LST}_i(\mathcal{J}^S, v_p))$ **do**
    $\mathcal{J}^I \leftarrow \mathcal{J}^I \cup \{J_j\}$;
  end
  if $\mathcal{J}^I = \emptyset$ **then**
    return $\mathcal{J}^S$;
  else
    $J_x \leftarrow$ a job in $\mathcal{J}^I$ according to criterion X;
    $\mathcal{J}^S \leftarrow \mathcal{J}^S \cup \{J_x\}$;
  end
end

hyperperiod (or a long observation window), only limited jobs will be included in $\mathcal{J}^S$ and considered in the calculation of $\widehat{EFT}_i(\mathcal{J}^S, v_p)$ and $\widehat{LFT}_i(\mathcal{J}^S, v_p)$. This can be confirmed by our experiments.

Determining $\delta_M(v_p)$ (line 4) happens in $O(m^2)$, as an idle interval for a single job can be determined in $O(m)$. The complexity of the while-loop (lines 9-11) is $O(m^2)$, as we can determine whether a single job interferes in $O(m)$ (finding lower-priority jobs in $\mathcal{J}^S$ takes at most $O(m)$ and the comparisons are $O(1)$). The heuristics we use to select a J_x on line 15 have a complexity of $O(m)$ because we only want to find the highest-priority job or the job with the earliest release and these two can be obtained by traversing over the interfering jobs sets once.

The algorithm performs the above operations until either $\mathcal{J}^S$ is accepted or rejected, so at most m times. Hence, the complexity of Algorithm 3 is $O(m^3)$.

In conclusion, a partial-order reduction of a final reduction set $\mathcal{J}^M$ is safe, i.e., maintains the exactness of the schedulability analysis and the safeness of the response-time analysis, if $\mathcal{J}^M$ is created according to Algorithm 3, namely, **(i)** there exists no J_x such that J_x can interfere with $\mathcal{J}^M$, **(ii)** there exists no $J_i \in \mathcal{J}^M$ such that J_i misses its deadline, **(iii)** the EFT and LFT of $\mathcal{J}^M$ are computed using Algorithms 1 and 2 respectively, and **(iv)** the EFT and LFT of $J_i \in \mathcal{J}^M$ are computed using (6.1) and (6.5) respectively.

Lemma 16. *A partial-order reduction of a reduction set $\mathcal{J}^M$ as returned by line 13 of Algorithm 3 maintains exact schedulability*

Proof. There are no jobs that can interfere with the jobs in $\mathcal{J}^M$ (Lemma 14). Therefore, the earliest finish time and latest finish time of $\mathcal{J}^M$ as computed in Algorithms 1 and 2 respectively, are exact because they are exact under the assumption that the processor schedules no $J_j \in \mathcal{J} \setminus \mathcal{J}^M$ (Corollary 1 and 2 respectively). Hence, the reduction of $\mathcal{J}^M$ to a single scheduling decision will not affect the analysis of the execution of jobs not contained in $\mathcal{J}^M$. Furthermore, the reduction will not affect the schedulability analysis of the reduction set $\mathcal{J}^M$ itself, as no $J_i \in \mathcal{J}^M$ can miss its deadline (Lemma 15). Thus, if the partial-order reduction of $\mathcal{J}^M$ is safe, it will maintain the exactness of the schedulability analysis. □

Lemma 17. *A partial-order reduction of a reduction set $\mathcal{J}^M$ as returned by line 13 of Algorithm 3 maintains safe response-time bounds.*

Proof. As the partial-order reduction of $\mathcal{J}^M$ will not affect the analysis of the execution of jobs not contained in $\mathcal{J}^M$ (Lemma 16), the response-time bounds of jobs not contained in $\mathcal{J}^M$ will remain exact as in the original analysis by Nasri et al. [18]. For the jobs contained in $\mathcal{J}^M$, the earliest and latest finish times are lower and upper bounds respectively (Lemma 5 and 9 respectively). Therefore, the response-time bounds of jobs in $\mathcal{J}^M$ are also safe. □

6.7 Algorithm for partial-order reduction

Algorithm 4 summarizes how to construct the schedule-abstraction graph using partial-order reduction. It is based on the schedule-abstraction graph construction algorithm of Nasri et al. [21], with the addition of partial-order reduction at lines 4-13.

First, the reduction set is created at line 4 using Algorithm 3. If the reduction set is not empty (line 5), the partial-order reduction was accepted, in which case a vertex v_k is created with the EFT and LFT of $\mathcal{J}^M$ (line 6). At line 7, path P is extended by connecting the new vertex v_k to the previous vertex v_p. Next, the algorithm performs the merge phase at lines 8-12, which works just like the merge phase (Section 5.1.2) for single jobs.

If the result from Algorithm 3 is $\emptyset$, it means that the partial-order reduction for the direct successors of v_p was rejected due to a potential deadline miss. In this case, the direct successors of v_p are scheduled individually as per the original schedule-abstraction graph algorithm to obtain exact response-time bounds for these jobs. This allows us exactly conclude whether there was a deadline miss.

Note that the original SAG algorithm by Nasri et al. [18,20,21] builds the graph in a breadth-first manner, as explained in Section 5.1. Therefore, they only need to check whether the states that are on the front of the graph could possibly be merged with a newly created state. As such, only states on the front of the graph are kept in memory. However, Algorithm 4 does not build the graph in an exclusively breadth-first manner since multiple jobs can be added to a single edge (by means of a reduction set). Due to these depth-first elements, not all jobs on the front of the graph will have the same number of jobs. Discarding all states outside of the front potentially removes states that could later be merged with, causing redundant paths to be explored, and thus increasing the potential for state-space explosion. Hence, we store states in a priority queue sorted by the number of scheduled jobs, so that the state with the minimum number of jobs is always the first to be expanded (line 3). This guarantees the other states in the queue have at least as many scheduled jobs, meaning that any potential state to merge with can be found.

Algorithm 4: Algorithm for constructing the schedule-abstraction graph using partial-order reduction

Input : Job set $\mathcal{J}$
Output: Schedule graph $G = (V, E)$

Initialize G by adding a root vertex v_1 with interval $[0, 0]$;
while $\exists$ *path* P *from* v_1 *to a leaf* v_p *s.t.* $|\mathcal{J}^P| < |\mathcal{J}|$ **do**
    $P \leftarrow$ the path with the smallest set $\mathcal{J}^P$ from v_1 to a leaf vertex v_p;
    Create $\mathcal{J}^M$ using Algorithm 3;
    if $\mathcal{J}^M \neq \emptyset$ **then**
        Create v_k with label $[\overline{EFT}(\mathcal{J}^M, v_p), \overline{LFT}(\mathcal{J}^M, v_p)]$ according to Algorithms 1 and 2;
        Connect v_p to v_k by an edge with label $\mathcal{J}^M$;
        while $\exists$ *path* Q *that ends with* v_q *such that* v_k *and* v_q *can be merged (Definition 3)* **do**
            Merge v_k and v_q by updating v_k using (5.8);
            Redirect all incoming edges of v_q to v_k;
            Remove v_q from V;
        end
    end
    else
        for *each direct successor job* J_j *of* v_p *(Definition 2)* **do**
            Create v_k with label $[A_1^{min}(v_k), A_1^{max}(v_k)]$ based on (5.6) and (5.7);
            Connect v_p to v_k by an edge with label J_j;
            while $\exists$ *path* Q *that ends with* v_q *such that* v_k *and* v_q *can be merged (Definition 3)* **do**
                Merge v_k and v_q by updating v_k using (5.8);
                Redirect all incoming edges of v_q to v_k;
                Remove v_q from V;
            end
        end
    end
end

Chapter 7

Empirical evaluation

We conducted experiments to answer the following questions: (i) Does partial-order reduction provide a speedup and state-space reduction over the original schedule-abstraction graph implementation? (ii) Does the order of adding interfering jobs to the reduction set affect the performance of partial-order reduction? (iii) How is the worst-case response time affected by the partial-order reduction? (iv) How do release jitter and execution-time variation affect the speedup?

Baselines. To answer these questions, we applied two variations of Algorithm 4 to task sets with scheduling policy NP-FP (with rate-monotonic priorities). The first variation chooses J_x in Algorithm 3 to be the highest-priority job in $\mathcal{J}^I$ (referred to as *partial-order reduction with priority order*), while the second variation chooses J_x to be the job in $\mathcal{J}^I$ with the earliest r^{min} (referred to as *partial-order reduction with release order*). The original schedule-abstraction graph algorithm from Nasri et al. [18] was used as a baseline (referred to as 'original').

Experiments. We have performed two wide sets of experiments, one using benchmark task sets that follow the specifications of automotive industry (Section 7.1) and one using synthetic task sets that are more representative of commercial systems (Section 7.2) to remove any bias the period distribution may have on the performance of POR. The benchmark task sets have semi-harmonic periods, which is beneficial for original algorithm since jobs are released simultaneously which limits the job combinations the analysis needs to explore. Therefore, we expect to see that POR provides a limited performance gain for benchmark task sets. In contrast, synthetic task sets have random periods, meaning that jobs are released randomly which causes more branching in the original analysis. As such, we expect that POR can provide a significant performance gain on synthetic tasks by combining all these branches.

Figure clarification. Line plots show averages and error bars. The error bars show the 95% confidence intervals of the averages. The gray background behind a data point indicates that this data point has fewer than 10 samples and as such has a lower confidence. Box plots have whiskers at the 2^{nd} and 98^{th} percentiles, and outliers are not shown.

Host. The experiments were performed on a Dutch national cluster called SURFsara Cartesius cluster, with nodes equipped with two Intel Xeon E5-2690 v3 processors clocked at 2.6 GHz and 64 GB RAM. The analysis was implemen-

ted as a single-threaded C++ program.

Metrics. We use the following metrics to evaluate the performance of partial-order reduction.

- **State-reduction ratio:** It is defined as $1 - N^A/N^O$, where N^A is the number of states explored by analysis A for a job set $\mathcal{J}$ and N^O is the number of states explored by the *original* schedule-abstraction-based analysis for the same job set. Namely, the closer the state reduction ratio is to 1, the more states the analysis is able to *remove* from the state-space in comparison to the states explored by the original analysis.

- **POR success ratio:** It is the number of successful attempts for partial-order reductions (Section 6.7) divided by the total number of attempted reductions. The closer it is to 1, the more reductions were successful (i.e., they satisfy the conditions of Definitions 6 and 7).

- **Speedup:** The speedup of an analysis A on a job set $\mathcal{J}$ is defined as the CPU time required by the original analysis to analyze $\mathcal{J}$ divided by the CPU time required by analysis A for $\mathcal{J}$. Therefore, a speedup > 1 indicates that the analysis was faster than the original analysis.

- **Relative WCRT:** It is the WCRT of a task τ_i reported by analysis A divided by the WCRT of τ_i reported by the original analysis. The BCRT and WCRT of τ_i is determined by taking the minimum BCRT and maximum WCRT amongst all jobs of τ_i in the hyperperiod. A relative response time > 1 indicates an over-approximated response time reported by analysis A, while a relative response time < 1 indicates an under-approximation. In Section 7.1.5 we will discuss the relative WCRT for different priority classes in a system.

7.1 Experiment on benchmark task sets

7.1.1 Task set generation

In this experiment, task sets were generated according to the description of automotive benchmark applications by Kramer et al. [16]. They describe period distribution and execution time statistics that are realistic for runnables in automotive applications, where runnables with the same period are grouped into tasks. A runnable is generated by drawing the period from {1, 2, 5, 10, 20, 50, 100, 200, 1000} with probabilities {0.04, 0.02, 0.02, 0.29, 0.29, 0.04, 0.24, 0.01, 0.05} and computing the WCET using the statistics for the respective period as also mentioned in [16].

Using these statistics, we randomly generate a task set with target utilization U by generating runnables until the sum of the utilizations of all runnables was equal to U and grouping these runnables into tasks. Runnables were grouped into tasks in a manner similar to [18]. The procedure for grouping runnables into tasks works as follows. We draw a packing threshold a_i uniformly at random from $[0, 2(T_1 - \sum_{\forall r_j, T_j = T_1} C_j^{max})]$, where r denotes a runnable. Runnables with the same period are aggregated into a task τ_i until C_i^{max} reaches the threshold a_i. The process of randomly choosing an a_i and aggregating runnables into a task is repeated until all runnables are assigned to tasks. A maximum

packing threshold of $2(T_1 - \sum_{\forall r_j, T_j = T_1} C_j^{max})$ is chosen because a necessary schedulability condition for non-preemptive tasks is that $\forall i, 2 \leq i \leq n : C_i^{max} \leq 2(T_1 - C_1^{max})$, where T_1 is the shortest period and n is the number of tasks.

A task set was deemed *unschedulable* as soon as an execution scenario containing a deadline miss was encountered, or when the timeout of four hours was reached. We generated 200 task sets for each configuration of release jitter, execution-time variation, and utilization $U \in \{0.1, \ldots, 0.9\}$.

Design of experiment

This section describes the parameters that were used for the experiments (values for execution time variation and release jitter). We also explain what type of analysis was used (i.e., response-time analysis or schedulability analysis), and how task sets that the analyses timed out on are handled in the results sections.

Experiment 1 (EXP1) The purpose of this experiment is to evaluate the state-reduction ratio, speedup, and WCRT of the analyses. The results will be discussed in Sections 7.1.2, 7.1.3, 7.1.4, and 7.1.5.

Task set parameters. We varied the utilization for four different configurations of release jitter and execution-time variation. For two configurations, the execution-time variation was set to $C_i^{min} = (1 - \beta) \cdot C_i^{max}$ for $\beta \in \{0, 1\}$, with the release jitter fixed to 100µs. For the remaining two configurations, the release jitter was drawn randomly with a uniform distribution from $[0, \alpha \cdot (T_i - C_i)]$ (i.e., the jitter is fraction of the slack of each task) for $\alpha \in \{0, 0.1\}$, with C_i^{min} fixed to $0.8 \cdot C_i^{max}$.

Type of analysis. We continued the analysis until either the end of the hyperperiod or the timeout was reached, even after finding a scenario containing a deadline miss. This allowed us to make a fair comparison between schedulable and unschedulable task sets. Otherwise, we would have an incomplete view on unschedulable task sets, since they would have lower CPU times and smaller state-spaces as the utilization increases because the analysis would be prematurely terminated.

Handling timeouts. We removed timed out task sets from the results because they could not be used to fairly compare response-times and state-spaces between the analyses as they would have progressed at different rates through the job set. Removing these task sets from the speedup results allowed us to directly compare the speedup results with the state-space results, as otherwise, the SAG's runtime would be undefined and hence it would not be possible to derive the speedup for task sets that could not be analyzed within the time limits.

Experiment 2 (EXP2) The purpose of this experiment is to evaluate the effect of release jitter and execution time variation on the speedup provided by POR. The results will be discussed in Section 7.1.6.

Task set parameters. For the experiments in Section 7.1.6, we varied the parameters that determine the release jitter and execution-time variation of the tasks. For the scenarios where the release jitter was varied, the release jitter was drawn uniformly at random from $[0, \alpha \cdot (T_i - C_i)]$ (i.e., the jitter is a fraction of the slack) where $\alpha \in \{0, 0.05, \ldots, 0.5\}$, and C_i^{min} was fixed to $0.8 \cdot C_i^{max}$. For the

scenarios where the execution-time variation was varied, $C_i^{min} = (1 - \beta) \cdot C_i^{max}$ where $\beta \in \{0, 0.1, 0.2, \ldots, 1\}$, and the release jitter was fixed to 100µs.

Type of analysis. We performed a schedulability analysis, i.e., we stopped the analysis as soon as a deadline miss was encountered. The reason for this will be explained in Section 7.1.6.

Handling timeouts. Timed out task sets were not removed as we only consider the speedup in this experiment, for which the timed out task sets are also relevant to include in the comparison.

7.1.2 The effect of the order in which interfering jobs are added to the reduction set (EXP1)

In this section, we determine whether the order in which interfering jobs are added to the reduction set $\mathcal{J}^M$ have any effect on the performance of partial-order reduction, in terms of speedup, state-space size, and WCRT.

Observation 1. There is no significant difference between adding jobs in order of highest priority and adding them in order of earliest release in terms of speedup, relative WCRT, and state-reduction ratio.

Figures 7.1[1], 7.2, 7.3, and 7.4 show speedup results for the four different configurations of execution-time variation and release jitter described in Section 7.1.1. The average speedup based on all tasks, schedulable tasks, and unschedulable tasks (shown in d, e, and f respectively) is very similar if not the same for both POR with priority order and POR with release order in most cases. The averages have a more apparent difference in a handful of cases. Figures 7.3d, e, and f show that release order has a slightly larger average speedup for task sets with no jitter. Figure 7.4f shows that release order has a larger average speedup for unschedulable task sets with $\alpha = 0.1$ and $U = 0.2$ and $U = 0.3$, while priority order has a larger speedup for $U = 0.9$. The 2^{nd} and 98^{th} percentiles are a bit more varied between the two, with sometimes priority order and other times release order having a larger value. Table 7.1 contains an overview of the statistics of the speedup, which confirms the similarity of both solutions. So there is no clear winner between priority order and release order when considering the speedup.

Table 7.2 shows the statistics of the relative WCRT for all task sets by analysis, i.e., for partial-order reduction with priority order and partial-order reduction with release order. The average WCRT and 98^{th} percentile only differ slightly (0.00007 and 0.003613 respectively) between the two analyses, while the other values are the same. This shows that the order in which interfering jobs are added to the reduction set barely impacts the response times.

Figures 7.5, 7.6, 7.7, and 7.8 show state-space results for the four different configurations of execution-time variation and release jitter described in Section 7.1.1. These state-space results are the number of states (a, b, c), state-reduction ratio (d, e, f), and POR success ratio (g, h, i) based on all tasks, schedulable tasks, and unschedulable tasks. The averages and percentiles for state-reduction ratio and POR success ratio are very similar for both variations of POR analyses. Table 7.3 shows more state-reduction ratio statistics, including

[1]The gray background indicates that a data point has fewer than 10 samples and as such has a lower confidence.

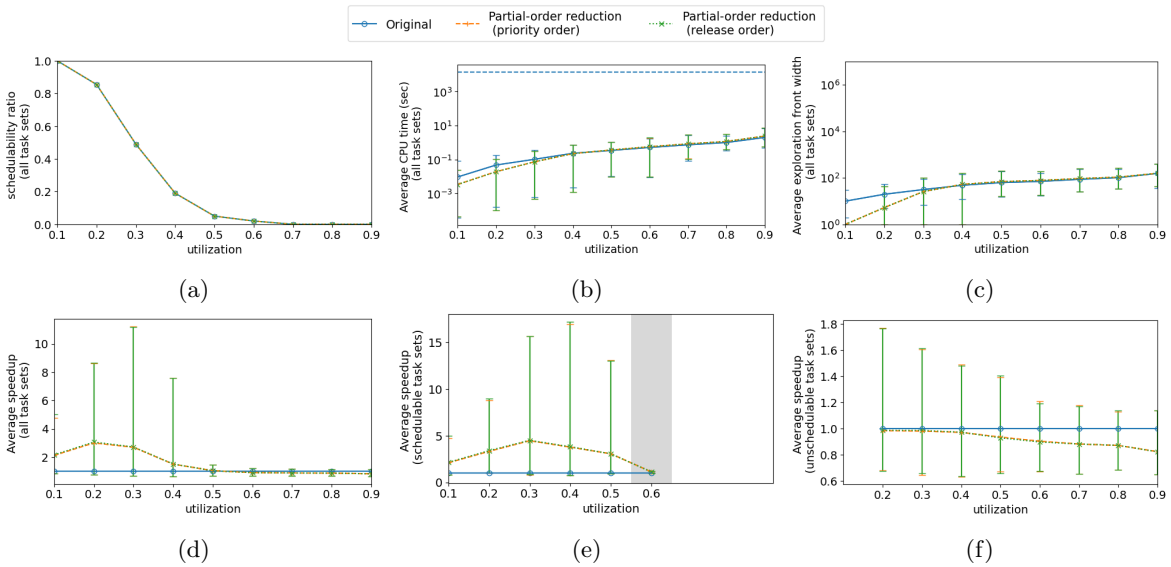


Figure 7.1: **Speedup results for benchmark tasks sets with no execution time variation (i.e., $\beta = 0$). (a, b, c) show the impact of utilization on schedulability, CPU time, and exploration front width. (d, e, f) show the impact of utilization on speedup for all, schedulable, and unschedulable tasks.**

Analysis	Schedulability	Min	2%	Median	Average	98%	Max
POR (priority order)	all	0.362	0.611	0.966	2.085	9.250	451.598
	schedulable	0.362	0.723	1.510	4.220	22.485	451.598
	unschedulable	0.372	0.598	0.880	0.935	1.619	5.630
POR (release order)	all	0.385	0.611	0.968	2.090	9.153	447.058
	schedulable	0.385	0.744	1.533	4.234	23.046	447.058
	unschedulable	0.420	0.598	0.882	0.935	1.630	5.413

Table 7.1: **Speedup statistics for benchmark task sets for both partial-order reduction variants**

minimum and maximum values and percentiles. Again, the differences between priority and release order are minimal.

In conclusion, it appears that between priority and release orders, the order in which interfering jobs are added to the reduction set $\mathcal{J}^M$ does not significantly affect the performance of partial-order reduction, as neither clearly outperforms the other. We have a number of hypotheses for the negligible difference between priority and release order: (i) the highest priority job and earliest released job are often the same job, i.e., both variations of the analysis add jobs in the same order, (ii) multiple interfering jobs need to be added to a reduction set regardless of the order in which they are added (e.g., because multiple jobs are released before some lower priority job in the reduction set), and (iii) some PORs only succeed for priority order while others only succeed for release order, but the number of PORs for which this holds evens out.

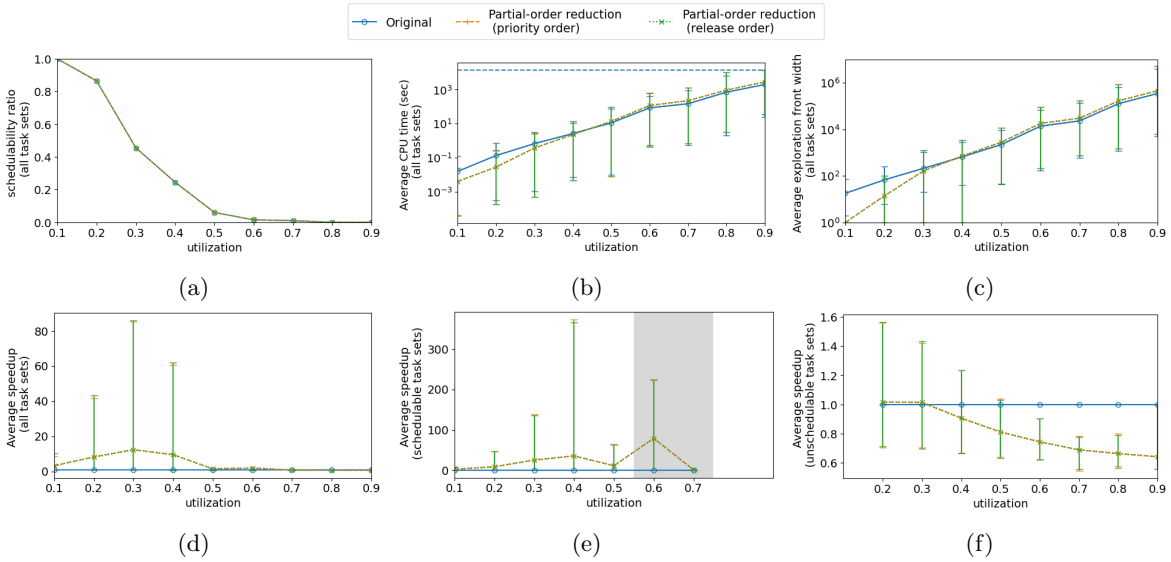


Figure 7.2: **Speedup results for benchmark tasks sets with BCET= 0 (i.e., $\beta = 1$). (a, b, c) show the impact of utilization on schedulability, CPU time, and exploration front width. (d, e, f) show the impact of utilization on speedup for all, schedulable, and unschedulable tasks.**

Analysis	Min	2%	Median	Average	98%	Max
POR (priority order)	1.000	1.000	1.000	1.020	1.195	8.377
POR (release order)	1.000	1.000	1.000	1.020	1.199	8.377

Table 7.2: **Response time statistics for both partial-order reduction variants**

7.1.3 The impact of utilization on state-reduction ratio (EXP1)

In this section, we look at the effect of utilization on the number of successful partial-order reductions and on how this reduces the state space that is explored by the analysis. Our hypothesis is that the state-space explored by the analyses with partial-order reduction is smaller than that of the original algorithm. Moreover, we expect that as the utilization increases, there are more deadline misses thus fewer successful partial-order reductions, meaning that the size of the state-space becomes closer to that of the original analysis.

As Section 7.1.2 showed, the difference between POR with priority order and POR with release order is negligible. Hence, we only focus on the results of the POR with priority order (to keep the figures more readable).

Figures 7.5, 7.6, 7.7, and 7.8, show different values representing the state-space and partial-order reductions as a function of the total utilization for the four different configurations of execution-time variation and release jitter ($\beta = 0$, $\beta = 1$, $\alpha = 0$, and $\alpha = 0.1$ respectively) described in Section 7.1.1. The depicted results are the number of states (a, b, c), state-reduction ratio (d, e, f), and POR success ratio (g, h, i) based on all tasks, schedulable tasks, and unschedulable tasks.

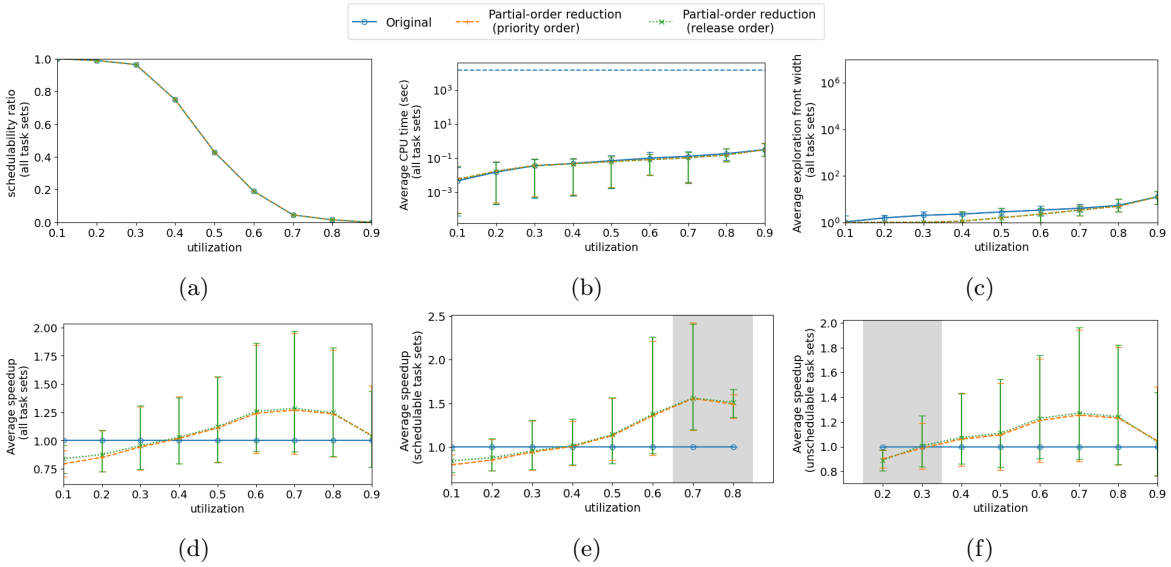


Figure 7.3: **Speedup results for benchmark tasks sets with no jitter (i.e., $\alpha = 0$). (a, b, c) show the impact of utilization on schedulability, CPU time, and exploration front width. (d, e, f) show the impact of utilization on speedup for all, schedulable, and unschedulable tasks.**

Analysis	Schedulability	Min	2%	Median	Average	98%	Max
POR (priority order)	all	−0.002	0.000	0.219	0.284	0.859	0.991
	schedulable	0.000	0.000	0.540	0.446	0.916	0.991
	unschedulable	−0.002	0.007	0.192	0.197	0.465	0.812
POR (release order)	all	−0.002	0.000	0.217	0.283	0.859	0.991
	schedulable	0.000	0.000	0.540	0.446	0.916	0.991
	unschedulable	−0.002	0.007	0.190	0.195	0.461	0.812

Table 7.3: **State-reduction ratio statistics for benchmark task sets for both partial-order reduction variants**

As shown in Figures 7.5a, 7.6a, 7.7a, and 7.8a, the analyses with partial-order reduction have on average a smaller state-space than the original analysis in their schedule-abstraction graph when applied on benchmark task sets. This is shown more clearly by the state reduction ratio in Figures 7.5d, 7.6d, 7.7d, and 7.8d. We computed the average state-reduction ratio across all utilization, jitter, and execution time variation values. On average, both partial-order reduction analyses have a state-space that is 28% smaller than that of the original analysis for benchmark task sets. This confirms our hypothesis that the state-space explored by the analyses with partial-order reduction is smaller than that of the original algorithm. Also note that for the original analysis, the average number of states for $\beta = 1$ and $\alpha = 0.1$ are larger than for $\beta = 0$ and $\alpha = 0$ respectively, i.e., uncertainty in the execution time and release jitter increase the state-space explored by the original algorithm.

The POR success ratio decreases as the utilization increases, as can be observed in Figures 7.5g, 7.6g, 7.7g, and 7.8g. As the utilization increases, more task sets become unschedulable and unschedulable task sets have on average

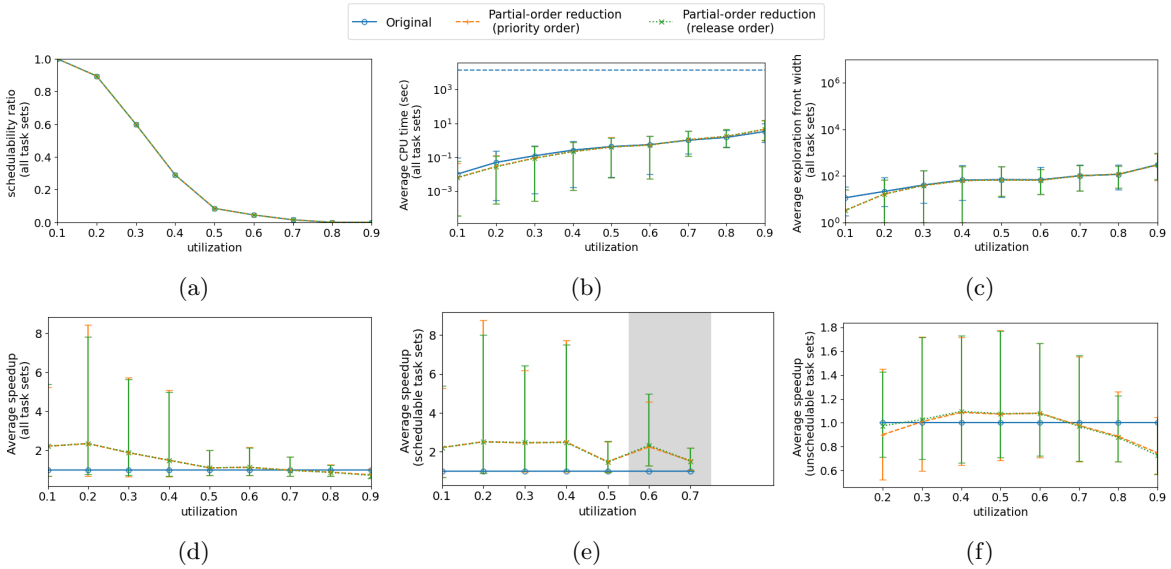


Figure 7.4: **Speedup results for benchmark tasks sets with 10% jitter (i.e., $\alpha = 0.1$). (a, b, c) show the impact of utilization on schedulability, CPU time, and exploration front width. (d, e, f) show the impact of utilization on speedup for all, schedulable, and unschedulable tasks.**

a lower POR success ratio than schedulable task sets, namely 0.42 and 0.95 respectively. However, the POR success ratio also decreases as the utilization increases for schedulable task sets. As the utilization increases, either the reduction sets or the execution times become larger, which causes more uncertainty for the partial-order reduction with regard to deadline misses. And since the partial-order reduction algorithm is pessimistic, there will be more failed reductions instead. This confirms our next hypothesis that the success of POR decreases as the utilization increases.

Figures 7.5a, 7.6a, and 7.8a show that for tasks with jitter, the size of the state-space explored by POR gets closer to that of the original analysis as the utilization increases. This is more apparent in Figures 7.5d, 7.6d, and 7.8d, which show that as the utilization increases, the state reduction ratio decreases. The decrease in state-reduction ratio can be explained by the decreasing number of cases that POR can successfully form a reduction set. This can be seen in Figures 7.5g, 7.6g, and 7.8g.

In contrast, for tasks without jitter, the difference between the state-space explored by POR and the original analysis *increases* with the utilization (Figure 7.7a). The increasing difference is caused by the state-reduction ratio becoming larger as utilization increases (Figure 7.7d). However, we can see in Figure 7.7g that the POR success ratio decreases with the utilization, while the increasing state-reduction ratio would suggest otherwise. This is unlike the tasks with jitter where these trends were the same. Our hypothesis was that the increasing state-reduction ratio could be explained by an increase in the number of jobs in the reduction sets. However, task sets with jitter show the same trend and even have larger reduction sets on average, so this is also not

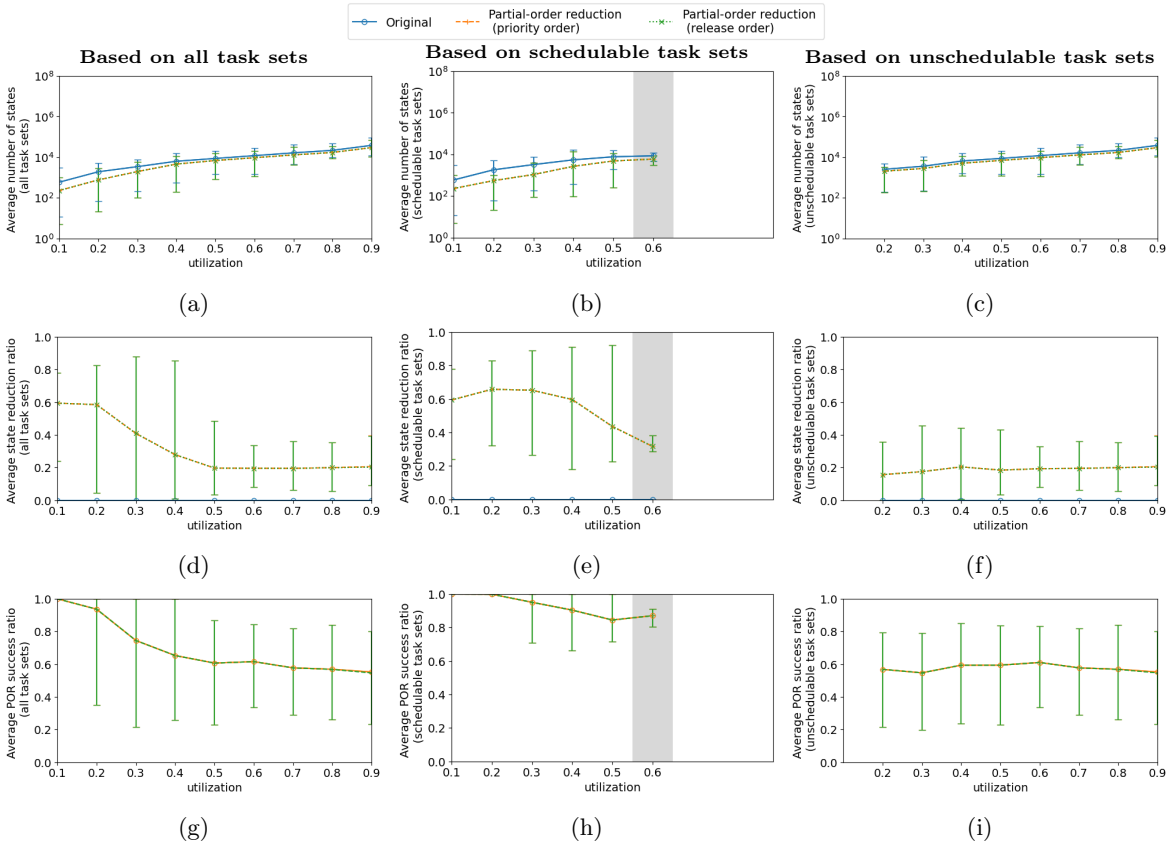


Figure 7.5: **State-space results for benchmark tasks sets with no execution time variation (i.e., $\beta = 0$). (a, b, c) show the impact of utilization on the size of the state-space for all, schedulable, and unschedulable tasks. (d, e, f) show the impact of utilization on state-reduction ratio for all, schedulable, and unschedulable tasks. (g, h, i) show the impact of utilization on the POR success ratio for all, schedulable, and unschedulable tasks.**

an explanation.

Observation 2. The average number of states in both the original and POR analyses is larger for unschedulable task sets than for schedulable task sets.

A task set is unschedulable if at least one job misses its deadline. When a job misses its deadline, some workload of this job is carried over beyond its deadline. This can create an area of variable blocking time (by tardy jobs) for the not-yet-processed jobs. The variable blocking time caused by the carry-in jobs is analogous with *release jitter* for the jobs that are affected by the carry-in workload. Consequently, the number of branches in the graph increases due to the increase in the number of jobs that can be scheduled next. This might be the cause of a general increase in the number of states for unschedulable task sets compared to schedulable task sets.

Consider the job set in Figure 7.9. If J_1 always finishes before its deadline, the execution order of J_2 and J_0 is deterministic as J_1 finishes before the release

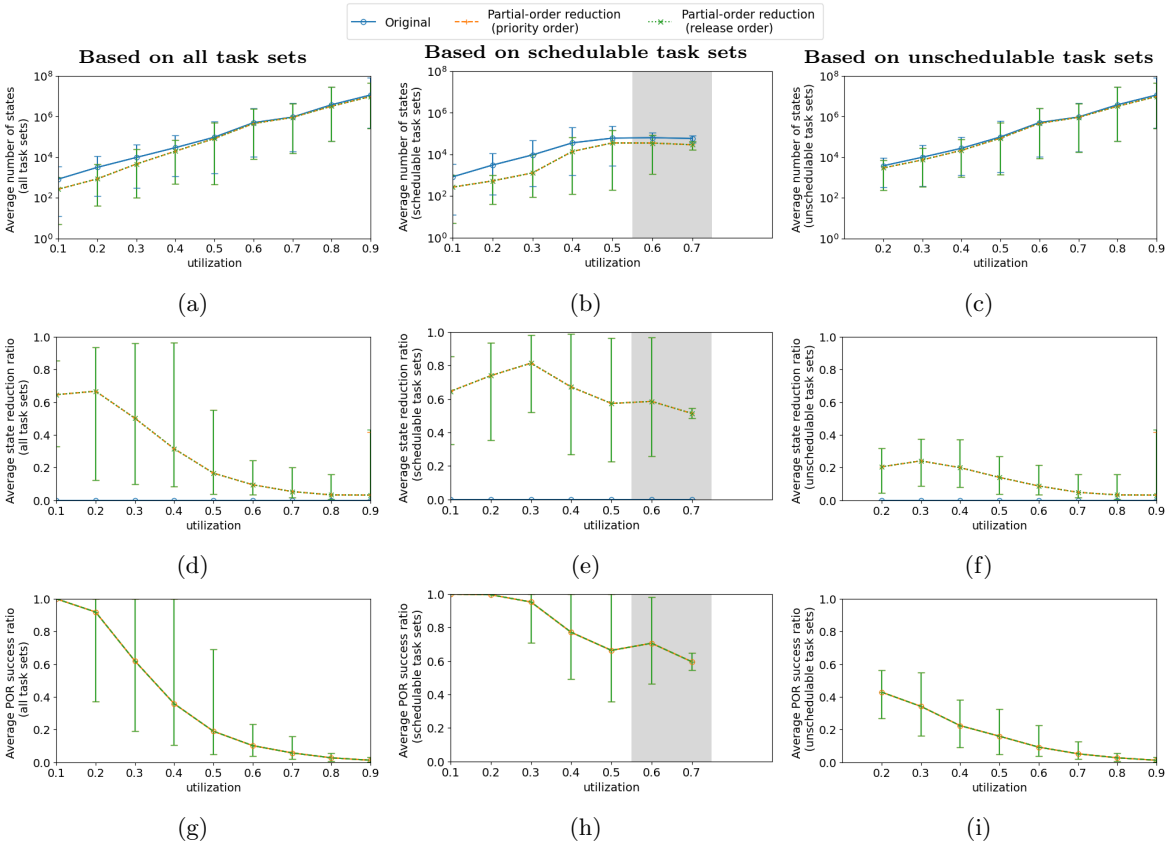


Figure 7.6: **State-space results for benchmark tasks sets with BCET= 0 (i.e., $\beta = 1$). (a, b, c) show the impact of utilization on the size of the state-space for all, schedulable, and unschedulable tasks. (d, e, f) show the impact of utilization on state-reduction ratio for all, schedulable, and unschedulable tasks. (g, h, i) show the impact of utilization on the POR success ratio for all, schedulable, and unschedulable tasks.**

of J_2, so J_2 will always is dispatched before J_0. However, if J_1 is able to miss its deadline as in the right schedule, it creates a different scenario where J_0 executes before J_2. If J_1 can either finish before or after its deadline, there are two execution scenarios instead of one.

Looking at the average number of states per graph, we can see that for task sets with jitter the difference between the original analysis and POR is larger for schedulable task sets (Figures 7.5b, 7.6b, 7.8b) than for unschedulable task sets (Figures 7.5c, 7.6c, 7.8c). This becomes even more apparent when looking at the difference in state-reduction ratio between schedulable (Figures 7.5e, 7.6e, 7.8e) and unschedulable (7.5f, 7.6f, 7.8f) task sets.

More specifically, the average state-reduction ratio is 65% for schedulable task sets and 18% for unschedulable task sets. We computed the average state-reduction ratio by taking the average state-reduction ratio across all utilizations and all tasks with jitter > 0 (i.e., for $\beta = 0$, $\beta = 1$, and $\alpha = 0.1$) separated by schedulability. This difference in state-reduction ratio might be attributed to

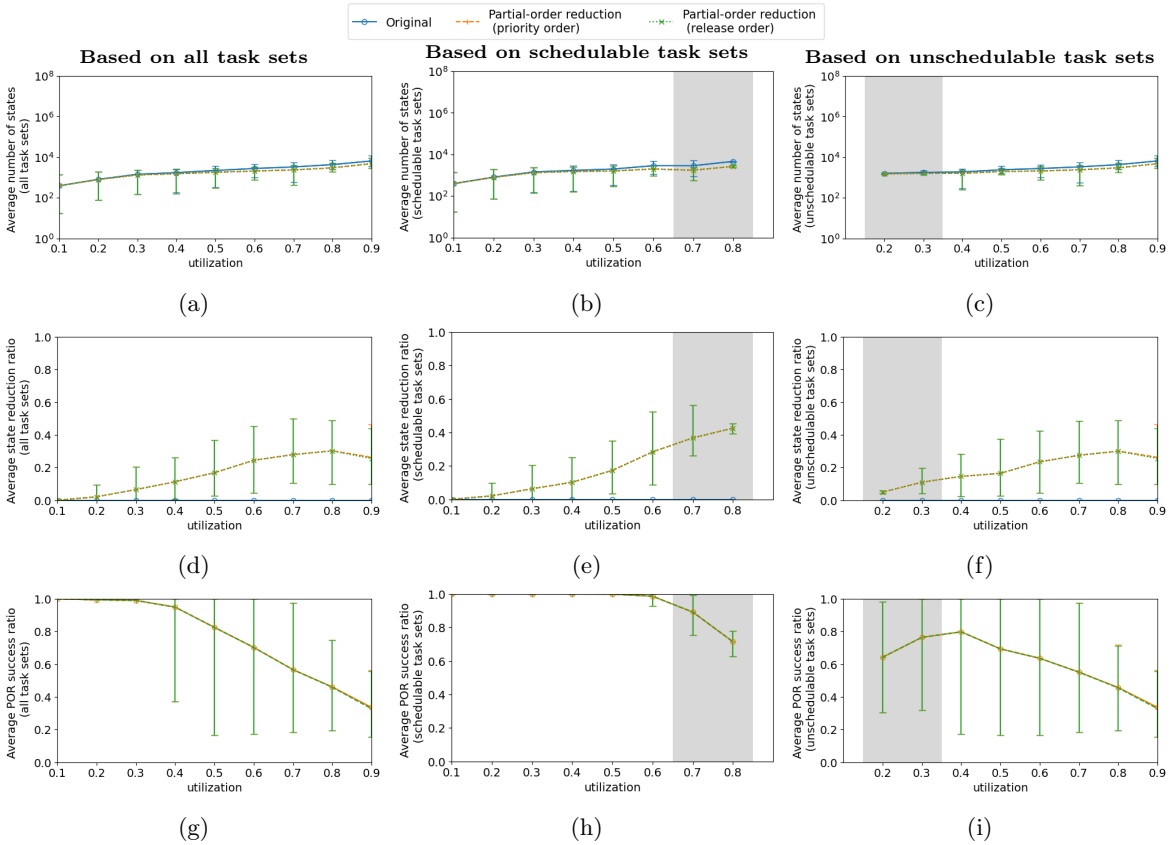


Figure 7.7: **State-space results for benchmark tasks sets with no jitter (i.e., $\alpha = 0$). (a, b, c) show the impact of utilization on the size of the state-space for all, schedulable, and unschedulable tasks. (d, e, f) show the impact of utilization on state-reduction ratio for all, schedulable, and unschedulable tasks. (g, h, i) show the impact of utilization on the POR success ratio for all, schedulable, and unschedulable tasks.**

the difference in POR success ratio, which is much higher for schedulable task sets (Figures 7.5h, 7.6h, 7.8h) than for unschedulable ones (Figures 7.5i, 7.6i, 7.8i), namely 0.93 and 0.39 respectively.

Contrarily, task sets without jitter (Figure 7.7) have a larger state-reduction ratio for unschedulable task sets (25%) than for schedulable ones (7%). However, the POR success ratio is larger for schedulable task sets, so this does not explain the difference in state-reduction ratio for tasks without jitter. Our next hypothesis was that the difference might be caused by the reduction set size, with the unschedulable task sets having larger reduction sets, but this also was not the case.

Observation 3. Unschedulable task sets have a lower POR success ratio, i.e., a lower ratio of PORs that satisfy the conditions of Definitions 6 and 7, than schedulable task sets.

On average, schedulable task sets have a POR success ratio of 0.95, whereas

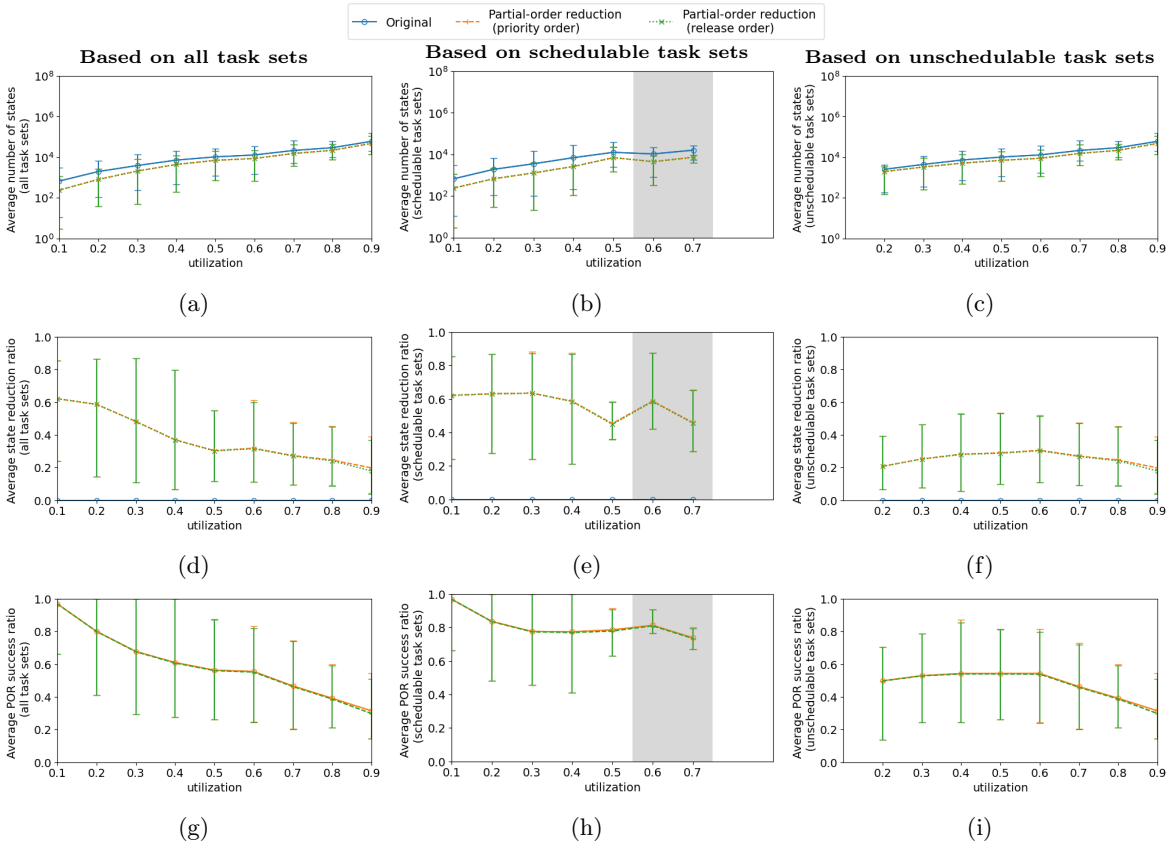


Figure 7.8: **State-space results for benchmark tasks sets with 10% jitter (i.e., $\alpha = 0.1$). (a, b, c) show the impact of utilization on the size of the state-space for all, schedulable, and unschedulable tasks. (d, e, f) show the impact of utilization on state-reduction ratio for all, schedulable, and unschedulable tasks. (g, h, i) show the impact of utilization on the POR success ratio for all, schedulable, and unschedulable tasks.**

unschedulable task sets have a POR success ratio of 0.42. As previously explained, when a job misses its deadline the workload that is carried over beyond its deadline may cause a blocking time for the next jobs that are not yet included in the graph. As this results in more branches, the set of jobs considered for partial-order reduction (i.e., the reduction set) also becomes larger, which in turn may result in a potential deadline miss being detected. Hence, the POR phase fails to form the reduction sets. Moreover, a deadline miss can also chain to other deadline misses, since a job missing its deadline can push other jobs to miss their deadline as well. Because the partial-order reduction is pessimistic, the reduction fails if there is at least one such potential deadline miss. Hence, if a task set is unschedulable it will reject partial-order reduction more often.

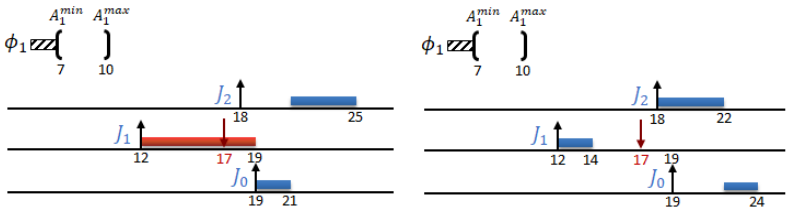


Figure 7.9: $\mathcal{J} = \{J_0, J_1, J_2\}$ **with priority assignment** $p_0 < p_1 < p_2$**. The execution time range for** J_0 **is** $[1, 2]$**, for** J_1 **it is** $[2, 7]$**, and for** J_2 **it is** $[3, 4]$**.**

7.1.4 The impact of utilization on speedup (EXP1)

In this section, we look at the effect of utilization on the performance of partial-order reduction in terms of speedup. Our hypothesis is that as the utilization increases, there are more deadline misses thus fewer successful partial-order reductions and therefore a smaller speedup.

As Section 7.1.2 showed that the difference between POR with priority order and POR with release order is negligible. Hence, we only focus on the results of the POR with priority order.

Figures 7.1 7.2, 7.3, and 7.4 show speedup results for the four different configurations of execution-time variation and release jitter described in Section 7.1.1. Depicted are the schedulability ratio (a), CPU time or runtime (b), and exploration front width (c) based on all tasks, and the speedup based on all tasks, schedulable tasks, and unschedulable tasks (d, e, and f respectively).

Observe that the schedulability ratio is the same for all analyses, as the POR algorithm (Algorithm 4) preserves the schedulability exactness of the original analysis (Lemma 16).

Figures 7.1b, 7.2b, 7.3b, and 7.4b show that the average CPU time increases as the utilization increases. This can be explained by the increase in exploration front width (Figures 7.1c, 7.2c, 7.3c, 7.4c) and state-space (Figures 7.5a, 7.6a, 7.7a, 7.8a).

The average speedup over all utilization, jitter, and execution time variation values (i.e., the average speedup of Figures 7.1d, 7.2d, 7.3d, and 7.4d combined) is equal to 2.1, so on average the POR algorithm is faster than the original algorithm.

Observe in Figures 7.1d, 7.2d, and 7.4d that the average speedup for POR is larger than 1 for utilizations between 0.1 and 0.5, i.e., POR is on average faster than the original analysis. The speedup becomes very close to 1 and even slightly smaller than 1 from 0.6 utilization onward. On average, the speedup for task sets with jitter is 2.43. An explanation for this decreasing speedup might be the average state-space for partial-order reduction approaching that of the original analysis, which in turn is a result of the decreasing POR success ratio as explained previously in Section 7.1.3.

For tasks without jitter (Figure 7.3d) we see the opposite, namely that the speedup is smaller than 1 between 0.1 and 0.3 utilization, while it is larger than 1 from 0.4 utilization onward. This might be attributed to the fact that the state-reduction ratio increases as utilization increases for task sets without jitter, in contrast to the task sets with jitter where the state-reduction ratio decreases instead (Section 7.1.3). On average, the speedup for task sets without jitter

is 1.06. The reason for the speedup being so small for low utilizations might be the following. If there is no jitter, the sequence of tasks is deterministic for low utilizations. Even though there is execution time variation, this only causes branching for high utilizations. In Figure 7.3c we can see that for $U \leq 0.4$, the maximum width of the graph is ranges from 1 to 2 for the original algorithm, meaning that the graphs for low utilizations often only have one or two branches. Partial-order reduction only has a benefit when there are a lot of branches to be potentially reduced, so in cases where the graph only has one or two branches, all the partial-order reduction algorithm does is adding overhead. Hence, partial-order reduction is slower than the original algorithm in these cases.

Observe that for task sets with jitter, the speedup of schedulable task sets (Figures 7.1e, 7.2e, 7.4e) is larger than for unschedulable task sets (Figures 7.1f, 7.2f, 7.4f). For schedulable task sets the average speedup across all utilizations is 5.98, while for unschedulable task sets it is 0.88. An explanation for this is the difference between the state-reduction ratio for schedulable and unschedulable task sets, which is on average larger for schedulable task sets than for unschedulable ones, as shown in Section 7.1.3.

Again, the task sets without jitter depict a different scenario for schedulable and unschedulable task sets (Figures 7.3e and 7.3f). Here, the average speedup for unschedulable task sets (1.16) is larger than for schedulable task sets (0.94). Like for the task sets with jitter, we can attribute this to the difference between the state-reduction ratio for schedulable and unschedulable task sets, which in this case is larger for unschedulable task sets. Note that the POR success ratio for unschedulable task sets (Figure 7.7i) is higher than in the other jitter and execution time variation configurations (Figures 7.5i, 7.6i, and 7.8i). Recall from Observations 2 and 3 that a deadline miss can introduce additional "jitter" that can hinder the success of partial-order reductions. Perhaps because these task sets do not have release jitter, the "jitter" introduced by deadline misses is by itself not enough to impede the performance of partial-order reduction. Also note that the average number of states for schedulable (Figure 7.7b) and unschedulable task sets (Figure 7.7c) is not too different, in contrast to the other configurations.

7.1.5 The impact of partial-order reduction on WCRT (EXP1)

In this section, we determine the effect of partial-order reduction on the accuracy of response times, as well as how this effect changes for different parameters. Our hypothesis is that partial-order reduction reports larger worst-case response times than the original analysis and the more successful partial-order reductions there are, the larger the error becomes. There should be no response times that are smaller than reported by the original analysis.

As Section 7.1.2 showed that the difference between POR with priority order and POR with release order is negligible, the numbers mentioned in the text will only be that of POR with priority order.

Table 7.4 shows the statistics of the relative WCRT by execution time variation.We see that the average and max relative WCRT is slightly larger for the largest execution time variation ($\beta = 1$), but the other values remain the same, namely 1. So, execution time variation has no effect on the relative WCRT for the 98^{th} percentile of task sets.

Observation 4. Utilization has a larger effect on relative WCRT for tasks with large jitter than for tasks with small or no jitter.

Table 7.5 shows the statistics of the relative WCRT by utilization for each value of jitter for partial-order reduction using priority order. Observe that for small jitters (i.e., no jitter and a jitter of 100µs), all values apart from the max are very close to 1 for all utilizations. For the large jitter ($0.1 \cdot (T_i - C_i)$) on the other hand, the relative WCRT is larger and there is a trend where as the utilization decreases, the relative WCRT increases. A visual representation of the trend for large jitter is provided in Figure 7.10a using box plots, where the whiskers of the box plots represent the 2^{nd} and 98^{th} percentiles.

The increasing relative WCRT as the utilization decreases can partially be explained by the fact that the lower the utilization, the more successful reductions are performed. And the more successful reductions, the more relative error in the WCRT the analysis introduces, because partial-order reductions over-approximate the WCRT of jobs. However, this does not explain why we only see this trend for larger jitters and not for very small ones.

Table 7.6 shows the statistics of the relative WCRT by priority family for each value of jitter for partial-order reduction using priority order. A visual representation of the relative WCRT statistics for large jitter is provided in Figure 7.10b using box plots.

For each task set, tasks are divided into different priority families because we cannot simply accumulate tasks by their priority level, as priorities are relative. For example, a priority of 4 is low for a task set with 5 tasks, but high for a task set with 25 tasks. In order of descending priority, the first 25% of tasks are assigned to the high category, the next 25% to medium-high, the following 25% to medium-low, and the final 25% to low.

We can see that for the small jitters, only high priority tasks have a slight error of about 5% at the 98^{th} percentile. Also, only high priority tasks have a larger average error than the other priorities. The errors for lower priority tasks are very small. Even the max values are 1 for medium-low and low priority tasks when there is no jitter, and for all except high priority tasks for a jitter of 100µs. Looking at the large jitter however, the averages, 98^{th} percentiles and maximum values are again larger, and we see a trend where the error is largest for high priority tasks and becomes lower as the priority decreases.

One aspect explaining the larger errors for the high priority tasks for all jitters is that with the over-approximated latest start time $\widehat{LST}_i(\mathcal{J}^M, v_p)$ we may consider scenarios that cannot actually happen to a job. The period for high priority tasks is so small that by doing this we over-approximate the response time by almost its entire period. Another explanation is that far more partial-order reductions are performed for high priority tasks in comparison to lower priority tasks, as shown in Figure 7.11. As the lower priority jobs are relatively rarely involved in successful reductions, their response-times are of course also less affected.

Observation 5. Despite its overapproximation of the WCRT, the POR analysis still reports the same schedulability as the original analysis for all task sets.

A partial-order reduction is only accepted when there certainly is no deadline miss, and in all other cases it is rejected so that the analysis can find the exact response-time bounds.

Execution time variation	Min	2%	Median	Average	98%	Max
0	1.000	1.000	1.000	1.000	1.000	1.264
1	1.000	1.000	1.000	1.001	1.000	1.528

Table 7.4: **Response time statistics for benchmark task sets by execution time variation for partial-order reduction with priority order**

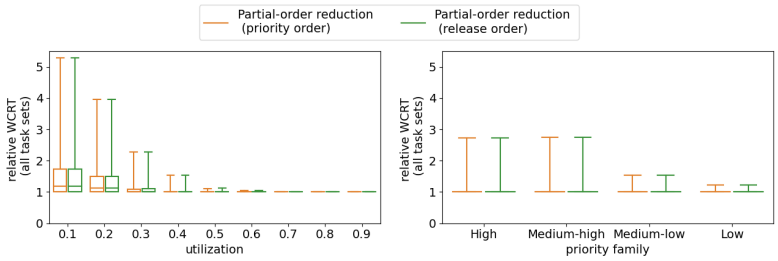
Jitter	Utilization	Min	2%	Median	Average	98%	Max
0	0.1	1.000	1.000	1.000	1.000	1.000	1.000
	0.2	1.000	1.000	1.000	1.000	1.000	1.000
	0.3	1.000	1.000	1.000	1.000	1.000	1.000
	0.4	1.000	1.000	1.000	1.000	1.000	1.073
	0.5	1.000	1.000	1.000	1.003	1.000	1.523
	0.6	1.000	1.000	1.000	1.006	1.069	1.899
	0.7	1.000	1.000	1.000	1.003	1.000	1.880
	0.8	1.000	1.000	1.000	1.001	1.000	2.073
	0.9	1.000	1.000	1.000	1.000	1.000	1.261
100µs	0.1	1.000	1.000	1.000	1.000	1.000	1.000
	0.2	1.000	1.000	1.000	1.002	1.024	1.209
	0.3	1.000	1.000	1.000	1.003	1.067	1.261
	0.4	1.000	1.000	1.000	1.003	1.016	1.528
	0.5	1.000	1.000	1.000	1.001	1.000	1.276
	0.6	1.000	1.000	1.000	1.000	1.000	1.173
	0.7	1.000	1.000	1.000	1.000	1.000	1.348
	0.8	1.000	1.000	1.000	1.000	1.000	1.153
	0.9	1.000	1.000	1.000	1.000	1.000	1.038
$0.1 \cdot (T_i - C_i)$	0.1	1.000	1.000	1.191	1.674	5.284	8.377
	0.2	1.000	1.000	1.135	1.414	3.952	5.759
	0.3	1.000	1.000	1.000	1.118	2.285	3.644
	0.4	1.000	1.000	1.000	1.041	1.546	2.237
	0.5	1.000	1.000	1.000	1.006	1.110	2.111
	0.6	1.000	1.000	1.000	1.004	1.045	1.878
	0.7	1.000	1.000	1.000	1.001	1.001	1.451
	0.8	1.000	1.000	1.000	1.000	1.000	1.076
	0.9	1.000	1.000	1.000	1.000	1.000	1.053

Table 7.5: **Response time statistics for benchmark task sets by utilization for partial-order reduction with priority order**

7.1.6 The effect of release jitter and execution time variation on the speedup (EXP2)

In this section, we look at the effect of release jitter and execution variation on the speedup of partial-order reduction over the original analysis. Our hypothesis is that for large jitters, there are fewer successful partial-order reductions resulting in a smaller speedup, and that for large execution time variations the speedup keeps increasing. For both jitter and execution time variation we saw in Sections 7.1.3 and 7.1.4 that as jitter and execution time variation increased, the state-reduction ratio and speedup both increased. But for jitter we expect to reach a tipping point where the jitter becomes too large and starts to impede the partial-order reduction.

In contrast to the previous experiments, we performed a schedulability analysis, i.e., we stopped the analysis as soon as a deadline miss was encountered. As we found in Section 7.1.4 that the largest average speedup came from the schedulable task sets and not the unschedulable ones, we decided to focus on the speedup of the schedulable task sets. Therefore, it is fine to stop analyzing the unschedulable task sets as soon as a deadline miss is encountered. Also, task sets that timed out were not removed from the results as time outs are still



(a) **WCRT by utilization** (b) **WCRT by priority family**

Figure 7.10: **Boxplots of WCRT for benchmark tasks sets with 10% jitter (i.e., $\alpha = 0.1$)**

Jitter	Priority	Min	2%	Median	Average	98%	Max
0	High	1.000	1.000	1.000	1.006	1.073	2.073
	Medium-high	1.000	1.000	1.000	1.000	1.000	1.548
	Medium-low	1.000	1.000	1.000	1.000	1.000	1.000
	Low	1.000	1.000	1.000	1.000	1.000	1.000
100µs	High	1.000	1.000	1.000	1.003	1.044	1.528
	Medium-high	1.000	1.000	1.000	1.000	1.000	1.000
	Medium-low	1.000	1.000	1.000	1.000	1.000	1.000
	Low	1.000	1.000	1.000	1.000	1.000	1.000
$0.1 \cdot (T_i - C_i)$	High	1.000	1.000	1.000	1.127	2.733	8.313
	Medium-high	1.000	1.000	1.000	1.106	2.740	8.377
	Medium-low	1.000	1.000	1.000	1.045	1.528	5.268
	Low	1.000	1.000	1.000	1.014	1.231	2.209

Table 7.6: **Response time statistics for benchmark task sets by priority for partial-order reduction with priority order**

relevant when looking at speedup only.

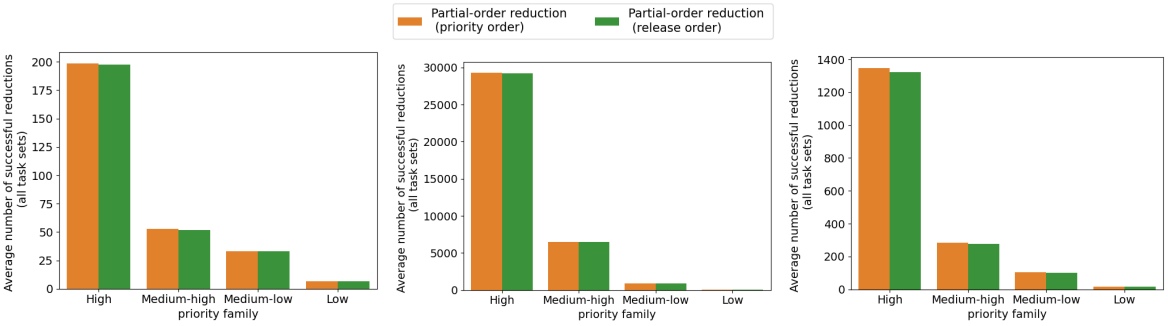
Figure 7.12a shows the average speedup as a function of execution time variation for schedulable task sets. Observe that the average speedup increases as the execution time variation increases. As expected following the results from Section 7.1.4, the speedup for unschedulable task sets is < 1 (Figure 7.12b).

Figure 7.12c shows the average speedup as a function of release jitter for schedulable task sets. The speedup peaks at $\alpha = 0.05$ for both POR with priority order and release order. After this peak the speedup decreases but remains > 1 until $\alpha = 0.50$. An explanation for the decreasing speedup as the jitter increases is that a larger jitter might cover the release of more jobs. In turn, more jobs might interfere with each other, creating more opportunities for a POR to be rejected. Again, Figure 7.12d shows that the speedup for unschedulable task sets is < 1 except for task sets without jitter, as explained in Section 7.1.4.

7.2 Experiment on synthetic task sets

7.2.1 Task set generation

In this experiment, task sets were generated according to the technique by Emberson et al. [12] for generating synthetic task sets. They describe period distribution and execution time statistics that are representative of commercial



(a) **Jitter = 0** (b) **Jitter = 100µs** (c) **Jitter** $= 0.1 \cdot (T_i - C_i)$

Figure 7.11: **Plots of the number of successful partial-order reductions for benchmark task sets by priority family and by jitter**

systems. We generate a task set with n tasks by randomly drawing n period values from a log-uniform distribution in the range $[10, 100]$ ms with a granularity of 5ms. Next, we generate n task-utilization values that sum to target utilization U using the RandFixedSum algorithm [23]. The periods and task-utilization values are combined to obtain C_i^{max} for each task.

Design of experiment

This section describes the parameters that were used for the experiments on synthetic task sets. The type of analysis used and the way that timed out task sets are handled is common for both experiments.

Type of analysis. For both experiments, we continued the analysis until either the end of the hyperperiod or the timeout of four hours was reached, even after finding a scenario containing a deadline miss.

Experiment 3 (EXP3) The purpose of this experiment is to evaluate the impact of the number of tasks on the state-reduction ratio, speedup, and WCRT of the POR analysis. The results of this experiment will be discussed in Section 7.2.2.

Task set parameters. The number of tasks n per task set was chosen from $\{5, 10, 15, 20, 25, 30, 35\}$. For each n we generated 200 task sets. We set C_i^{min} to 0, the jitter to 100µs, and the utilization to 0.3.

Jobs per hyperperiod. Task sets with more than 50.000 jobs per hyperperiod were discarded to limit the runtimes.

Handling timeouts. Task sets that reached the timeout of four hours were included in all results except for the relative WCRT to demonstrate the scalability improvement provided by POR. Timed out task sets were removed from the relative WCRT results because it is unknown whether the analysis has explored the scenario that leads to the WCRT of a job. If not, the analysis will report an underestimation on the WCRT which we cannot use for comparison.

Experiment 4 (EXP4) The purpose of this experiment is to evaluate the scalability of the POR analysis. We only evaluate the scalability of POR with priority order. The original algorithm is not included as we will go beyond

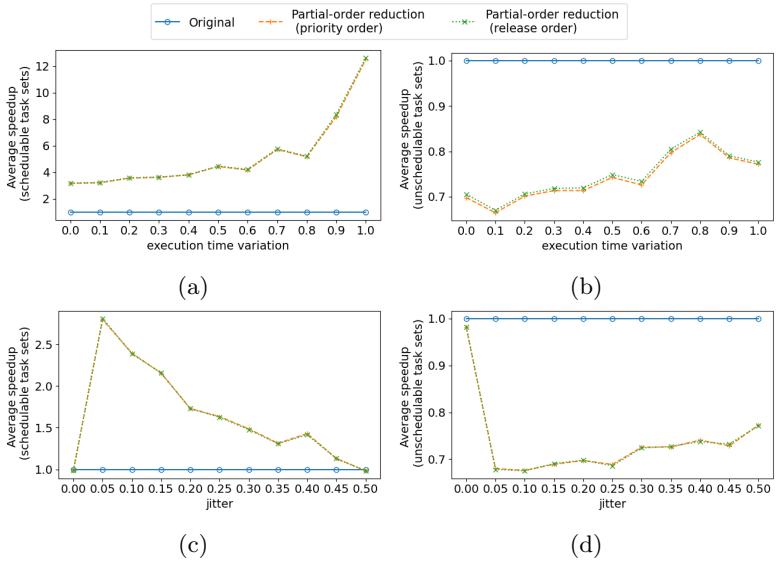


Figure 7.12: **Experimental results for benchmark task sets for speedup by execution time variation and release jitter. (a) and (b) show the speedup by execution time variation for schedulable and unschedulable task sets respectively. (c) and (d) show the speedup by release jitter for schedulable and unschedulable task sets respectively.**

the number of jobs per hyperperiod that the original algorithm can explore within four hours. As Section 7.1.2 showed that the difference between POR with priority order and POR with release order is negligible, we decided to not include POR with release order in this experiment. The results of this experiment will be discussed in Section 7.2.3.

Task set parameters. The number of tasks n per task set was chosen from $\{10, 20, 30, 40, 50, 60, 70\}$. For each n we generated 200 task sets. We set C_i^{min} to 0, the jitter to 100µs, and the utilization to 0.3.

Jobs per hyperperiod. Task sets with more than 6.000.000 jobs per hyperperiod were discarded.

Handling timeouts. Task sets that reached the timeout of four hours were included in all results as we don't make any comparison between analyses.

7.2.2 The impact of the number of tasks on the performance of POR (EXP3)

In this section, we look at the impact of the number of tasks per task set on the performance of the POR analyses in terms of speedup, state-reduction ratio, and relative WCRT.

Figure 7.13 shows different values as a function of the number of tasks per task set. The depicted results are the schedulability ratio (a), number of states (b), CPU time (c), the ratio of timed out task sets (d), state-reduction ratio (e), speedup (f), POR success ratio (g), and relative WCRT (h).

Observation 6. POR has a higher schedulability ratio than the original algorithm for $n \geq 15$.

The POR analyses have a schedulability ratio close to 1 for all n, while the schedulability ratio quickly drops to 0 for the original analysis as n increases (Figure 7.13a). Observe that this decrease in schedulability corresponds to the ratio of timeouts (Figure 7.13d) that the original analysis suffers. The task sets are deemed unschedulable by the original analysis because the analysis timed out after four hours, while the POR analyses were able to explore the entire state-space and conclude the task sets were schedulable within the time limit. This shows that the POR allows for more complex task sets to be analyzed before the timeout is reached.

Figure 7.13e shows that the state-reduction ratio increases as n increases. While the number of states explored by POR remains rather constant for all values of n (with the exception of a peak at $n = 20$ caused by an outlier), the size of the state-space explored by the original analysis keeps growing as n increases (as can be seen in Figure 7.13b). As a result, the state-reduction ratio for POR keeps increasing and approaches 1. The average state-reduction ratio is 98.53%.

Observation 7. The average speedup increases as n increases until $n = 25$, after which the speedup decreases slightly.

We can observe in Figure 7.13c that the runtime of the original analysis increases as n increases, while the runtime of POR is significantly lower and increases more slowly and even decreases from $n = 20$ to $n = 25$. This together with the increasing state-reduction ratio results in an increasing speedup. The slight speedup decrease for $n \geq 30$ is caused by the runtime for POR increasing while the runtime for the original analysis remains constant (namely four hours). The average speedup is $1.12 \cdot 10^5$.

Observation 8. The speedup for log-uniform task sets is orders of magnitude larger than the speedup for automotive task sets.

The average speedup for automotive tasks is about 2.08, whereas it is $1.12 \cdot 10^5$ for log-uniform tasks. Note that most of the log-uniform task sets were schedulable, and that the average for automotive also includes all utilizations and execution time variations. If we apply the same restrictions to the automotive tasks we get an average speedup of about 25.95, which is still three orders of magnitude smaller than the speedup for log-uniform tasks.

The difference is between the automotive and log-uniform task sets is their period distribution. Automotive task sets have semi harmonic periods, whereas log-uniform task sets have random periods. This randomness causes the hyperperiods for log-uniform task sets to quickly become very large (e.g. 50.000+), while they stay relatively small for automotive task sets (e.g. up to 5000). Also, automotive task sets have a smaller state-space to explore due to the semi harmonic periods. Because the tasks are (periodic and) synchronous, a lot of jobs arrive at time 0 where they will all interfere with each other in case there is release jitter. So, at the start there will be a number of branches for the different decisions that can be made. However, after passing the area of release jitter, the job order becomes deterministic (i.e., the SAG becomes a line) until the next set of jobs arrives. At that point, the same thing happens because of the semi harmonic nature of the periods. In contrast, log-uniform tasks will not have the

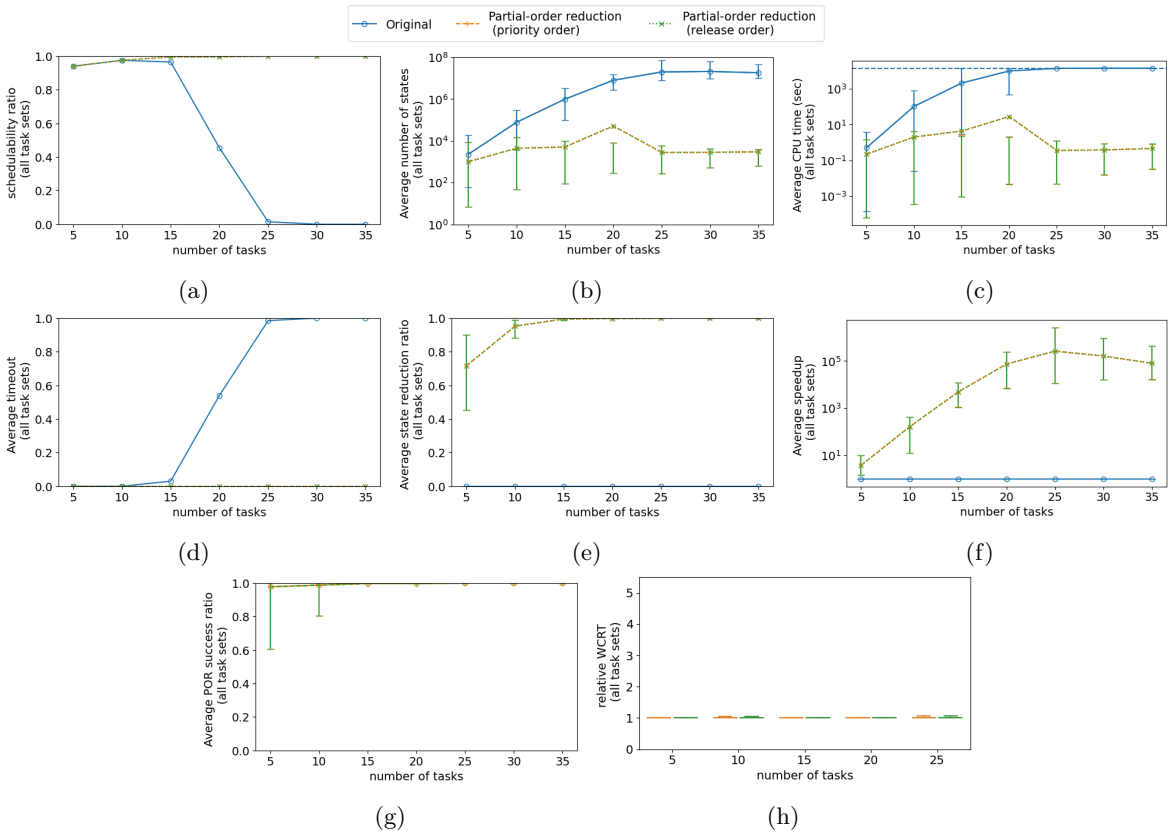


Figure 7.13: **Results for synthetic task sets. (a, b, c) show the impact of the number of tasks on schedulability, state-space size, and CPU time. (d, e, f) show the impact of the number of tasks on timeouts, state-reduction ratio, and speedup. (g, h) show the impact of the number of tasks on the POR success ratio and relative WCRT.**

same behavior after the initial release at time 0. As the periods are random, at any point in time there may be a job release that can cause more branching. As a result, the state-space for log-uniform tasks can easily grow beyond that of automotive tasks. Since the state-space of automotive task sets is already rather small, the potential performance gain of POR is also relatively small and hence the speedup is lower. Conversely, the state-space for log-uniform tasks can become huge which also increases the potential for POR to achieve a large performance gain.

Figure 7.13h shows the relative WCRT for task sets for which none of the analyses timed out. Observe that the relative WCRT is very close to 1 for all values of n, with an average of 1.001.

7.2.3 The scalability of partial-order reduction (EXP4)

In this section, we evaluate the scalability of the POR analysis in terms of the number of tasks per task set and number of jobs per hyperperiod that the analysis is able to analyze before timing out on all job sets.

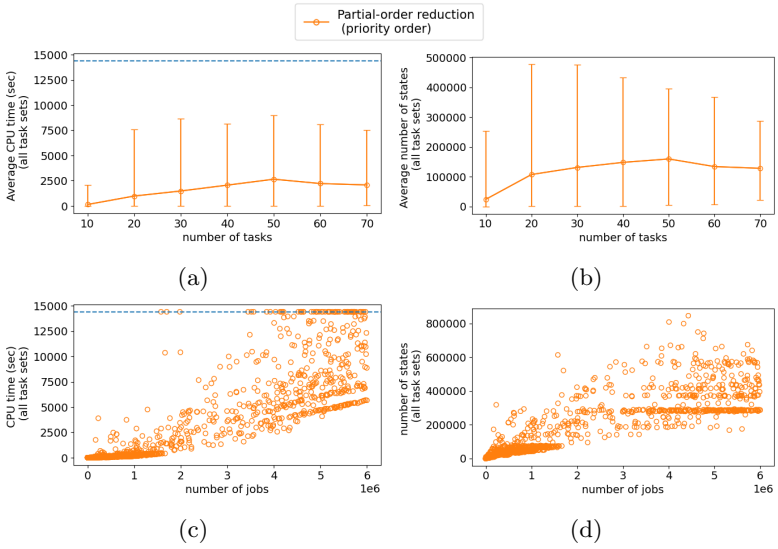


Figure 7.14: **Results for the scalability experiment for synthetic task sets. (a, b) show the impact of the number of tasks on CPU time and state-space size. (c, d) show the impact of the number jobs per hyperperiod on CPU time and state-space size.**

Figures 7.14a and 7.14b show the average runtime and number of states in the state-space plotted by the number of tasks per task set. To reduce the effect of outliers on the averages, we only included task sets with runtimes in the 90^{th} percentile. As a result, we can see the trend for the number of tasks more clearly. Observe in Figure 7.14a that the average runtime increases linearly with the number of tasks, with a slight decrease for $n \geq 60$. The same trend holds for the number of states in Figure 7.14b. This decrease for $n \geq 60$ might be an artifact of the distribution of the number of jobs per hyperperiod.

Figures 7.14c and 7.14d show a scatter plot of the runtime and number of states in the state-space by the number of jobs per hyperperiod. In these figures we included all task sets as we are not plotting averages. Observe that both runtime and number of states appear to have a linear relation with the number of jobs per hyperperiod. Even though there are some timeouts between $1.5 \cdot 10^6$ and $6 \cdot 10^6$ jobs, the vast majority of task sets are analyzed within four hours. This shows that the POR analysis is able to scale to 70 tasks per task set and $6 \cdot 10^6$ per hyperperiod with the potential to scale even further.

Chapter 8

Conclusions and future work

8.1 Summary of contributions

We have improved the scalability of the schedule-abstraction-based analysis by introducing partial-order reduction (POR) rules that allow combining multiple scheduling decisions on one edge and hence avoiding combinatorial exploration of all possible orderings between jobs in cases where there are large uncertainties without jeopardizing the soundness of the analysis and without making it more pessimistic.

The key idea of our POR technique is to identify subsets of jobs for which the combinatorial exploration of all orderings is *irrelevant* to the schedulability of the job set. Exploring these combinations is irrelevant when none of these scenarios lead to a deadline miss. Our POR rules allow the dispatch of such jobs to be considered in a single step (that combines all those scheduling decisions), which further defers the state-space explosion.

Our solution is an *exact schedulability analysis* and a *safe response-time analysis* that reduces the size of the graph while only introducing a small overestimation on the WCRT as shown by our experiments.

8.2 Conclusions

Coming back to the research questions RQ1 to RQ3 defined in Section 1.2, in this dissertation we showed that

RQ1 The execution order of a set of jobs (called the candidate reduction set) does not contribute to a violation of timing constraints if Lemma 15 holds for that set of jobs (see Section 6.5).

RQ2 Our POR maintains the exactness of the schedulability analysis (Lemma 16) and provides safe response-time bounds (Lemma 17) by constructing reduction sets (see Definition 7) using Algorithm 3. We also provide solutions to compute tight lower and upper bounds on the finish time of the reduction set (Corollary 1 and 2) without having to explore each job execution ordering.

RQ3 The POR rules (derived in Chapter 6) can be applied during the expansion phase of the original schedule-abstraction analysis as shown in Algorithm 4.

On average, our solution was able to reduce the runtime in half for benchmark task sets and by five orders of magnitude for synthetic task sets. Furthermore, it reduced the number of explored states by 28% for benchmark tasks and over 98% for synthetic task sets. We found that the performance gain was the largest for schedulable task sets, whereas for unschedulable task sets POR was often slower than the original analysis while still providing some reduction in the explored number of states.

To achieve this performance gain, we compromised the exactness of the response-time bounds (though they remained safe). Yet, our experiments show this over-estimation to be very small on average, namely 2% for benchmark tasks and 0.1% for synthetic tasks.

Lastly, on synthetic task sets, our solution was able to scale to 70 tasks with $6 \cdot 10^6$ jobs per hyperperiod without even approaching the time limit of four hours, whereas the original analysis already failed to complete at 25 tasks with $5 \cdot 10^4$ jobs per hyperperiod. This shows that POR allows us to analyze more complex task sets (with more jobs) than the original analysis and has the potential to scale even further.

In conclusion, partial-order reduction has proven itself to be a successful and promising technique for improving the scalability of schedule-abstraction-based analyses.

8.3 Future work

When there is a potential deadline miss, our current POR solution simply gives up on forming a reduction set. This happens more often when there is a large uncertainty in the release time of the jobs or when there are tardy (missed) workload from previous jobs in a system state. The need to explore all possible scenarios in case of a potential deadline miss may result in an exponential increase in states. That is why we currently decided to just drop a candidate set if there is a risk of deadline miss. However, we still see some potential rooms for improvement when forming a candidate set.

One such solution could be a two-step algorithm where instead of rejecting a reduction set when a potential deadline miss is detected, the analysis allows the reduction set to be applied on the graph anyway. This creates a single state representing the processor availability after executing all jobs in the partial-order reduction, which can then be used to continue the analysis for the remaining jobs. Then, the set of jobs in the partial-order reduction set is *analyzed separately* (using the original SAG analysis) to get the actual response times of those jobs and conclude whether there is an actual deadline miss or not. Note that during the aforementioned analysis, no partial-order reduction will be tried as we already concluded that merging the entire set is not possible, saving overhead from more (failed) attempts at partial-order reduction. Once this set of jobs has been analyzed, the resulting sub-graph is not expanded anymore, as the reduced state is already in the main schedule-abstraction graph. Essentially, this creates a hierarchical schedule-abstraction graph where the states created from

partial-order reduction represent a sub-graph in case of a potential deadline miss.

Continuing the analysis from the reduced state instead of expanding it in place can have an exponential impact on the performance. Imagine a graph where $\mathcal{J}^M$ is the same for all states on the front, and the partial-order reduction of $\mathcal{J}^M$ will be rejected because of a potential deadline miss. In our current approach, all possible orderings of jobs in $\mathcal{J}^M$ will be explored for each state on the front. Let's say that the front consists of x states, and that exploring all possible orderings of jobs in $\mathcal{J}^M$ requires y states. Then, $x \cdot y$ states are added to the graph. Now consider the two-step approach where the reduced state is added to the graph and $\mathcal{J}^M$ is analyzed separately. In this case, only $x + y$ states are added to the graph.

An additional benefit of this new approach is that it allows for more efficient parallelization of the analysis because the sub-graphs can be analyzed independently from the main graph. This is in contrast to the current approach which does not lend itself for efficient parallel computing. Currently, states can be generated in parallel but threads have to be merged again in case of a merge in the graph, which creates a lot of overhead in creating and merging threads.

Another way to reduce the pessimism is by making the latest start time $\widehat{LST}_i(\mathcal{J}^M, v_p)$ of a job $J_i \in \mathcal{J}^M$ more precise. Currently, the computation of $\widehat{LST}_i(\mathcal{J}^M, v_p)$ uses the worst-case scenario where J_i suffers from both maximum blocking and maximum interference, which is a scenario that rarely happens. Having a tighter upper bound on the latest start time of a $J_i \in \mathcal{J}^M$ will result in fewer false-positives for jobs interfering with $\mathcal{J}^M$, in turn increasing the number of successful partial-order reductions.

The current POR only supports work-conserving scheduling policies, whereas the original SAG by Nasri et al. [18] also supports non-work-conserving policies. In the future, POR could be extended for non-work-conserving policies by adapting the work-conserving interference condition (6.9) to incorporate various *idle-time insertion policies*.

Furthermore, support for precedence constraints can be added by applying the condition that a J_x can only interfere with $\mathcal{J}^M$ if all predecessors of J_x are finished before J_x can start. Finally, POR can be extended for multiprocessor systems.

Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. *SIGPLAN Not.*, 49(1):373–384, January 2014.

[2] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[4] Theodore P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Syst.*, 32(1–2):49–71, February 2006.

[5] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 182–190, 1990.

[6] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[7] Edmund M Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[8] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. Controller area network (can) schedulability analysis : refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[9] Friedrich Eisenbrand and Thomas Rothvoß. Edf-schedulability of synchronous periodic task systems is conp-hard. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, page 1029–1034, USA, 2010. Society for Industrial and Applied Mathematics.

[10] Friedrich Eisenbrand and Thomas Rothvoß. Static-priority real-time scheduling: Response time computation is np-hard. In *2008 Real-Time Systems Symposium*, pages 397–406, 2008.

[11] Pontus Ekberg. Rate-monotonic schedulability of implicit-deadline tasks is np-hard beyond liu and layland's bound. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 308–318, 2020.

[12] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[13] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 110–121, New York, NY, USA, 2005. Association for Computing Machinery.

[14] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *Proceedings of the 5th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems*, SEUS'07, page 263–272, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pages 129–139, 1991.

[16] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[17] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[18] 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.

[19] Mitra Nasri, Morteza Mohaqeqi, and Gerhard Fohler. Quantifying the effect of period ratios on schedulability of rate monotonic. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, page 161–170, New York, NY, USA, 2016. Association for Computing Machinery.

[20] 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*, Leibniz International Proceedings in Informatics, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, June 2018.

[21] 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, Leibniz International Proceedings in Informatics, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, July 2019.

[22] Suhail Nogd, Geoffrey Nelissen, Mitra Nasri, and Björn B. Brandenburg. Response-time analysis for non-preemptive global scheduling with fifo spin locks. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 115–127, 2020.

[23] Roger Stafford. Random vectors with fixed sum. `https://nl.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum`, 2006.

[24] Youcheng Sun and Giuseppe Lipari. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling. *Real-Time Systems*, December 2015.

[25] Beyazit Yalcinkaya, Mitra Nasri, and Björn B. Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. pages 1228–1233, May 2019.