Delft University of Technology
Master of Science Thesis in Embedded Systems

Schedulability analysis of globally scheduled preemptive applications

Srinidhi Srinivasan

Supervisor : Dr. Geoffrey Nelissen
Co-supervisor : Dr. Mitra Nasri

Embedded
Networked
Systems

TUDelft Delft University of Technology

Schedulability analysis of globally scheduled preemptive applications

Master of Science Thesis in Embedded Systems

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

Srinidhi Srinivasan
S.Srinivasan-2@student.tudelft.nl
nidhi.2396s@gmail.com

Author
Srinidhi Srinivasan (S.Srinivasan-2@student.tudelft.nl)
(nidhi.2396s@gmail.com)

Title
Schedulability analysis of globally scheduled preemptive applications

MSc Presentation Date
30-Sept-2020

Graduation Committee

Abstract

For any real-time system, being predictable with respect to time is a basic necessity. The combination of a preemptive execution model and a multiprocessor platform poses a challenge when analysing the predictability of a system. In this thesis, we present a new type of framework for the worst-case response time analysis for preemptive tasks scheduled on multiprocessor platforms. The proposed framework analyses this worst-case response time by building a schedule abstraction graph that abstracts all the execution scenarios that occur in the system. Since preemptive tasks scheduled on a multiprocessor platform creates a large state space of execution scenarios to explore, a schedule abstraction graph can easily face a state space explosion problem. A novel methodology has been introduced in this thesis, that allows us to eliminate state space explosion altogether. We used this new schedule abstraction graph framework to initially develop an analysis for uniprocessor platforms and compare it to the state-of-the-art. We then explain how the analysis can be extended from uniprocessor to multiprocessor platforms.

Preface

This journey has been many things but I could have never imagined it to be this enjoyable. And the reason for this was because of some of the amazing people that I had the privilege of working with.

I am especially grateful to Geoffrey Nelissen for the immense motivation, for helping me stay grounded and level headed and more than anything, for being a friend and a mentor. I also thank Mitra Nasri, who not only introduced me to the fascinating world of real-time systems but also inspired me carry out my own research in this wonderful field. I remain eternally grateful to the both of them.

And finally, to my parents and my brother, who even though are thousands of kilometers away, have always been there and supported me throughout this incredible journey.

Though this journey has come to an end, it is has opened up many more exciting and adventurous doors, and I cannot wait to find out what lies ahead

Srinidhi Srinivasan

Delft, The Netherlands
24th September 2020

Contents

List of Figures

Chapter 1

Introduction

Dependency on computing systems have been on the rise as the integration of these systems in our daily lives have become more seamless day by day. These closely integrated devices are known as cyber-physical systems. Cyber-physical systems have critical functionalities, where a small malfunction could have adverse impacts on the portion of the physical world that are influenced by them. Hence, checking the behaviour of such systems is important. If we also account for the timing of their behaviour, then the systems are known as real-time systems. Therefore, real-time systems are a subset of cyber-physical systems, where both the functional and temporal correctness are necessary requirements.

1.1 The problem

Real-time systems like automotive systems typically have a repeating (periodic) pattern of workload (tasks) [20],[14],[6]. Since they are periodic, we can predict when a task should be executed and if it executes within a stipulated amount of time. In a multi-tasking system, tasks must be *scheduled* by a scheduling algorithm in order to be able to access shared hardware resources such as the processor. The system is said to be temporally correct if all tasks for every possible schedule in the system finish before their respective deadlines [11]. In this thesis, we propose new tools that can help us analyse the temporal correctness of systems, assuming that the system functions correctly.

With the burgeoning demand for high-performance applications, parallel processing is becoming progressively relevant to cyber-physical systems and hence many critical applications are being developed as parallel programs[6]. The availability and use of multicore and many-core processing platforms allow for more efficient use of resources and thus leads to lower response times.

A common execution model for tasks in a real-time system is *preemptive execution* [6]. In preemptive execution, a task that is running on the processor can be preempted (halted) for some time before it is resumed. Most multiprocessor scheduling policies allow to resume a preempted task on a different core than it was executing before (this phenomena is called task migration). While the combination of preemptive execution and migration increases the ability of the scheduling algorithm to meet the timing constraints of the tasks (also called *schedulability*) and better utilize the platform, it significantly increases the complexity of the response-time analysis of the tasks in the system as all the execution scenarios need to be accounted for. The goal of this thesis is to derive a schedulability analysis for preemptive applications on multiprocessor platforms while ensuring that pessimism is curbed to the maximum extent.

1.2 Limitations of the state-of-the-art

Attempts at solving the problem of building a schedulability analysis technique for preemptive tasks under global scheduling have been made in the past. Those solutions have been reviewed in Sun et al. [27] and all the analysis methodologies either have scalability or accuracy issues. Sun et al. [27] shows us that solutions that are accurate cannot handle realistic system sizes and solutions that can scale very well to large system sizes are highly inaccurate. This causes a gap in the state-of-the-art making it important to develop a system that is highly accurate while being able to scale to realistic system sizes.

The state-of-the-art solutions can be categorized into two main categories. Many solutions form an analysis technique by extending the solution by Audsley et al. [2] which is a schedulability analysis technique for tasks that can be preempted and are scheduled on uniprocessor systems. Many different works such as the solutions by Guan et al. [16] and Bertogna et al. [7] extend the response time analysis solution by Audsley et al. [2] to multiprocessor systems. This forms the basis of the first category of the state-of-the-art. The second category of solutions are those that that try to tackle the problem by using finite state machines such as the solution by T.P. baker and M. Cirinei [4].

1.2.1 Solutions based on finite state machines

The solution by T.P Baker and M. Cirinei [4] explores the schedulability problem where the entire state space was computed by means of a finite state machine. In such solutions, the expanse of the state space is determined by the granularity of time that is considered while exploring all possible schedules. By using finite state machines we can ensure that we explore the entire state space thoroughly thereby obtaining a solution that is highly accurate. The drawback, however, is that as the task sets start getting larger, the state space that has to be explored also gets very large. Hence, such solutions do not scale well at all and are only able to handle systems of very small sizes. Many works such as the solution by Burmyakov et al. [10] and Bonifaci et al.[9] try to improve this method by reducing the number of states that are explored by eliminating unnecessary states that do not contribute to the schedulability of a task set. Even with these improvements, the state space that needed to be explored still remains very large and hence such solutions cannot scale to large system sizes. In the work of this thesis, we build a schedulability technique that does not look at schedules with respect to time but rather only looks at the ordering of the arrival of the jobs which helps to reduce expanse of the state space.

1.2.2 Solutions that extend the response time analysis solution on uniprocessor systems

The response time analysis solution established by Audsley et al. [2] allows for the analysis of tasks that can be preempted and are scheduled on uniprocessor systems. Extension of this solution to multiprocessor platforms is very simple and hence we have many various works by Guan et al. [16] and Bertogna et al. [7] and many more that try to extend this solution to handle global scheduling. These solutions however do not aim to obtain an accurate analysis and therefore try to obtain over approximated bounds on the response time in order to determine the schedulability of a system. Due to this over-approximation method, these class of solutions are able to scale up to very large system sizes but fall short when accuracy is in question. These solutions are very similar and try to find the response time of a task by determining the interference that the task faces. The goal of this class of works is to make sure that upper bounds on the interference are made as tight as possible thereby increasing the accuracy of the solution. Even then, these solutions are very pessimistic and from a set of schedulable tasksets, only a small portion of these tasksets are deemed schedulable.

1.2.3 Schedule abstracion graph

Our motivation at aiming to fix this gap in the state-of-the-art comes from building a directed acyclic graph that abstracts preemptive tasksets and their schedules. Schedule abstraction graphs have previously been built for non-preemptive tasks as can be seen in the works of Nasri et al. [22], [23] and [24]. Here, we are introduced to a new methodology that designs a schedulability analysis technique that builds a graph that abstracts the schedules of a system. These solutions handle tasks that are non-preemptive or limited-preemptive which means that the execution of the tasks cannot be halted at any time. These solutions are the motivation for building a schedulability technique for preemptive tasks that are both accurate and can scale to realistic system sizes. The details of these solutions are discussed in Section 4.1.

1.3 Our Solution

In order to build a schedulability analysis that can analyse tasks under global scheduling, the initial approach is to first build a solution that works well for uniprocessor systems. Once this solution on uniprocessor systems has been tested and evaluated to see how accurate and efficient it is when compared to the state of the art, the solution can then be extended to analyze multicore platforms executing tasks under global preemptive job-level fixed-priority scheduling. In Section 5.2 we first see a solution where a schedule abstraction graph has been built for preemptive tasks scheduled on a uniprocessor system. The extension of this system to multiprocessor systems have then been discussed in detail in Section 5.1.

1.4 Manuscript organization

The thesis report has been subdivided into five sections. Chapter 2 presents the system model and describes the terminology required to understand the problem. Chapter 3, provides an overview of the state-of-the-art solutions and their analysis methodologies. Background information required to build a schedulability analysis technique has also been explored in Chapter 4. Chapter 5 illustrates the detailed approach to the solution on a uniprocessor platform and the reasoning behind the design decisions. An extension of the solution from a uniprocessor platform to a multiprocessor platform has been elucidated in Chapter 6. Chapter 7 describes the key findings of the thesis and illustrates what can be done in future work.

Chapter 2

System model

We consider the problem of globally scheduling preemptive tasks on a multiprocessor platform with m identical cores. In this work, we assume that tasks have the ability to migrate between the cores.

2.1 Workload Model

The term *task* refers to a piece of code that implements one of the system's functionalities. We denote the i^{th} task of the system by τ_i. In this work, we assume that tasks must execute periodically. Each time it must execute, it releases an instance that is added to the ready queue of the system. An instance of a task is called a job. We denote the j^{th} job of τ_i by $J_{i,j}$. Each job $J_{i,j}$ is defined by its release time $r_{i,j}$ (i.e., the time at which it becomes ready for execution), its actual execution time $c_{i,j}$ (i.e., how much time it keeps a core busy) and its absolute deadline $d_{i,j}$ (i.e., the absolute time by which it must finish its execution). Due to non-determinism aspects, it is usually impossible to know the exact release and execution time of a job a priori. Therefore, for analysis purposes, we model a job by its earliest and latest release times $r_{i,j}^{min}$ and $r_{i,j}^{max}$, its minimum and maximum execution time $c_{i,j}^{min}$ and $c_{i,j}^{max}$, and its absolute deadline $d_{i,j}$. The earliest release time $r_{i,j}^{min}$ is also called the arrival time of job $J_{i,j}$.

Similar to a job, a task is defined by various parameters such as its execution time, period, relative deadline etc. The *execution time* of a task is the amount of time each of its jobs takes to perform its functionality. As already mentioned above, in real systems, this execution time is rarely constant and each job released by the task may exhibit a different execution time. Therefore, in this work, we specify the execution time of a task τ_i with a lower bound (C_i^{min}) and an upper bound (C_i^{max}) on the execution time of each job released by τ_i. The *period* (T_i) of a task τ_i denotes the inter-arrival time between two consecutive jobs of τ_i. That is, we have $r_{i,j+1}^{min} - r_{i,j}^{min} = T_i$ for $j >= 0$. The *relative deadline* (D_i) of the task τ_i is the time within which the execution of a job of τ_i must be completed. This parameter is relative to the time at which each job $J_{i,j}$ of a task τ_i arrives in the system. That is, for the deadline of a job, we have $d_{i,j} = r_{i,j}^{min} + D_i$. A simple example of a task, its jobs and their parameters have been illustrated in Figure 2.1.

We also assume that tasks may have an offset and experience a release jitter. The task release *jitter* refers to the uncertainty on the actual time at which a job is effectively released in the system and thus competes for the computational resource. A release jitter of X_i denotes that a job $J_{i,j}$ can release anytime between $r_{i,j}^{min}$ and $r_{i,j}^{max} = r_{i,j}^{min} + X_i$. An *offset* O_i defines the arrival time of the first job of τ_i, i.e., $r_{i,0}^{min} = O_i$.

We assume that the time is discrete, i.e., all the jobs and tasks parameters are integer multiples of a basic time unit (e.g., a clock tick).

Task ID (i)	Minimum execution time (C_i^{min})	Maximum execution time (C_i^{max})	Period (T_i)	Jitter (X_i)	Offset (O_i)	Relative deadline (D_i)
1	4	5	10	1	1	7

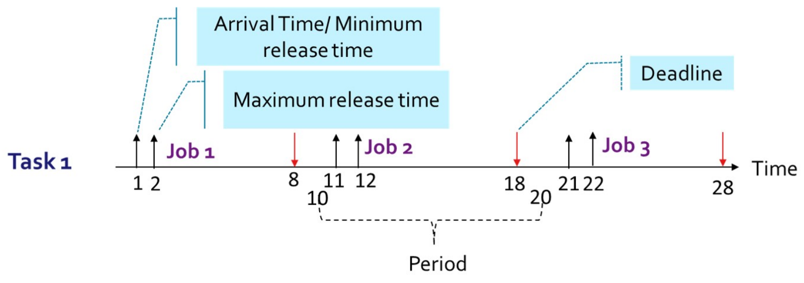
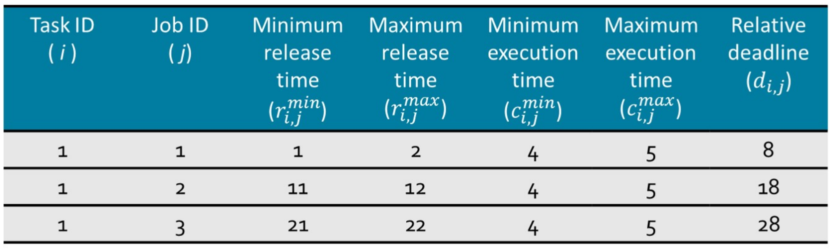


Figure 2.1: **Example of a task its parameters**

We denote the set of all tasks in the system by τ, and the set of all jobs released by tasks in τ in an a priori computed observation window by $\mathcal{J}$. In Figure 2.2, we see an example of a set of the first three jobs of task τ_1 seen in Figure 2.1. These jobs and their parameters have been displayed in Figure 2.2.



Task ID (i)	Job ID (j)	Minimum release time ($r_{i,j}^{min}$)	Maximum release time ($r_{i,j}^{max}$)	Minimum execution time ($c_{i,j}^{min}$)	Maximum execution time ($c_{i,j}^{max}$)	Relative deadline ($d_{i,j}$)
1	1	1	2	4	5	8
1	2	11	12	4	5	18
1	3	21	22	4	5	28

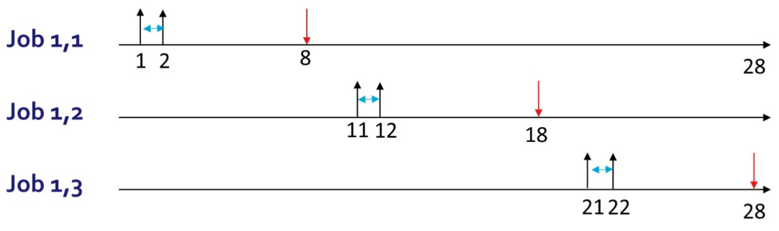


Figure 2.2: **Example of jobs and their parameters**

A job's response time is considered as the time between the earliest release time of the job and the time at which the job has completed its execution. This is in accordance to Audsley et al. [1], which states that the release jitter of a job is also a part of the response time of a job. Since the parameters of the job arre not deterministic, the response time can have a variety of values too, and hence for a taskset, each job is said to have a *best-case response time*(BCET) and the *worst-case response time*(WCET). If the worst-case response time is always smaller or equal to the deadline of a job, for all jobs, in every possible schedule of taskset, then that taskset is said to schedulable.

To avoid notation clutter, in the rest of this document, we omit the index i of the task when referring to a generic job $J_{i,j}$, hence we only write J_j.

2.2 Execution Model

We assume that tasks are preemptive, which means that the execution of their jobs can be interrupted by a higher priority task as exemplified in Figure 2.3. We assume that priorities are assigned to jobs using any *job-level fixed priority*(JLFP) scheduling policy. This means that the priority of the job is fixed and does not change during runtime. The family of JLFP scheduling policies include well-known policies such as *rate monotonic*(RM), *deadline monotonic*(DM) and *earliest deadline first*(EDF). The priority of a job J_i is denoted as p_i. The lower the numeric value of p_i, the higher it's priority i.e., if $p_i < p_j$ then J_i has a higher priority than J_j. For the sake of simplicity, in this work, we assume that no two jobs have the same priority and therefore a tie-breaking rule is not required.
In this work, we assume that each job runs sequentially, which means that a job can run on only one core of the multiprocessor platform at any given time. However, we assume a global scheduling policy, meaning that a job may migrate from one core to another during runtime. A job is said to be ready if it has been

released into the system and is not running(i.e., executing on a processor). Since we assume a global JLFP scheduling policy, the scheduler always dispatches the m highest priority ready jobs. The scheduler is assumed to be work-conserving which means that the scheduler does not leave a core idle if there is a ready job, that is, if a core is free and at least one job is ready, then the highest priority job is immediately dispatched on that core. The tasks considered here do not have any precedence constraints and hence when a job arrives in the system, it can be released onto the system without checking for any dependency with other jobs.

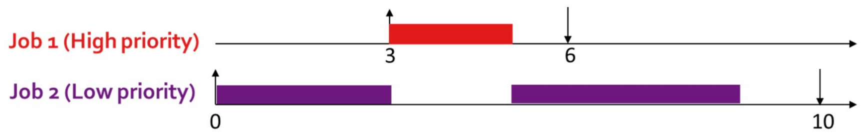


Figure 2.3: **A lower priority task being preempted due to the arrival of a higher priority task**

Chapter 3

Related Work

Analysing the worst-case response time of *sporadic* preemptive tasks scheduled with task-level fixed priority on uniprocessor platforms is deterministic as the worst-case scenario was proven in Liu and Layland [21] to occur when all the tasks release their first job into the system synchronously and all next jobs are released as fast as possible. Therefore, exact analyses were already proposed for such systems. Exact schedulability tests are those tests that are able to find all the schedulable tasksets and reject all the unschedulable ones while sufficient solutions can only determine a subset of the schedulable tasksets as schedulable. Figure 3.1 illustrates the difference between exact and sufficient solutions.

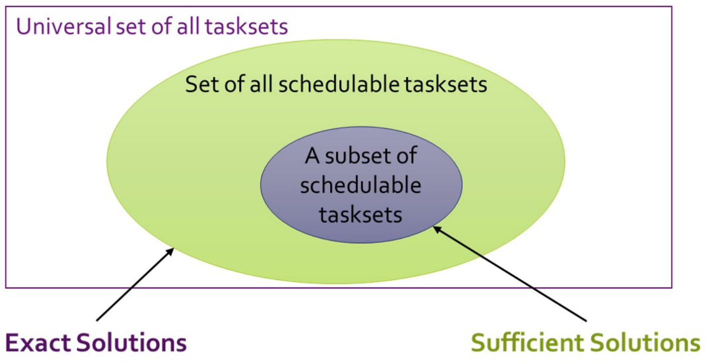

Universal set of all tasksets
Set of all schedulable tasksets
A subset of schedulable tasksets
Exact Solutions
Sufficient Solutions

Figure 3.1: **Visualization of exact solutions and sufficient solutions**

The analysis methodology in use for uniprocessor platforms which finds the worst-case scenario that applies to all jobs in the system cannot be easily applied to multiprocessor platforms as there does not exist just a single worst-case scenario for each job. Both the works of Sun et al. [28] and Davis et al.[12] show that each job may experience its worst-case for a different execution scenario, and no common pattern as yet been identified. Due to this, an exact analysis for sporadic tasks scheduled on multiprocessor platforms under a task-level or job-level fixed priority scheduling scheme requires analysing every possible job release pattern. Since such an exhaustive search does not scale, many works attempt at deriving sufficient solutions that are pessimistic instead.

Hence, there are two ways in which this problem can be handled. On the one hand, an exhaustive analysis that is exact can be derived but the scalability problem has to be overcome. Such an exhaustive search was performed by many solutions by using finite state machines. On the other hand, we have solutions that are pessimistic instead because they only aim at being sufficient, but the problem of being too inaccurate has to be overcome. Sufficient tests usually extend the analysis of Audsley et al. [2] from uniprocessor to multiprocessor platforms. Based on these two approaches, we divide the state-of-the-art analysis into two categories. Section 3.1 provides an overview of the solutions that use finite state machines and Section 3.2 provides an overview of the solution that provides analysis methodologies that extend the solution by Audsley et al. [2].

3.1 Solutions based on finite state machines

The earliest solutions that used finite state machines to perform an exhaustive search of the large state-space caused by the numerous job release patterns were by T.P. baker and M. Cirinei [4]. The proposed solution is only practical for very small system sizes because the time and space complexity of this solution prevents it from scaling to large system sizes. Many solutions provided improvements to this initial solution. Due to the complexity of T.P. baker and M. Cirinei [4], Geeraerts et al.[15] improved this solution by identifying and eliminating states in the state space that need not be explored. The solution by Bonifaci et al. [9] also provides a methodology that improves the solution by T.P. Baker and M. Cirinei [4] by developing a new solution based on a new two-player game strategy. This methodology inspired Burmyakov et al. [10], to extend this methodology and find a faster and less memory consuming schedulability test by further reducing the state-space that needed to be explored. Even though many solutions did try to reduce the state-space explored, the complexity of the solutions still did not allow for scalability as established by Sun et al. [27].

3.2 Solutions that extend the response time analysis solution on uniprocessor systems

Due to the exhaustive search problem faced by exact solutions, the majority of the research that has been conducted derived solutions that are sufficient and hence pessimistic. Baker[3] provides a schedulability analysis technique that computes the interference and worst-case response time of tasks within a selected time interval. Bertogna et al.[8], shows that the solution can be analysed within a problem window and found a solution that was less pessimistic when a large execution time variation was present.

The solution by Bertogna et al.[7] divided the tasks that cause interference within a problem window into two categories: 1) carry-in tasks that release before the start of the problem window and 2) non carry-in tasks that start within the problem window. Baruah [5], finds a way to upperbound the interference caused by the carry-in jobs thereby improving the accuracy of the solution.

The solution of Baruh [5] dealt with tasks that were scheduled under a global fixed priority scheduling policy. The extension of this solution to global earliest deadline first scheduling policy was proposed by Guan et al.[16]. Sun and Lipari [28] use a similar method as in Guan et al.[16] and improved the schedulability of tasks scheduled under global fixed priority scheduling policy. Motivated by building an analysis that handles scheduling policies that are non-fixed priority, Lee and Shin [19] built a schedulability analysis technique for any work conserving scheduling algorithm.

Since all the solutions discussed above are sufficient solutions, they are pessimistic which means that the results are inaccurate. On the other hand, all the exact solutions that use finite state machines perform exhaustive search and thus do not scale. Our aim in this thesis is to bridge this gap and build a solution that is accurate while being able to scale to realistic system sizes.

Chapter 4

Background

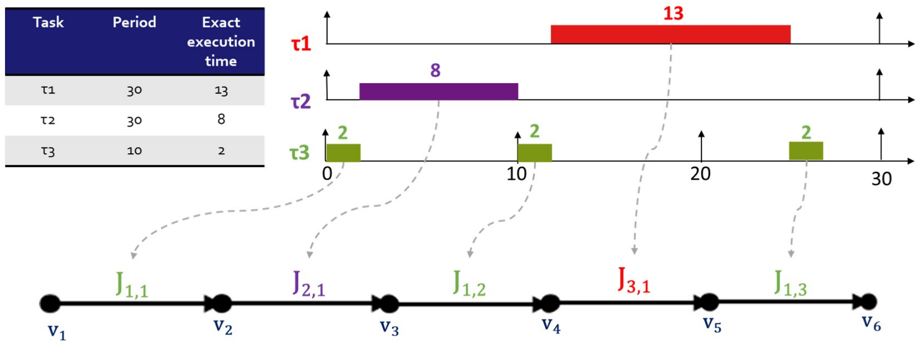
4.1 Schedule abstraction graph

The schedulability analysis presented in this document builds upon the notion of schedule abstraction introduced in [24] and [23]. In order to understand how such an analysis works, we present the original schedule abstraction graph analysis by Nasri et al. [24] in this section.

The schedule abstraction graph framework developed by Nasri et al. [24] was intended to analyse the schedulability of *non-preemptive* tasks by exploring, at design time, all possible schedules that may potentially happen at runtime. The analysis was building a graph (called schedule abstraction graph) by analysing all possible job orderings that may result from different execution scenarios (see Definition 4.1). Each job ordering is represented by a path in the graph. As stated in Definition 4.1, different execution scenarios result from different possible release time and execution times of a set of jobs. Note that if there was no release jitter and execution time variation, then we would have only one possible execution scenario resulting in a single possible job execution ordering. For such a deterministic system, the schedule abstraction graph would just be composed of a single path as there is only a single possible sequence of job execution ordering (see Figure 4.1).

Definition 4.1. [24] An execution scenario $\gamma = (r_1, C_1), (r_2, C_2), ..., (r_n, C_n)$, where $n = |J|$, is an assignment of execution times and release times to the jobs of J such that, for each job $J_i, C_i \in [C_i^{min}, C_i^{max}] and r_i \in [r_i^{min}, r_i^{max}]$.

Formally, a schedule abstraction graph is a directed acyclic graph $G =< V, E >$ where V is the set of vertices and E is the set of directed edges. The vertex $v_0 \in V$ represents the system state when all cores are idle. Each edge $e_x \in E$ connecting a vertex v_y to v_z (with $v_y, v_z \in V$) is labelled with a job J_j as can be seen in Figure 4.2. It represents a job being dispatched by the scheduler in the system state represented by v_y. Then, the vertex v_z models the state of the system after dispatching the job J_j in system state v_y. Hence, a path in the graph G defines a possible job execution ordering that could occur at runtime and each vertex on that path represents the intermediate states of the system during the schedule associated to that job execution ordering.



Task	Period	Exact execution time
τ1	30	13
τ2	30	8
τ3	10	2

Figure 4.1: **Schedule of a deterministic taskset with exact execution time modelled by the schedule abstraction graph**

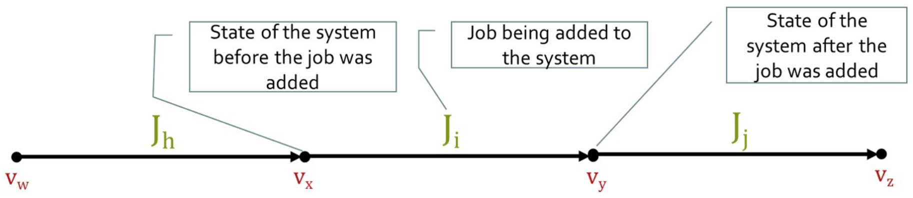
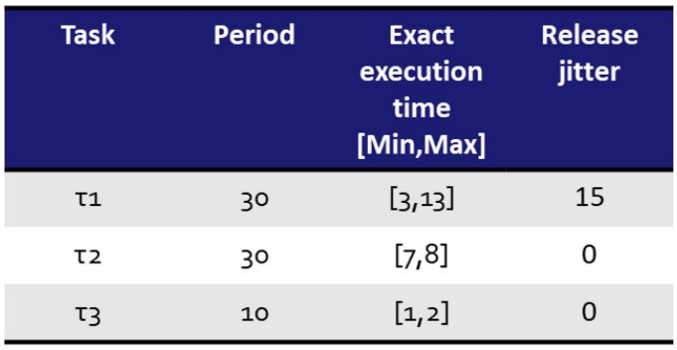

State of the system before the job was added
Job being added to the system
State of the system after the job was added
J_h
J_i
J_j
v_w
v_x
v_y
v_z

Figure 4.2: **Visual representation of the components of the schedule abstraction graph**

Example 4.1.1. When there are uncertainties on the actual release and execution time of each job, the schedule abstraction graph no longer comprises a single path. Instead, it branches every time the scheduler may take a different scheduling decision. To understand this further, Figure 4.3 presents the schedule abstraction graph of a taskset with execution time uncertainty and release jitter. Every task experiences execution time variation as specified in Figure 4.3(a). Task τ_3, is the only task with a release jitter (equal to 15 time units). There are multiple job execution orderings that can occur at runtime. In this example, we only consider two of them (see Figure 4.3(b) and 4.3(c)). The first schedule (see Figure 4.3(b)) assumes that the first job of τ_2 executes for 8 units of time and the first job of τ_3 is released at time 13. Given these assumptions, the taskset is schedulable , i.e., there is no job that misses its deadline. In the second schedule (see Figure 4.3(c)), it is assumed that the first job of τ_2 executes only for 7 time units and τ_3 releases its first job at time 5. This leads to a schedule where τ_3 starts executing before the second job of τ_1, causing a deadline miss for that job. Hence, the system is not schedulable under these assumptions.

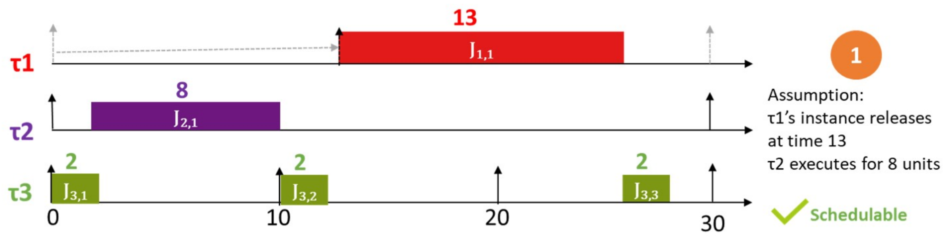
All possible schedules that may occur at runtime are included in the schedule abstraction graph. As we can see in Figure 4.3(d), when there is a variation in the job orderings, the graph branches.

Note that for systems composed of has a large number of tasks and where the tasks themselves have large execution time variations and release jitters, the number of possible execution scenarios and thus the schedule abstraction

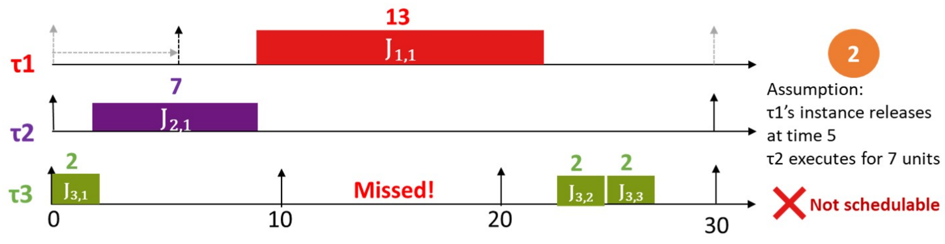


Task	Period	Exact execution time [Min,Max]	Release jitter
τ1	30	[3,13]	15
τ2	30	[7,8]	0
τ3	10	[1,2]	0

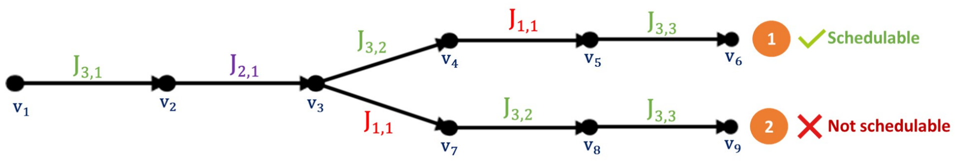
(a) Example taskset



(b) Schedule where τ_1 releases at time 13 and τ_2 executes for 8 units



(c) Schedule where τ_1 releases at time 5 and τ_2 executes for 7 units



(d) Generated schedule abstraction graph

Figure 4.3: **Schedule of a non-deterministic taskset with execution time variation and release time jitter modelled by the schedule abstraction graph**

graph would rapidly grow. This could quickly lead to a state space explosion problem where there are too many parallel branches to analyse, to be able to complete the analysis in an acceptable amount of time. In order to curb this problem, the schedule abstraction graph analysis framework implements merge techniques where similar paths with similar states are merged together. The states are said to be similar if the paths up until those states contain the same set of jobs on their edges and some other properties (specific to non-preemptive systems and thus not relevant in the context of this document) are respected. Using a breadth-first search approach, the states of all the paths in the graph are compared and merged in order to reduce the total number of vertices in the graph. This helps to ensure that the number of vertices and paths are kept as small as possible for as long as possible (see Figure 4.4for an example of what a schedule abstraction graph would look like after applying the merge rule).

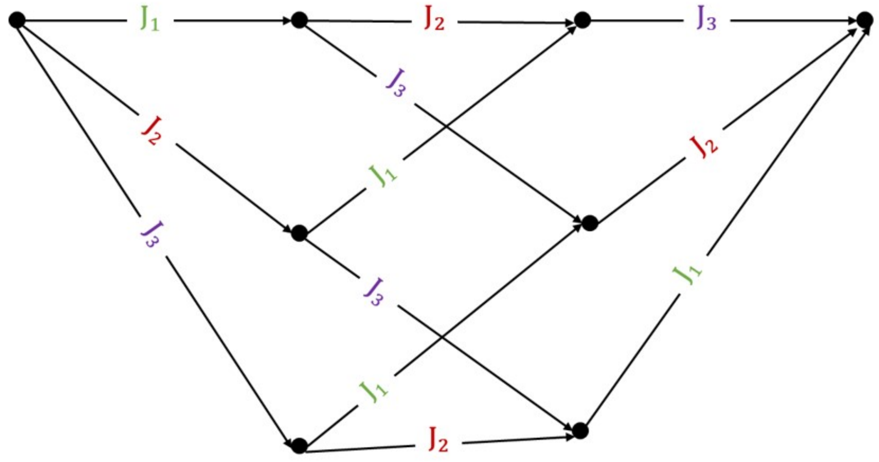


Figure 4.4: **An abstract depiction of the merge technique used in schedule abstraction graphs in order to curb the state space explosion problem.**

Chapter 5

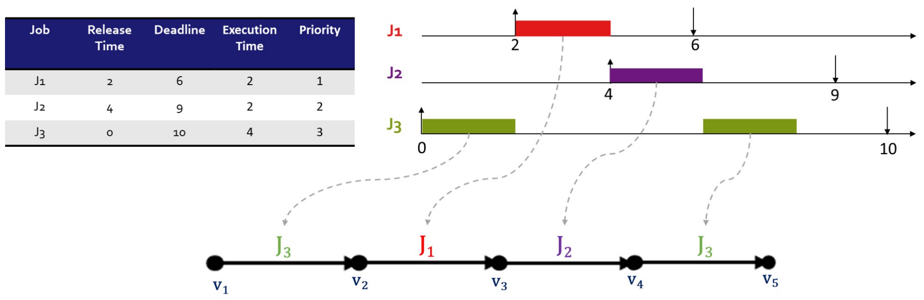
Response-time analysis for single-core platforms

To build a schedulability analysis for preemptive parallel tasks under global scheduling using the schedule abstraction graph idea, we first built a solution for the simpler, yet still challenging problem of analyzing preemptive tasks executed on a uniprocessor platform. The intent is to later extend the developed solution for single-core platforms to analyse tasksets scheduled under a global scheduling policy on multicore platforms. We present the solution for single-core platforms in this chapter, and present the extension to multicore platforms in Chapter 6.

5.1 Naive extension of the schedule abstraction graph to preemptive scheduling

The most intuitive approach to extending the methodology of Nasri et al.[22] is to keep the same semantic for the graph. That is, paths depict possible job execution orderings and nodes represent abstract system states. The major challenge with this approach is that in preemptive systems, jobs can now be preempted and hence the execution of a job may be divided into several segments. Thus, instead of just maintaining the orderings of jobs, the ordering of the segments of the jobs need to be kept track of in the schedule abstraction graph. For instance, Figure 5.1 presents an example where a high priority job J_1 is released at time instant 2. Therefore, the currently running job J_3 is preempted to allow the higher priority job J_1 to execute instead. This means that a segment of J_3 is added to the schedule abstraction graph before job J_1 and the second segment of J_3 is added later in the graph when J_3 resumes its execution. That is, the job J_3 appears twice in the same path of the graph.

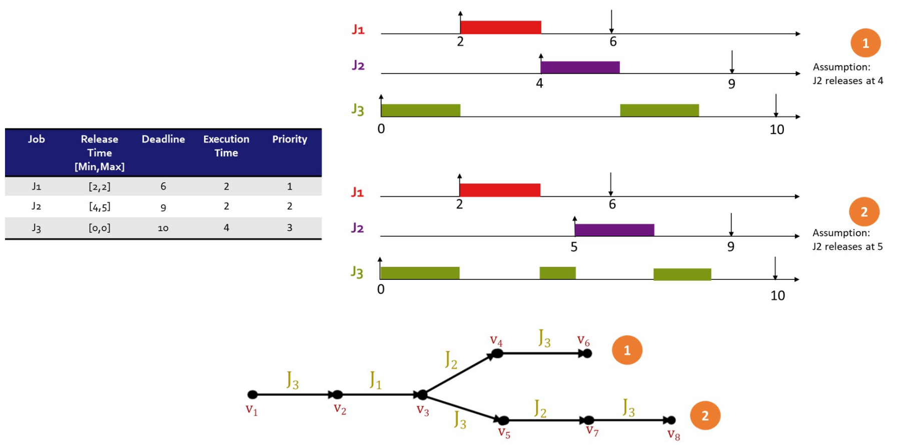
When release time jitter or execution time variation is introduced to the example in Figure 5.1, then for a single taskset, there would be multiple ways a job may be preempted, and thus the same job may be divided in different numbers of segments in each execution scenario. An example of such a situation is shown in Figure 5.2. In this example, out of multiple possible execution scenarios, we show two possible schedules. Here, we can see that in scenario 1, J_3 is preempted once, creating two segments in the graph while in scenario 2, J_3



Job	Release Time	Deadline	Execution Time	Priority
J1	2	6	2	1
J2	4	9	2	2
J3	0	10	4	3

Figure 5.1: **Segment ordering with preemptive tasks**

is preempted twice creating three segments in the schedule abstraction graph. Similar to the solution in Nasri et al. [22], each job segment execution ordering creates a different branch in the schedule abstraction graph.



Job	Release Time [Min,Max]	Deadline	Execution Time	Priority
J1	[2,2]	6	2	1
J2	[4,5]	9	2	2
J3	[0,0]	10	4	3

Figure 5.2: **Segment ordering with preemptive tasks and non-deterministic parameters**

Now that we understand, at a high level, how a schedule abstraction graph would look like for preemptive systems, we discuss how states would be merged together to curb the state space explosion problem. In order to apply the merge techniques from Nasri et al. [22], we have to merge states in a breadth first search manner. States are similar if they have the same set of completed jobs. Therefore, the states that could potentially merge in Figure 5.2 are states v_6 and v_8 as both of them have the same set of completed jobs ($\{J_1, J_2, J_3\}$). But with breadth-first, search states v_6 would only be compared to v_7 for the possibility of a merge. Yet in state v_7, J_3 has not yet been completed and hence states v_6 and v_7 cannot be merged. We, therefore, see that the breadth-first search methodology proposed in Nasri et al.[22], for merging states in a non-preemptive system, would not be effective in the preemptive case.

The next intuitive step here would be to assume a merge technique where depth-first search is carried out instead. A depth-first search methodology for merge allows the states v_6 and v_8 to have an opportunity to merge. However, a breadth-first search would require to make decisions on which branch to explore first. If the analysis picks an appropriate branch early on, the merge technique will be very effective else it will not merge often and state space explosion will be more likely to happen.

In order to eliminate this randomness, we decided to take a different route. Hence, we defined a completely new semantic for the schedule abstraction graph and developed a new system state abstraction. Our proposed solution ensures that the graph never branches. Since the graph never branches (literally reducing itself to a single path), merge techniques are not needed at all and the problems discussed above do not apply anymore.

5.2 New system state abstraction

Due to the constraints seen in the previous section on the extension of the schedule abstraction graph by Nasri et.al. [22], we redefine the semantic of the schedule abstraction graph. Our new schedule abstraction graph contains only a single path. Every edge in the path is labelled with a job. Contrary to solutions proposed in earlier work, in our new solution, a job appears only once in the graph. Jobs appear in the path in order of their *earliest* release time.

Now, let $\mathcal{J}_x$ be the set of jobs recorded on the path until reaching vertex v_x. Then, the vertex v_x records the earliest and latest finish time of every job in $\mathcal{J}_x$ *assuming that the jobs in* $\mathcal{J}_x$ *are the only jobs executed in the system.* That is, we iteratively analyze the system by adding one more job to the analysis each time we create a new vertex.

Note that unlike the solutions proposed in the previous work, the earliest and latest finish times of *every job in* $\mathcal{J}_x$ must be recomputed whenever a new job J_a is added to the system since J_a may now preempt or block the execution of other jobs. Each vertex of the new schedule abstraction graph records the finish times of every job in $\mathcal{J}_x$. That is, for each job J_i in $\mathcal{J}_x$ we record the priority level-i completion interval, which is bounded by the *earliest finish time*(EFT) and *latest finish time*(LFT) of job J_i as shown in Figure 5.3. The completion intervals are maintained based on the priority of their associated job as this helps to easily determine the jobs in $\mathcal{J}_x$ that might be affected by a newly added job J_a.

To summarize, jobs are added to the graph based on their *earliest* release time (r_i^{min}). The job with the smallest earliest release time is the job added to the graph first. When a new state is created, the earliest and latest finish times of the newly added job J_a are computed assuming that the system is composed only of a J_a and the jobs added before J_a in the graph. Then, we also update the finish time intervals of all the jobs in the previous state that have a lower priority than J_a. Those jobs that have a higher priority that J_a in the previous state are carried over as a lower priority job cannot preempt them and hence cannot impact their finish times. The techniques involved in calculating the finish times and updating a state are explained in the following subsections in

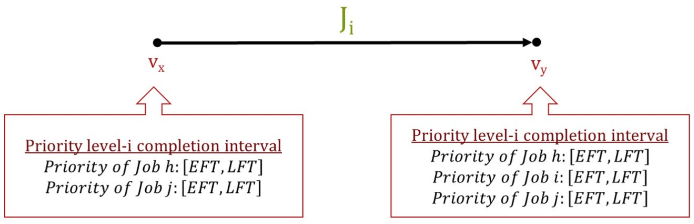

J_i
v_x
v_y
Priority level-i completion interval
Priority of Job h: [EFT, LFT]
Priority of Job j: [EFT, LFT]
Priority level-i completion interval
Priority of Job h: [EFT, LFT]
Priority of Job i: [EFT, LFT]
Priority of Job j: [EFT, LFT]

Figure 5.3: **Terminology and structure of the state of the graph.**

detail.

5.2.1 Creating a new state

Let v_x be the last vertex added to the schedule abstraction graph and let $\mathcal{J}_x$ be the set of jobs considered on the path ending in v_x. We build the next abstract system state v_{x+1} as follows. We pick the job J_a, that has not been considered yet, with the earliest minimum release, that is, $J_a = \text{argmin}_{\{J_i \in \mathcal{J} \setminus \mathcal{J}_x\}} r_i^{min}$. Then, we divide $\mathcal{J}_x$ in two subsets based on the earliest release time r_a^{min} of J_a. We define $\mathcal{A}_x$ as the set of jobs that may still be potentially active (i.e., did not complete their execution) when J_a is released. That is, $\mathcal{A}_x = \{J_i | J_i \in \mathcal{J}_x \wedge LFT_i > r_a^{min}\}$. We define $HP(J_a, v_x)$ as the set of jobs in $\mathcal{A}_x$ that have a higher priority than J_a .i.e, $HP(J_a, v_x) = J_i | J_i \in \mathcal{A}_x \wedge p_i < p_a$. We consider two cases depending on the priority of J_a relative to that of the jobs in $\mathcal{A}_x$ to create the next abstract system state v_{x+1} resulting from scheduling J_a.

Case 1. If J_a has a higher priority than all the jobs in $\mathcal{A}_x$, i.e., $HP(J_a, v_x) = \emptyset$, then there is no job in $\mathcal{A}_x$ that can block or preempt the execution of J_a. Hence, job J_a can start executing as soon as it is released and will finish its execution without interruption. Therefore, the earliest and latest finish time of J_a in v_{x+1} is straightforwardly given by Equations 5.1 and 5.2.

$$EFT_a(v_y) = r_a^{min} + C_a^{min} \tag{5.1}$$

$$LFT_a(v_y) = r_a^{max} + C_a^{max} \tag{5.2}$$

Equation 5.1 states that J_a finishes at the earliest when J_a is released as soon as possible and executes for the shortest possible duration. Similarly, Equation 5.2 states that J_a has the latest finish time when it is released as late as possible and executes for the longest possible duration.

Case 2. When the newly added job J_a has a higher priority than all the jobs in $\mathcal{A}_x$ i.e., $HP(J_a, v_x) \neq \emptyset$, then the active jobs with higher priority than J_a determine when J_a finishes its execution.

Therefore, we compute the latest finish times of the newly added job J_a using the finish time intervals of all the jobs in $\mathcal{A}_x$ that have a higher priority than J_a. To do so, we introduce the notion of worst-case *interference*, that is, the longest duration of time during which J_a remains blocked or is preempted by higher priority job when experiencing its worst-case response time. A job J_a suffers it's worst-case interference when it has been released as late as possible (r_a^{max}) as has been proven in Redell et al. [26]. Hence, the largest time that J_a is blocked or preempted when it has released at r_a^{max} is to be computed. The worst-case interference of J_a, when it is added to the new state v_{x+1} is denoted by $I_a(v_{x+1})$ and is given by Equation 5.3

$$I_a(v_{x+1}) \leq \begin{cases} 0 & \text{if } r_a^{max} \geq LFT_i(v_x), \forall J_i | J_i \in HP(J_a, v_x) \\ \sum_{J_i | J_i \in HP(J_a, v_x)} C_i^{max} & \text{if } r_a^{max} \leq r_i^{max}, \forall J_i | J_i \in HP(J_a, v_x) \\ LFT_b(v_x) - r_a^{max} & \text{otherwise} \end{cases} \tag{5.3}$$

where $LFT_b(v_x)$ is the latest finish time of job J_b such that J_b is the lowest priority job in $\mathcal{A}_x$ that has a higher priority than J_a, i.e., J_b is given by Equation 5.4.

$$J_b = \underset{J_u \in HP(J_a, v_x)}{\text{argmax}} \{p_u\} \tag{5.4}$$

We now prove in Lemma 5.2.1, 5.2.2 and 5.2.3 and Theorem 5.2.4 that Equation 5.3 provides a safe upper bound on $I_a(v_{x+1})$.

Lemma 5.2.1. If $\forall J_i \in HP(J_a, v_x), r_a^{max} \geq LFT_i(v_x)$, then the interference suffered by J_a when J_a is released at r_a^{max} is $I_a(v_{x+1}) = 0$.

Proof. For the job J_a to experience interference due to higher priority jobs, there must be higher priority workload left to execute after J_a's release. However since by assumption, $r_a^{max} \geq LFT_i(v_x)$, for all $J_i \in HP(J_a, v_x)$, all higher priority active jobs have completed their execution when J_a is released. Hence there is no interference experienced i.e., $I_a(v_{x+1}) = 0$. □

Lemma 5.2.2. If $\forall J_i \in HP(J_a, v_x), r_a^{max} \geq LFT_i(v_x)$, then $I_a(v_{x+1}) = \sum_{J_i | J_i \in HP(J_a, v_x)} C_i^{max}$.

Proof. From [26], job J_a experiences its worst case response time when it is released at r_a^{max} and all higher priority active jobs are released as close as possible from J_a's release and J_a will suffer its maximum interference when all higher priority active jobs are released at r_a^{max} and executes for their worst-case execution time (i.e., C_i^{max}).

Since all higher priority active jobs (i.e., jobs in $HP(J_a, v_x)$) were added before J_a in the schedule abstraction graph, we have $\forall J_i \in HP(J_a, v_x), r_i^{min} \leq r_a^{min} \leq r_a^{max}$. Furthermore, by assumption, we have $r_i^{max} \geq r_a^{max}, \forall J_i \in HP(J_a, v_x)$. Therefore $\forall J_i \in HP(J_a, v_x), r_i^{min} \leq r_a^{max} \leq r_i^{max}$, thereby meaning that, all higher priority active jobs can be released synchronously with J_a at r_a^{max}. Thus, the worst case interference suffered by J_a is $\sum_{J_i \in HP(J_a, v_x)} C_i^{max}$ □

Lemma 5.2.3. The interference of a newly added job J_a is upper-bounded by $LFT_b(v_x) - r_a^{max}$, where J_b is given by Equation 5.4.

Proof. We prove this lemma by contradiction. Let us assume that there is a job $J_h | J_h \in HP(J_a, v_x)$ with higher priority than J_b that interferes with the execution of J_a but whose interference on J_a is not accounted for in $LFT_b(v_x) - r_a^{max}$.

Let us now consider two cases:

Case 1. $LFT_h(v_x) \leq LFT_b(v_x)$. In this case, J_h finishes before J_b. Since $LFT_b(v_x)$ is an upper bound on the finish time of J_b, it must thus account for all possible interference generated by jobs released before its completion, and hence accounts for the interference caused by J_h. This contradicts the assumption that LFT_b does not account for J_h's interference.

Case 2. $LFT_h(v_x) > LFT_b(v_x)$. Since J_b is in $\mathcal{A}_x$, we know from the definition of $\mathcal{A}_x$ that $LFT_b(v_x) > r_a^{min}$. Furthermore, because J_h was added to the graph before J_a, we have that $r_h^{min} \leq r_a^{min}$. Therefore, we have that $r_h^{min} \leq r_a^{min} < LFT_b(v_x) < LFT_h(v_x)$. Since J_h may be potentially released before the completion of J_b (i.e., $r_h^{min} < LFT_b(v_x)$) and may also finish after the completion of J_b (i.e., $LFT_b(v_x) < LFT_h(v_x)$), it must hold that J_h can be released during J_b's execution. Because J_h's priority is higher than J_b's, J_h can thus preempt J_b's execution and therefore the worst-case finish time $LFT_b(v_x)$ of J_b must account for that preemption. Again, it contradicts the fact that the interference generated by J_h is not accounted for in $LFT_b(v_x)$.

Therefore, in both cases, there is a contradiction with the assumption that $LFT_b(v_x)$ does not account for the interference of J_h.

□

Theorem 5.2.4. Equation 5.3, is an upper-bound on the interference experienced by J_a, when it is released at r_a^{max}.

Proof. From Lemma 5.2.1, given that when the latest release time r_a^{max} of J_a is larger than the latest release time of all the jobs in $HP(J_a, v_x)$, then the $I_a(v_{x+1})$can be exactly calculated using the first case of Equation 5.3. This proves the first case of Equation 5.3.

Similarly, from Lemma 5.2.2, we know that when the latest release time r_a^{max} of J_a is smaller than the latest release time of all the jobs in $HP(J_a, v_x)$, then the $I_a(v_{x+1})$ can be exactly calculated using the second case of Equation 5.3. This proves the second case of Equation 5.3.

From Lemma 5.2.3, the third case of Equation 5.3 is always an upper bound on $I_a(v_x + 1)$.
This proves the theorem.

□

Example 5.2.1. Equation 5.3 accounts for various scenarios for the value of the interference based on the positioning of r_a^{max} with respect to the other higher priority jobs in the state. The three scenarios of the equation have been visualized in Figure 5.4.

The first scenario is when the latest release time of the job J_a (J_3 in the example) is larger than the latest finish times of all the other higher priority jobs in the system. In this scenario, the newly added job does not face any interference at all and hence can start executing as soon as it is released into the system.

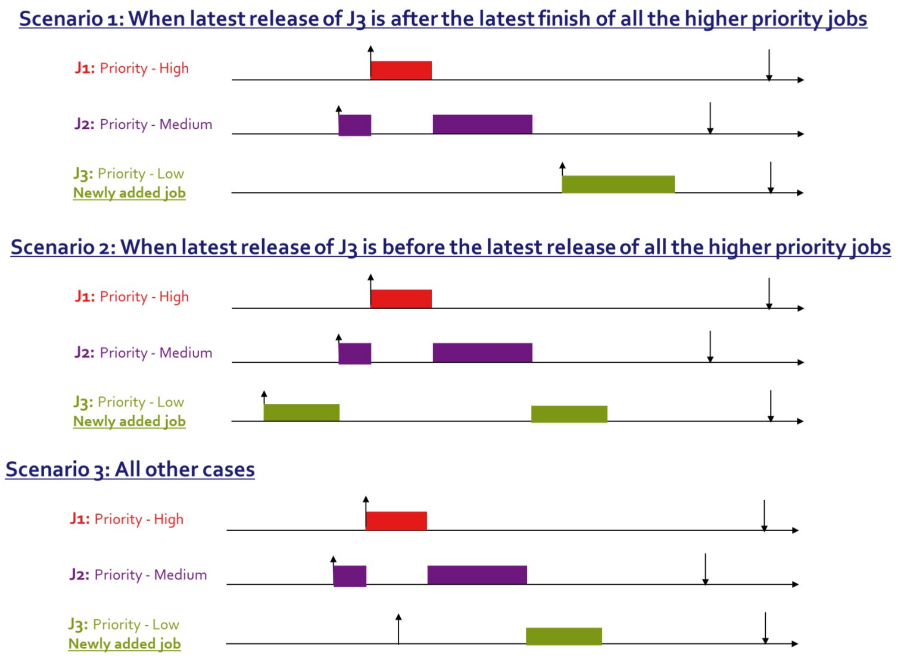

Scenario 1: When latest release of J3 is after the latest finish of all the higher priority jobs
J1: Priority - High
J2: Priority - Medium
J3: Priority - Low
Newly added job
Scenario 2: When latest release of J3 is before the latest release of all the higher priority jobs
J1: Priority - High
J2: Priority - Medium
J3: Priority - Low
Newly added job
Scenario 3: All other cases
J1: Priority - High
J2: Priority - Medium
J3: Priority - Low
Newly added job

Figure 5.4: **Reasoning for Equation 5.3. All release times depicted in this image are the latest release times of the jobs.**

In the second scenario, the latest release time of the newly added job is earlier than the latest release times of all the higher priority jobs, then the interference faced by newly added job J_a is the sum of the execution time of all the higher priority jobs.
In any other case, the release time might coincide with the execution of a higher priority job and hence the upper-bound on the interference would be until J_b (J_2 in the example) has finished its execution.

Now, that the worst-case interference has been calculated, this value can be used while calculating the latest finish time of the newly added job.

$$LFT_a(v_y) = r_a^{max} + I_a(v_{x+1}) + C_a^{max} \tag{5.5}$$

Since the maximum interference has already been calculated in equation 5.3, the $LFT_a(v_y)$ just needs its own largest execution to be added to the interference $I_a(v_{x+1})$.

Similarly, we compute a lower bound on the earliest finish time of J_a as follows:

$$EFT_a(v_y) = r_a^{min} + C_a^{min} \tag{5.6}$$

The earliest finish time of a newly added job J_a would just be the time at which J_a starts at the earliest added to its best case execution time.

5.2.2 Update a state

When a new job J_a is added onto the graph, the jobs in the state that have a lower priority than J_a may now be preempted by J_a and hence may have their finish time impacted. Thus, we update the finish time interval of each of the lower priority jobs in $\mathcal{A}_x$. When the finish time of a job is updated in system state v_{x+1}, only the interference of the newly added job J_a needs to be considered as all the higher priority job's interference were already accounted for when building v_x. The equation for computing the interference caused by J_a on a lower priority job J_i is a simplified version of Equation 5.3 , as it now needs to account for the interference of only one job. Figure 5.5 shows three scenarios depicting the ways in which lower priority jobs can face interference by J_a. The scenarios depict the various positions the r_i^{max} value can assume. If the latest release of job J_i is later than the latest finish time of the newly added job, then there is no additional interference that occurs due to J_a as seen in scenario 1. If the latest release of job J_i is earlier than the latest finish time of the newly added job, then J_a may preempt J_i and the additional interference suffered by J_i would be equal to the maximum execution time of the newly added job. In the event that the latest release of job J_i is between the latest release of J_a and the latest finish of J_a, J_i could only start to execute when J_a completes its execution and thus the interference suffered by J_i due to J_a would be upper-bounded by the difference between the release time of J_i and the latest finish time of J_a as depicted in the third scenario in Figure 5.5.

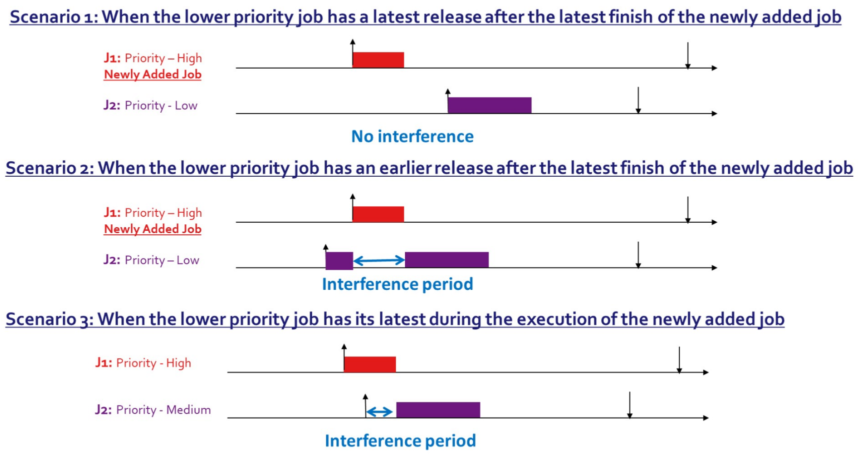

Scenario 1: When the lower priority job has a latest release after the latest finish of the newly added job
J1: Priority – High
Newly Added Job
J2: Priority - Low
No interference
Scenario 2: When the lower priority job has an earlier release after the latest finish of the newly added job
J1: Priority – High
Newly Added Job
J2: Priority – Low
Interference period
Scenario 3: When the lower priority job has its latest during the execution of the newly added job
J1: Priority - High
J2: Priority - Medium
Interference period

Figure 5.5: **Reasoning for the interference caused to a lower priority job J_2 by a newly added job J_1. All release times depicted in this image are the latest release times of the jobs.**

Before showing how we can compute the worst-case interference $I_{i,a}$ a job J_a may cause on J_i, we note that when J_a causes interference on a lower priority job $J_i \in \mathcal{A}_x \backslash HP(J_a, v_x)$ such that $r_a^{max} < r_i^{max} < LFT_a v_{x+1}$, according to scenario 3 in Figure 5.5, $I_{i,a} = LFT_a(v_{x+1}) - r_i^{max}$. However, the example presented in Figure 5.6, we see that $LFT_a(v_{x+1}) - r_i^{max}$ is in fact optimistic. That is, it returns an interference that is smaller than the actual additional worst-case interference that could occur. With the taskset in Figure 5.6(a), the worst-case schedule, when J_2 and J_3 are the only jobs in the system, is depicted in Figure 5.6(b). In that case, J_3 completes at time 13. When a new higher priority job J_1 is added, scenario 3 in Figure 5.5 would compute $I_{3,1} = LFT_1(v_{x+1}) - r_3^{max} = 1$. But if $I_{i,a}$ is only 1 time unit, then J_2 should complete at time 14, which would result in J_2 and J_3 executing simultaneously as can be seen in Figure 5.6. This is of course not possible on a uniprocessor platform. Hence, the computed value is not the additional worst-case interference but is in fact a value smaller than the additional worst-case interference.

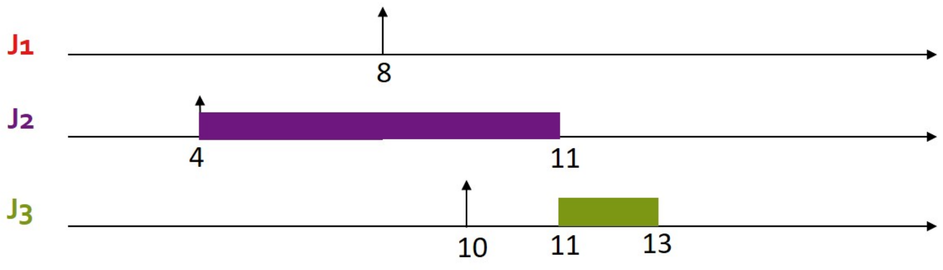
In Figure 5.6(d), we see a schedule that provides the actual additional worst-case interference. This scenario occurs because J_1 preempts J_2 and thus causes an additional interference $I_{2,1} = 3$ on J_2. This additional interference on J_2, in turn, causes the same amount of interference to J_3.

In order to properly calculate an upper-bound on the additional worst-case interference J_a may cause to a lower priority job J_i, we define the new notion of, the *start of the last level-i busy period* (SB_i), which is equal to the last time the processor was idle before the release time r_i^{max} of J_i, assuming that only jobs with equal or higher priority than J_i execute, i.e., it refers to the start of a level-i busy period. [1]. For example, in Figure 5.6(d), we see that the $SB_3 = 4$ as from time 0 to 4 the processor is idle. The main idea is that, if J_a executes within the level-i busy period, then J_a's execution fully participates in the interference suffered by J_i. For example, in Figure 5.6(d), since the level-3 busy period starts at time 4 and J_1 executes for 3 time units between time 4 and the finish time of J_3, then, J_1 interferes for 3 time units with J_3 (indirectly through J_2 in this case). The value of (SB_i), is calculated using Algorithm 1.
s

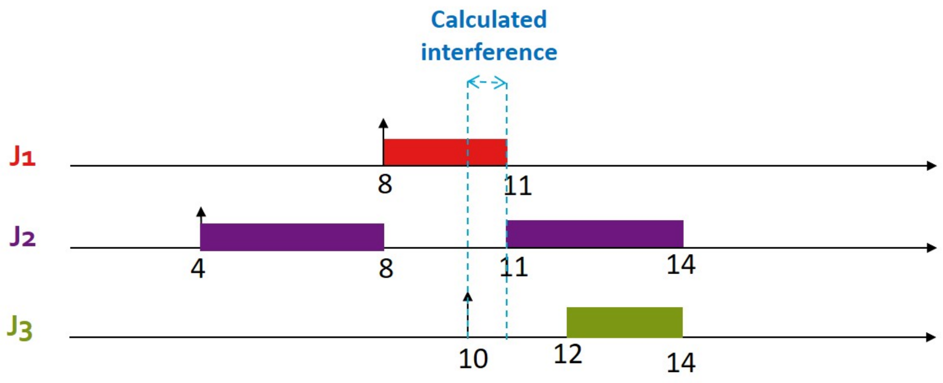
[1]A level-i busy period is an interval of time during which only jobs with priority higher or equal to that of job J_i execute [xx].

Job	Release Time [Min,Max]	Execution Time	Priority
J1	[8,8]	3	1
J2	[3,4]	7	2
J3	[6,10]	2	3

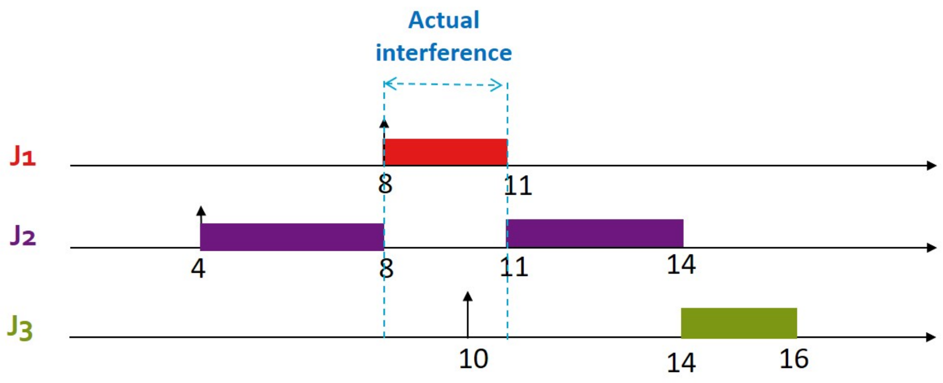
(a) Example jobset



(b) Schedule when only J_2 and J_3 have been added to the system



(c) Schedule when J_1
has been added to the system, and the $I_{i,a}$ is calculated as per Scenario 3 in Figure 5.5.



(d) Schedule when J_1 has been added to the system, and the actual $I_{i,a}$

Figure 5.6: **Under-estimation of the additional worst-case interference**

Algorithm 1: Algorithm calculating the start of the level-i busy period and updating the latest finish time of each job with lower priority than J_a

```
   Input: $\mathcal{A}_x \setminus HP(v_x, J_a)$
   Output: $SB_i$ and $LFT_i(v_{x+1})$
   Data: Vector of all the end of idle times $E$
1  for all $J_i \in \mathcal{A}_x \setminus HP(v_x, J_a)$ in decreasing priority order do
2      Initialize $SB_i$ to 0
       /* If there is no identified idle gap yet then the $SB_i$ is $r_i^{max}$ */
3      if $E = \emptyset$ then
4          Append $r_i^{max}$ to $E$
5          $SB_i = r_i^{max}$
6      else
           /* $J_k$ is the lowest priority job among the jobs that have a higher
              priority than $J_i$ */
7          $J_k = \text{argmax}_{J_u \in HP(J_i, v_{x+1})} p_u$
           /* If $r_i^{max} > LFT_k(v_{x+1})$, there is an idle time before the start
              of $J_i$ and hence $SB_i$, would be $r_i^{max}$ */
8          if $r_i^{max} > LFT_k(v_{x+1})$ then
9              $SB_i = r_i^{max}$
10             Append $r_i^{max}$ to $E$
11             Sort $E$
12         else
               /* If $r_i^{max}$ is smaller than the smallest element in $E$, then the
                  $SB_i$ is $r_i^{max}$ */
13             if $r_i^{max} \leq min(x | x \in E)$ then
14                 $SB_i = r_i^{max}$
               /* For any other value of $r_i^{max}$, $SB_i$ is the end of the last gap
                  that occurs just before $r_i^{max}$ */
15             else
16                 $SB_i = \max\{x | x \in E \land x \leq r_i^{max}\}$
17     $I_{i,a} = min\{max\{0, LFT_a(v_{x+1}) - SB_i\}, C_a^{max}\}$
18     $LFT_i(v_{x+1}) = LFT_i(v_x) + I_{i,a}$
```

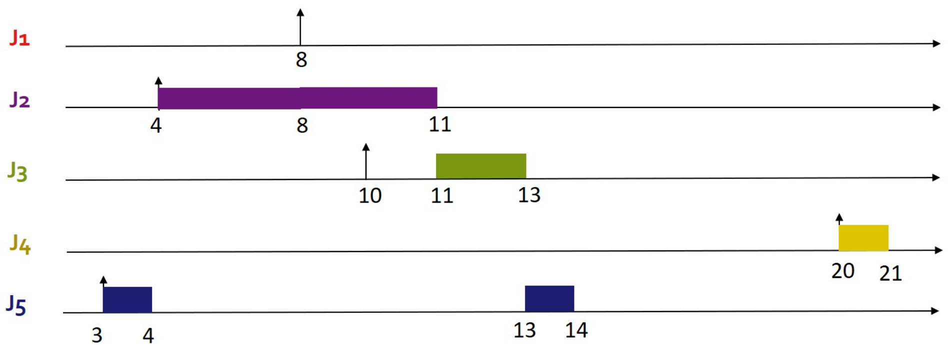
In Algorithm 1, the start of the level-i busy period (SB_i) that includes J_i is computed. In Algorithm 1, E is an ordered set maintained for all the jobs in v_x that need to be updated when building v_{x+1}. In order to explain Algorithm 1, we use the following example.

Example 5.2.2. In Figure 5.7(a), we have a set of jobs, and in Figure 5.7(b), shows the schedule all the jobs in the jobset before adding J_1. In Figure 5.7(c), we see how the schedule will be updated when J_1 is added and we must calculate the start of the level-i busy period of each job using Algorithm 1. For J_2, from line 3 in Algorithm 1, we can see that since it is the first job to be updated, E is empty. Hence, $SB_2 = r_2^{max} = 4$. This value is appended to E. Note that it is indeed true that the level-2 busy period starts at time 4 in Figure 5.7(b). For J_3, since $E = \{4\}$ and hence is not empty and $r_3^{max} \not> LFT_2(v_{x+1})$, from line

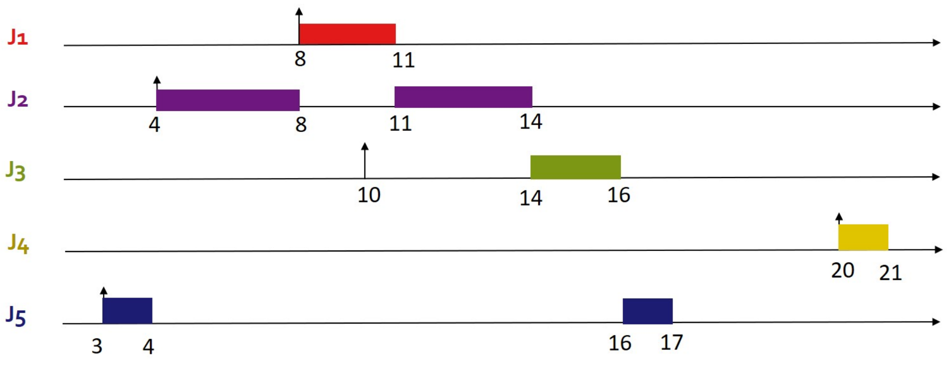
16 in Algorithm 1, we have $SB_3 = 4$. Again, we can see in Figure 5.7(b) that only jobs with a priority higher than or equal to that of J_3 executes from time 4 onward. For J_4, since we have $(r_4^{max} = 20) > (LFT_3(v_{x+1}) = 16)$, from line 8 in Algorithm 1, we have found a new idle period in the schedule. The end of this idle time, i.e., r_4^{max} is appended to E and the start of the level-4 busy period is given by $SB_4 = r_4^{max} = 20$. For job J_5, we have $E = 4, 20$ and thus $(r_5^{max} = 3) < (min(x|x \in E) = 4)$ and hence from line 14 in Algorithm 1, we calculate $SB_5 = r_5^{max} = 3$.

Job	Release Time [Min,Max]	Execution Time	Priority
J1	[8,8]	3	1
J2	[3,4]	7	2
J3	[6,10]	2	3
J4	[19,20]	1	4
J5	[2,3]	2	5

(a) Example jobset



(b) Schedule before J_1 is added onto the system



(c) Schedule when J_1 has been added to the system, $I_{i,a}$ is calculated according to Algorithm 1.

Figure 5.7: **Upper bound of the additional worst-case interference as given by Algorithm 1**

We now prove that the additional interference that J_a causes to J_i , i.e., $I_{i,a}$ can be computed with Equation 5.7 using the start of the busy period SB_i.

$$I_{i,a}(v_{x+1}) = min\{max\{0, LFT_a(v_{x+1}) - SB_i\}, C_a^{max}\} \tag{5.7}$$

Lemma 5.2.5. If $J_i \in \mathcal{A}_x \setminus HP(J_a, v_x)$, such that $SB_i \geq LFT_a(v_{x+1})$, then $I_{i,a}(v_{x+1}) = 0$.

Proof. Similar to the proof for Lemma 5.2.1, for the job J_i to suffer interference ($I_{i,a}(v_{x+1}) > 0$) due to the newly added higher priority job J_a, the level-i busy period should start either before or during the time when J_a executes i.e., the processor must be busy executing J_i or higher priority jobs than J_i within the interval $[r_a^{min}, LFT_a(v_{x+1})]$.

However, since by assumption $SB_i \geq LFT_a(v_{x+1})$, J_a has completed its execution before J_i's busy period starts. Hence J_a cannot cause interference to J_i i.e., $I_{i,a}(v_{x+1}) = 0$. □

Lemma 5.2.6. If $J_i \in \mathcal{A}_x \backslash HP(J_a, v_x)$, such that $SB_i \leq r_a^{max}$, then $I_{i,a}(v_{x+1}) = C_a^{max}$.

Proof. Since J_i is a part of $\mathcal{A}_x$, we know that $LFT_i(v_x) > r_a^{max}$. Since $SB_i \leq r_a^{max} < LFT_i(v_x)$, it means that J_a may be released after the start of the level-i busy window but before J_i's completion (i.e., the end of the level-i busy window). Therefore, in the worst-case, J_a will be released during the level-i busy window and will thus fully execute before J_i's completion (because J_a has a higher priority than J_i). Therefore, the additional worst-case interference suffered by J_i due to J_a is equal to J_a's worst-case execution time C_a^{max}. □

Lemma 5.2.7. Let J_i be a job in $\mathcal{A}_x \setminus HP(J_a, v_x)$, such that SB_i is in $[r_a^{max}, LFT_a v_{x+1}]$, then the interference caused by the newly added job J_a is upper-bounded by $LFT_a(v_{x+1}) - SB_i$.

Proof. Since SB_i contains J_i, we have $SB_i <= r_i^{max}$. We assume that SB_i is in $[r_a^{max}, LFT_a v_{x+1}]$ and hence, we have $r_i^{max} \geq r_a^{max}$. Since J_i releases during the execution of the higher priority job J_a, J_i cannot start execution until J_a completes, i.e., $LFT_a(v_{x+1})$. Therefore, the interference caused by J_a on J_i is upper bounded by $LFT_a(v_{x+1}) - SB_i$.

□

Theorem 5.2.8. Equation 5.7 results in an upper bound worst case interference caused by a newly added job J_a on a lower priority job J_i in $\mathcal{A}_x$.

Proof. From Lemma 5.2.6, we know that given that when the latest release time r_a^{max} of J_a is smaller than or equal to the start of the level-i busy window, then $I_{i,a}(v_{x+1}) = C_a^{max}$.

Similarly, from Lemma 5.2.5, we know that when the latest release time r_a^{max} of J_a is larger than or equal to the level-i busy period SB_i, then the interference it causes on J_i is given by $I_a(v_{x+1}) = 0$.

From Lemma 5.2.7, we have that if $r_a^{max} < SB_i < LFT_a(v_{x+1})$,i.e., in the cases that are not covered by the special cases above, then $I_{i,a}(v_{x+1})$ is an upper bounded by $LFT_a(v_{x+1}) - SB_i$. Now, note that if $SB_i \leq r_a^{max}$ then, $LFT_a(v_{x+1}) - SB_i \geq LFT_a(v_{x+1}) - r_a^{max} \geq C_a^{max}$ (from Equations 5.2 and 5.5) . Therefore, Equation 5.7 returns C_a^{max} as it should according to Lemma 5.2.6. Similarly, if $SB_i \geq LFT_a(v_{x+1})$, then $LFT_a(v_{x+1}) - SB_i \leq 0$ and Equation 5.7, returns 0 as expected according to Lemma 5.2.5. In all other cases, Equation 5.7, returns $LFT_a(v_{x+1}) - SB_i$. Therefore, Equation 5.7 is an upper bound on $I_{i,a}(v_{x+1})$. □

The latest finish time of the job J_i can now be updated to account for the interference generated by J_a as follows:

$$LFT_i(v_{x+1}) = LFT_i(v_x) + I_{i,a}(v_{x+1}) \tag{5.8}$$

Similarly, the *earliest finish time* (EFT) of the lower priority job J_i can be updated as follows:

$$EFT_i(v_{x+1}) = \begin{cases} EFT_i(v_x) + C_a^{min} & \text{if } EFT_i(v_x) > (LFT_a(v_{x+1}) - C_a^{max}) \\ EFT_i(v_x) & otherwise \end{cases} \tag{5.9}$$

The earliest finish time of the job to be updated J_i depends on whether the newly added job J_a preempts it or not.

Lemma 5.2.9. Equation 5.9 provides a lower bound on the finish time of J_i for any execution scenario involving the jobs in $J(v_{x+1})$

Proof. Since adding a job to a preemptive system can only increase the response time of lower priority jobs and because in $EFT_i(v_x)$ is a lower bound on the finish time of J_i for all execution scenarios involving jobs in $J(v_x)$, it directly holds that $EFT_i(v_x)$ is a lower bound on the finish time of J_i for all execution scenarios involving jobs in $J(v_x) \cup J_a$. This proves the second case of Equation 5.9.

Therefore, we focus on proving that $EFT_i(v_{x+1})$ is lower bounded by $EFT_i(v_x) + C_a^{min}$ when $EFT_i(v_x) > LFT_a(v_{x+1}) - C_a^{max}$. To do so, we first state that because $LFT_a(v_x)$ is the latest time at which J_a may finish its execution, it holds true that $LFT_a(v_{x+1}) - C_a^{max}$ is the latest time at which J_a may start executing in any execution scenario involving the jobs in $\mathcal{J}(v_{x+1})$. Therefore, if $EFT_i(v_x) > LFT_a(v_{x+1}) - C_a^{max}$, then J_a must start executing before the earliest finish time of J_i. Associating this to the fact that the earliest release of J_a is no later than that of J_i (because J_i was added before J_a to the schedule abstraction graph), it results that J_a certainly preempts J_i when J_i is released at r_i^{min} (see Figure 5.8 for an illustration of that scenario). Thus, the response time of J_i is certainly increased by the execution time of J_a, i.e., by J_a's best-case execution time C_a^{min} in the best case.

□

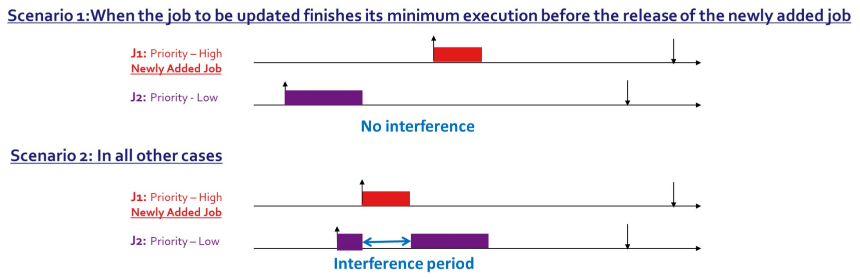

Scenario 1:When the job to be updated finishes its minimum execution before the release of the newly added job
J1: Priority – High
Newly Added Job
J2: Priority - Low
No interference
Scenario 2: In all other cases
J1: Priority – High
Newly Added Job
J2: Priority – Low
Interference period

Figure 5.8: **Reasoning for Equation 5.9. All the release times in this figure depict the earliest release times of the jobs.**

5.2.3 Removing jobs from the graph

Keeping track of all jobs in $\mathcal{J}(v_x)$ in each vertex v_x of the schedule abstraction graph would be memory and computation time inefficient. Instead, we prove that if the response time of a job J_h in the state v_x is no longer affected by the release of the next job added to the graph, then J_h can be removed from the graph. This is so because (1) its worst-case response time is then known (see Lemma 5.2.5) and (2) J_h does not impact the response time of any job in $\mathcal{J} \setminus \mathcal{J}(v_x)$ (see Lemma 5.2.1).
If the earliest release of the newly added job J_a is later than the latest finish time of an already existing job J_h in state v_x, then we can say that no job will preempt J_h anymore and J_h can hence be removed from the graph. This statement has been proved and justified in Lemma 5.2.10

Lemma 5.2.10. Let J_i in state v_x, be a job such that $r_a^{min} \geq LFT_i(v_x)$, and let J_a be the last job added to the schedule abstraction graph. No job added later in the graph will cause interference to J_i.

Proof. By contradiction, let us assume that there is a job J_j added after J_a that causes interference to J_i. Based on the order of adding J_i, J_a and J_j to the schedule abstraction graph, we know that $r_i^{min} \leq r_a^{min} \leq r_j^{min}$. Since a newly added job can only cause interference to J_i if its earliest release is earlier than the latest finish time of J_i, we have that $LFT_i(v_x) > r_j^{min}$. However, by assumption, we also have that $LFT_i(v_x) \leq r_a^{min}$. From these two inequalities, we get that $r_j^{min} < r_a^{min}$ which is in contradiction with the fact that $r_i^{min} \leq r_a^{min} \leq r_j^{min}$ as proven before. Hence, if a newly added job J_a does not cause interference to another job J_i, then no job added later to the graph will cause interference to J_i. □

Lemma 5.2.11. If $D_i \leq T_i$ for all τ_i in τ, and τ is schedulable, then every vertex v_x in the schedule abstraction graph has at most $|\tau|$ jobs in $\mathcal{A}_x$.

Proof. Let J_a be the job added to vertex v_x to create vertex v_{x+1}. By contradiction, let us assume that two jobs of task τ_i, say $J_{i,k}$ and $J_{i,l}$, are in $\mathcal{A}_x$ at the same time. Then, by definition of $\mathcal{A}_x$, we have that $LFT_{i,k}(v_x) > r_a^{min}$ and $LFT_{i,l}(v_x) > r_a^{min}$. Furthermore, because $J_{i,k}$ and $J_{i,l}$ were added to the schedule abstraction graph before J_a, we have $r_{i,k}^{min} \leq r_a^{min}$ and $r_{i,l}^{min} \leq r_a^{min}$. Now, without any loss of generality, let us assume that $J_{i,k}$ was released before $J_{i,l}$ by τ_i. Then, we have $r_{i,k}^{min} \leq r_{i,l}^{min} - T_i$, and thus by the assumption that $D_i \leq T_i$, $r_{i,k}^{min} + D_i \leq r_{i,l}^{min}$. Since we assume that the system is schedulable, we must have that $J_{i,k}$ must complete before its deadline and thus $LFT_{i,k}(v_x) \leq r_{i,l}^{min}$. Using the fact that $r_{i,l}^{min} \leq r_a^{max}$, we have $LFT_{i,k}(v_x) < r_a^{min}$ which is a contradiction with the assumption that $J_{i,k} \in \mathcal{A}_x$.

This proves that each task in τ has at most one job in $\mathcal{A}_x$ and hence, $\mathcal{A}_x$ contains at most $|\tau|$ jobs. □

Lemma 5.2.12. If τ is schedulable, then vertex v_x has at most $\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil$ jobs in $\mathcal{A}_x$.

Proof. Let J_a be the job added to vertex v_x to create vertex v_{x+1}. By contradiction, let us assume that k jobs of a task τ_i are in $\mathcal{A}_x$ such that $k > \lceil D_i/T_i \rceil$. Let the jobs of τ_i in $\mathcal{A}_x$ range from $J_{i,p}$ to $J_{i,p+k}$. By the definition of $\mathcal{A}_x$, we have that $LFT_{i,p} > r_a^{min}$ and $LFT_{i,p+k} > r_a^{min}$. Since, all the jobs from $J_{i,p}$ to J_{p+k} have already been added to the graph as they belong in $\mathcal{A}_x$, we have $r_{i,p}^{min} \leq r_a^{min}$ and $r_{i,p+k}^{min} \leq r_a^{min}$. By assuming that $J_{i,p}$ is the job of τ_i in v_x with the earliest release and $J_{i,p+k}$ is the job of τ_i in v_x with the latest release, we have $r_{i,p}^{min} \leq r_{i,p+k}^{min} - (k \times T_i)$. Since we have assumed that $k > \lceil D_i/T_i \rceil$, we generalize and say $k \times T_i > D_i$. Therefore, we have $r_{i,p}^{min} + D_i \leq r_{i,p+k}^{min}$. Since, we assume the system is schedulable, $LFT_{i,p}(v_x) \leq r_{i,p+k}^{min}$ as job $J_{i,p}$ must finish before its deadline. Using the fact that $r_{i,p}^{min} \leq r_a^{max}$ which is a contradiction with the assumption that $J_{i,p} \in \mathcal{A}_x$.

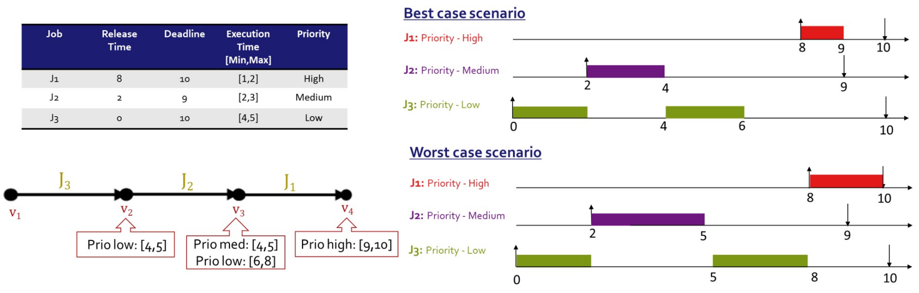
Hence, we can say that there are at most $\lceil D_i/T_i \rceil$ jobs af a task τ_i in $\mathcal{A}_x$ and hence, the maximum number of jobs in $\mathcal{A}_x$ is $\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil$. $\square$

5.2.4 Worst-case response time

Formally written, we have $WCRT(J_i) = max_{v_x \in V} LFT_i(v_x) - r_i^{min}$. Thus, the worst-case response time of a job is recorded when a job is either removed from the graph or the graph has reached its last state and there are no more jobs to be added. Indeed, when a job J_i is removed from the graph during the creation of an intermediate state, the latest finish time of J_i at that point minus its earliest release is an upper bound the worst case response time of the job (since LFT_i will not increase any more after that point as proven by Lemma 5.2.5). For all the jobs that are still present in the last state of the graph, the latest finish time of each job in the last state would be their respective worst-case response time. For a graph built from a jobset $\mathcal{J}$, we can say that $\mathcal{J}$ is schedulable if the worst-case response time of all the jobs in $\mathcal{J}$ is smaller or equal to their respective deadlines.

5.2.5 A working example of the new abstraction graph

In order to understand the working of the concepts of creating a state, adding, updating and removing jobs as have been discussed in Subsections 5.2.1, 5.2.2, 5.2.3, the working of a simple example has been discussed in this section. Figure 5.9 shows a jobset with execution time variation and also depicts what the graph would look like for this jobset.



Job	Release Time	Deadline	Execution Time [Min,Max]	Priority
J1	8	10	[1,2]	High
J2	2	9	[2,3]	Medium
J3	0	10	[4,5]	Low

Figure 5.9: **Working example of the new abstraction graph**

In Figure 5.9, among the jobs in the jobset the job to release the earliest if J_3 and hence it is the job that has been added to the graph the first. Since the system so far only has J_3, it is the highest priority job in the state and hence the earliest and latest finish times of the job would follow Equations 5.1 and 5.2. Once the state v_2 has been formed, among the remaining jobs J_1 and J_2, the job that will be added to the graph next would be J_2 as it has an earlier release. When this job J_2 is added to the new state of v_3, it is the highest priority job in the state and hence the earliest and latest finish times would follow Equations 5.1 and 5.2, while updation of the finish intervals of J_3 would follow Equations 5.9 and 5.8. The last remaining job would then be added to the new state of v_4. While adding this new job, it can be seen that the release time of J_1 is larger than the latest finish times of both J_2 and J_3 and hence J_1 can never cause any interference to both the existing jobs. These jobs are therefore removed from the forthcoming states. In order to check if the jobset is schedulable or not the worst-case response time of each job is checked to see if it is smaller or equal to their respective deadlines. Since in this example we can see that this condition holds for all the jobs in the jobset, this jobset is said to be schedulable.

5.2.6 Graph generation algorithm

Now that we have seen a working example of how the graph is built we discuss the graph generation algorithm that is used to generate the graphs as seen in Figure 5.9. The graph generation algorithm starts with an input of a job set which consists of all the jobs in an observation window whose length can be calculated *a priori*. With this jobset as input, the algorithm produces a graph G as output. The jobs are first sorted in ascending order based on their earliest release times. Since the graph currently has no states, it is initialized by adding state v_0 which is an empty state (i.e., assuming that no job has been executed yet). Then one by one as a new job is added to the system, a new state is added to the graph.

For a job that is newly added to the graph, the finish times intervals are calculated based on the relative priority of the job with respect to the jobs in v_x. If the newly added job is the highest priority job, then the finish times are calculated using Equations 5.1 and 5.2, else the finish times are calculated using Equations 5.6 and 5.5. Then every job in state v_x is considered. It is first checked if the job is carried on further or whether it is removed from the graph. This can be done by checking the condition $LFT_j(v_x) < r_i^{min}$. If this condition holds true, then the job from v_x can be removed from the graph else the job is prepared to be added to the new state.

If it has been decided that a job from v_x is to be added to v_{x+1}, the next step is to decide whether the finish time intervals have to be updated or not. If the job in contention is of a higher priority than the newly added job, then the job can be added to v_{x+1} as is. Otherwise, the finish time intervals have to be updated using Equations 5.9 and 5.8. Once the state v_{x+1} has been built it is then added to the graph. This procedure is repeated for all the jobs in the jobset.

While building this graph, from both the example in Figure 5.9 and Algorithm 2, we can see that this solution does not build a graph that branches. Due to this property of the new schedule abstraction graph, we do not need to employ merge techniques as state space explosion problems have been completely eradicated.

Algorithm 2: Graph generation algorithm

```
   Input: Jobset $\mathcal{J}$
   Output: Graph $G$
 1 sort($\mathcal{J}$) based on $r_i^{min}$
 2 Initialize $G$ by adding $v_0 = \emptyset$
 3 for $J_i$ in $\mathcal{J}$ do
 4     Create new state $v_{x+1}$
 5     State $v_x$ is the last state that as added to $G$
 6     if $J_i$ is the highest priority job in $v_x$ then
 7         $EFT_i(v_{x+1}) \leftarrow$ Equation 5.1
 8         $LFT_i(v_{x+1}) \leftarrow$ Equation 5.2
 9     else
10         $EFT_i(v_{x+1}) \leftarrow$ Equation 5.6
11         $LFT_i(v_{x+1}) \leftarrow$ Equation 5.5
12     Add $J_i$ to state $v_{x+1}$
13     for all $J_j$ in state $v_x$ do
           /* If the job in the previous state has a smaller latest finish
              time than the earliest release of the newly added job        */
14         if $LFT_j(v_x) < r_i^{min}$ then
15             $J_j$ is not carried forward to $v_{x+1}$
           /* If the job in the previous state has a larger latest finish
              time than the earliest release of the newly added job        */
16         else
               /* If the job in the previous state has a higher priority than
                  the newly added job                                      */
17             if $p_i > p_j$ then
18                 Add $J_j$ to $v_{x+1}$
               /* If the job in the previous state has a lower priority than
                  the newly added job                                      */
19             else
20                 $EFT_j(v_{x+1}) \leftarrow$ Equation 5.9
21                 $LFT_j(v_{x+1}) \leftarrow$ Equation 5.8
22                 Add $J_j$ to $v_{x+1}$
23     Add $v_{x+1}$ to $G$
```

Theorem 5.2.13. The time and space complexity to build any vertex v_x in the schedule abstraction graph is $O((\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil)^2)$ and $O(\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil)$, respectively.

Proof. Lemma 5.2.12 proves that there are at most $\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil$ jobs in $\mathcal{A}_x$. Hence, the space complexity of v_x is bounded by $\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil$. Moreover, Algorithm 2 goes at most once through each job in $\mathcal{A}_x$. For each node v_x, Algorithm 1 also goes at most once through each job in $\mathcal{A}_x$ and for each job may have to perform at most $\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil$ operations (Lines 13 and 16). Thus, the time complexity of Algorithm 1 is bounded by $O(\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil^2)$. Since Algorithm 2, calls Algorithm 1 for every job in v_x, the time complexity is bounded by $O((\sum_{\tau_i \in \tau} \lceil D_i/T_i \rceil)^2)$. $\square$

Corollary 5.2.13.1. If $D_i \leq T_i$ for all tasks in τ, then the time and space complexity of Algorithm 2 is $O(|\tau|^2)$ and $O(|\tau|)$, respectively.

Proof. The proof is a direct consequence of Theorem 5.2.13, since Algorithm 1 and Algorithm 2 are bounded by $\sum_{tau_i \in \tau} \lceil D_i/T_i \rceil$ and $(\sum_{tau_i \in \tau} \lceil D_i/T_i \rceil)^2$, respectively. From Lemma 5.2.11, we know that when $D_i \leq T_i$,$\sum_{tau_i \in \tau} \lceil D_i/T_i \rceil = |\tau|$. hence, the time complexity of Algorithm 1 is $O(|\tau|)$. Therefore, the time complexity of Algorithm 2 is $O(|\tau|^2)$. $\square$

5.3 Evaluation

In order to understand how effectively the newly developed solution performs against the state of the art, experiments were conducted. Algorithm 2 was implemented as a python program and this analysis was tested against the state of the art using synthetically generated tasksets. The state of the art that was tested were both exact and sufficient tests.

5.3.1 Comparison baseline selection

The first stage of experiments was performed on the schedule abstraction graph built to handle tasksets scheduled on uniprocessor systems. The state of the art chosen for this methodology can be separated into three categories depending on their ability to handle jitter and offsets. The first category of solutions are those that do not handle both jitter and offset. In this category, we have the solution by Audsley et al. [2] which is an exact solution that helps analyse preemptive tasks on a uniprocessor platform. Two other tests were also chosen, namely the Park test by Park et al. [25] and the *distance constrained taskset* [DCT] algorithm by Han et al. [17]. The second category has a solution by Goossens [18] that can handle tasksets having various offsets. This solution too is exact in nature. The final category resembles the solution presented in this paper as close as possible and looks at handling tasks that contain both release time jitter and offsets. The solution by Redell et al. [26] is an exact solution that handles both jitter and offsets.

5.3.2 Synthetic taskset generation

In order to be able to perform experiments, tasksets were synthetically generated using the Emberson and Davis tool [13]. The tasksets were generated such that the tasks per taskset ranged from 3 tasks to 18 tasks with a step of 3. The period of the taksets followed either a uniform or a log-uniform distribution and ranged from 10 to 10,000 time units, and the total utilization of the system $U = \sum_{\tau_i} \in C_i/T_i$ was equal to a predefined value. When experiments for task sets with release jitter were performed, each task was assigned a release jitter proportional to its period (i.e., the release jitter was set to 5% of the task period). Similarly, when experiences accounting for release offsets were performed, each task was also assigned an offset picked as a random value smaller than or equal to the period of the task. Since all the solutions stated in Section 5.3.1, provide only the worst-case response time, the execution time variation is not considered. This is because in a worst case analysis, all jobs always execute for C_i^{max} time units. A rate-monotonic priority assignment, where a shorter period results in a higher priority, was designated to each task as the deadlines of the tasks was assumed to be equal to the period of the task i.e., $D_i = T_i$. For each combination of parameters, 100 different tasksets were generated.

5.3.3 Experiments

Two types of experiments were performed. First, the utilization is varied for constant values of the number of tasks in a taskset. Then in the second set of experiments the utilization is kept constant while the number fo tasksets is varied is presented.

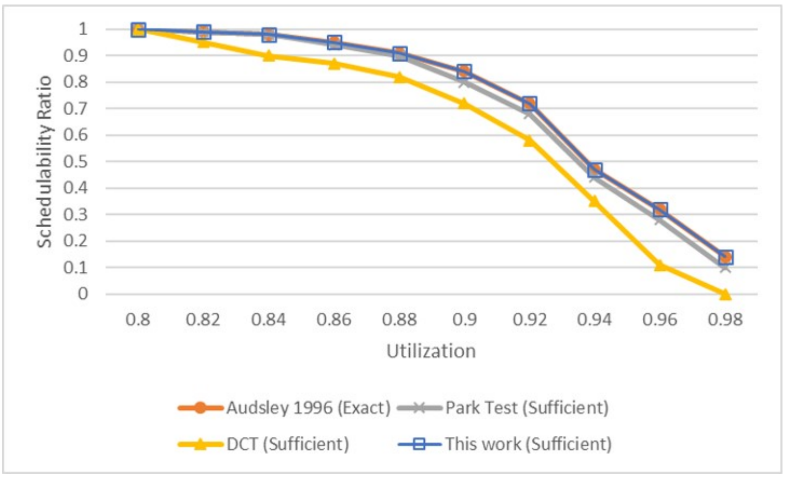
Experiments by varying the total utilization U

In order to compare Algorithm 2 against the state of the art, for each setup, we plot the *schedulability ratio* i.e., the ratio of schedulable tasksets to the number of tasksets that were synthetically generated. With such a plot the best analysis is the analysis that shows the highest schedulability ratio.

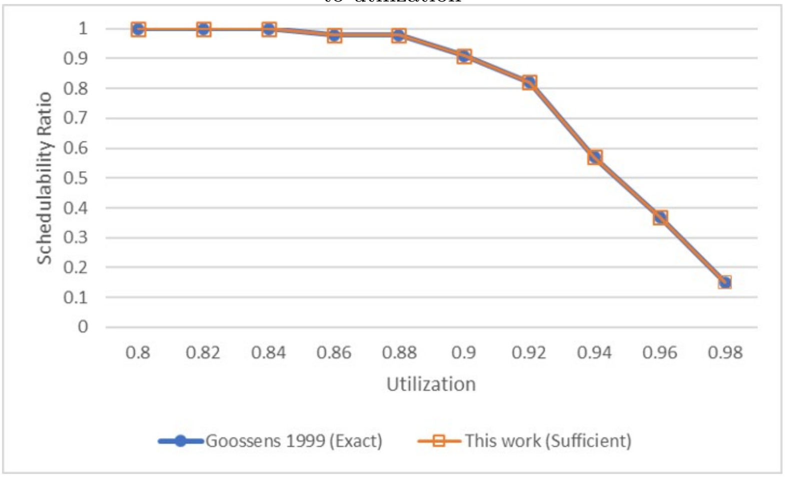
Figure 5.10 shows the comparison of Algorithm 2 with the three categories of the state of the art. All three graphs represent tasksets that have log-uniform periods, such that each taskset is composed of 9 tasks. Figure 5.10(a) compares Algorithm 2 with the solution by Audsley et al. [2], Park test [25] and *distance constrained taskset*[DCT] algorithm by Han et al. [17]. From this graph, we can see that Algorithm 2 overlaps the exact solution by Audsley et al. [2]. Figure 5.10(b) compares Algorithm 2 with the solution by Goossens [18] and from this graph too we can see that the solution of both the solutions overlap. Figure 5.10(c) compares Algorithm 2 with the solution by Redell et al. [26] and from this graph too we can see that the solution of both the solutions overlap. In all three solutions Algorithm 2 behaves just as well as the exact solutions in all three categories.

Experiments by varying the number of tasks in a taskset

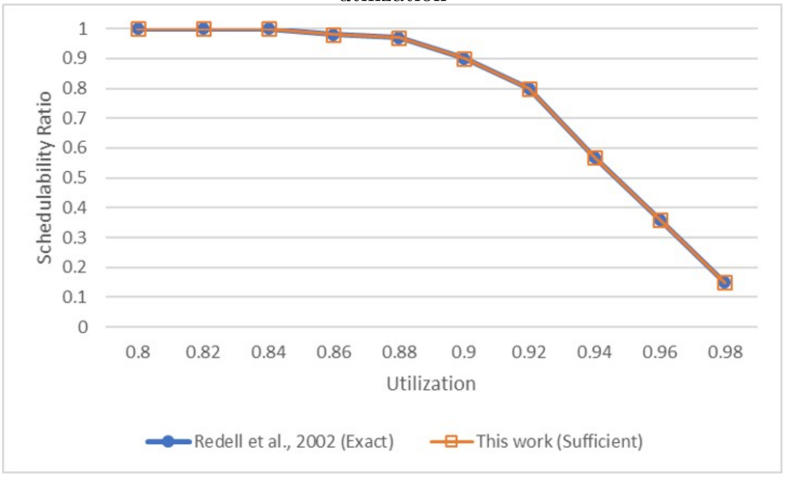
In this set of experiments, the schedulability ratio is a plot against varying values of the total number of tasks in a taskset for each category. As the number of tasks in a taskset increases, the schedulability decreases, as can be seen in Figure 5.11. All three graphs represent tasksets that have log-uniform periods, such that all the tasksets have a utilization of 0.94. Here, too similar as before we see that in all three setups, our solution overlaps with the exact solution.



(a)Evaluation results of solutions that do not deal with jitter and offset with respect to utilization

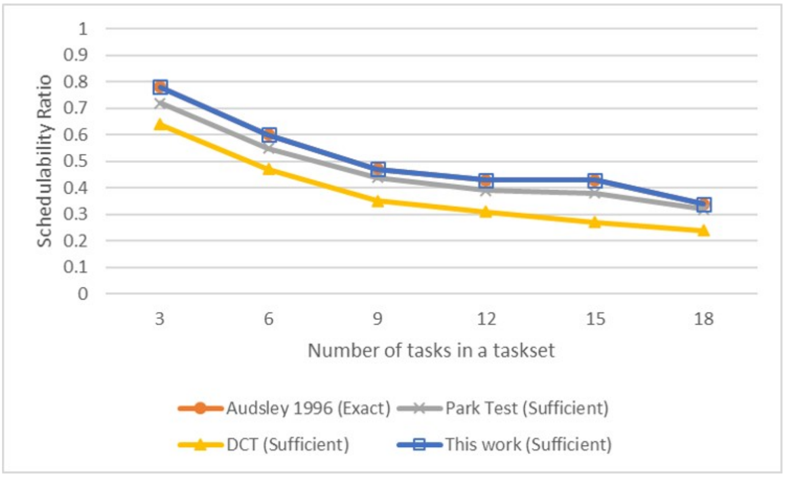


(b)Evaluation results of solutions that only deals with offsets with respect to utilization

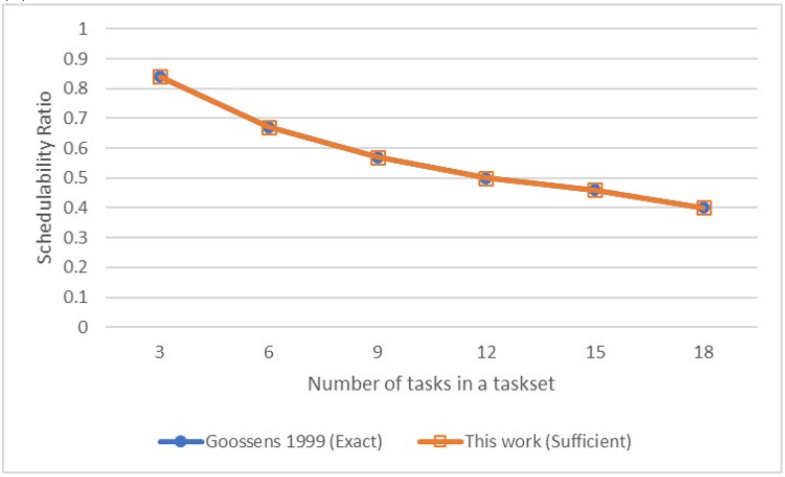


Evaluation results of solutions that deal with both jitter and offset with respect to utilization

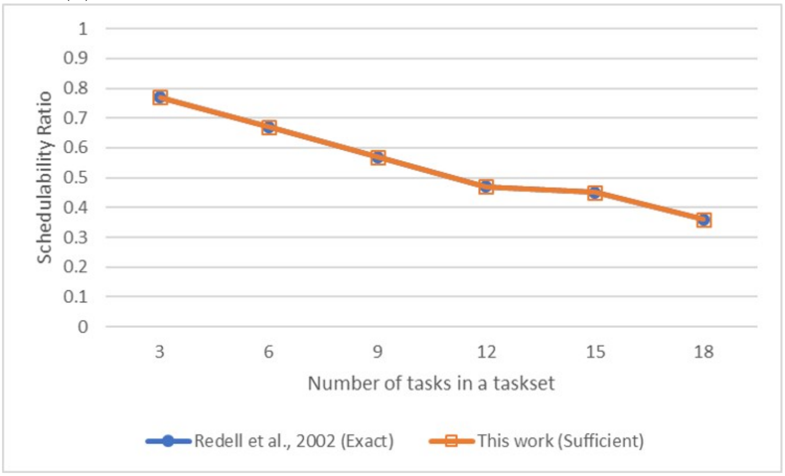
Figure 5.10: **Evaluation results of solutions where the utilization is varied over a fixed number of tasks in a taskset**



(a) Evaluation results of solutions that do not deal with jitter and offset.



(b) Evaluation results of solutions that only deal with offsets.



(c) Evaluation results of solutions that deal with both jitter and offset.

Figure 5.11: **Evaluation results of solutions where the number of tasks in a taskset is varied over a fixed utilization**

Chapter 6

Response-time analysis for multi-core platforms

The solution that has been described so far deals with scheduling preemptive tasks on uniprocessor systems. In order to extend this solution to multiprocessor systems with m cores, a modification to the worst case interference was performed. The structure and terminology of the graph remains the same. The only difference is the methodology used to calculate the interference caused by higher priority jobs. In order to further simplify the extension, only the worst case scenarios are considered and hence the earliest finish time is not a timing property that is maintained by the graph.

6.1 Modified worst case interference

In the solution for single processor system, there were two instances where the worst-case interference had to be calculated. (1) A job of a lower priority was added to the graph and the worst case interference was calculated using equation 5.3. (2) When updating the intervals of jobs the worst case interference of a job was calculated using equation 5.7. For the multiprocessor solution, in both cases, this calculated interference is divided by the number of processors(m) in order to give us an upper bound on the interference in an m core system.

Let the worst-case interference of J_a scheduled on a multiprocessor platform, when it is added to a new state v_{x+1} is denoted by $MI_a(v_{x+1})$. For calculating the worst-case interference, while adding a job J_a that has a higher priority than all the jobs in $\mathcal{A}_x$, i.e., $HP(J_a, v_x) = \emptyset$, Equation 5.3 is modified as follows:

$$MI_a(v_{x+1}) \leq \begin{cases} 0 & \text{if } r_a^{max} \geq LFT_i(v_x), \forall J_i | J_i \in HP(J_a, v_x) \\ \left\{\sum_{J_i | J_i \in HP(J_a, v_x)} C_i^{max}\right\} / m & \text{if } r_a^{max} \leq r_i^{max}, \forall J_i | J_i \in HP(J_a, v_x) \\ \{LFT_b(v_x) - r_a^{max}\} & \text{otherwise} \end{cases} \tag{6.1}$$

The worst-case additional interference that a job J_a can cause to another lower priority job J_i scheduled on a multiprocessor platform needs to be calculated. Unlike the extension above that extends the uniprocessor equation for calculating the interference that a new job J_a faces, directly to the multiprocessor system the Equation 5.7 cannot be modified directly. This is because there is a notion of the *the start of priority level-i busy window* (SB_i), which is not simple to extend to multiprocessor systems as these priority levels are core dependent. Since the idle time is core-specific, if this idle time is to be monitored for calculating interference, then which jobs go to which core also needs to be monitored. Since this would leave us with an exhaustive search problem, we simplify Equation 6.1, and adapt it to compute the interference a lower priority job $J_i \in \mathcal{A}_x \setminus HP(J_a, v_x)$ as follows:

$$MI_{i,a}(v_{x+1}) \leq \begin{cases} 0 & \text{if } r_i^{max} \geq LFT_a(v_{x+1}) \\ C_i^{max}/m & \text{if } r_i^{max} \leq r_a^{max} \\ \{LFT_k(v_{x+1}) - r_i^{max}\}/m & \text{otherwise} \end{cases} \tag{6.2}$$

where $LFT_k(v_x)$ is the latest finish time of a job J_k such that J_k is the lowest priority job in v_{x+1} that has a higher priority than J_i , i.e., J_k is given by Equation

$$J_k = \underset{J_u \in HP(J_i, v_{x+1})}{\text{argmax}} \quad p_u \tag{6.3}$$

We now prove that the extension of the uniprocessor equations in Equations 6.1 and 6.2, divided by the number of cores m provides a safe upper bound on he interference.

Lemma 6.1.1. On a multiprocessor platform, an upper bound on the worst case interference can be obtained by $MI_i(v_{x+1}) = I_i(v_{x+1})/m$ where, m is the number of processors and $I_i(v_{x+1})$ is the interference calculated on a uniprocessor platform.

Proof. $I_i(v_{x+1})$ provides an upper bound on the interference because if $m = 1$, then $MI_i(v_{x+1}) = I_i(v_{x+1})$. When $m > 1$, $I_i(v_{x+1})$ is distributed among m cores. If the distribution of the interference was not even among all the cores, a particular core might have less interference scheduled on it that the other intervals. The worst case interference is when all the cores have $I_i(v_{x+1})/m$ units of interference. When a core has an interference that is greater than $I_i(v_{x+1})/m$ by k units, there would be another core, that is less $I_i(v_{x+1})/m$ by k units. This happens as the total interference $I_i(v_{x+1})$ always remain the same. Hence, $I_i(v_{x+1})/m$ gives an upperbound on the interference. □

An example of Lemma 6.1.1 is seen in Figure 6.1. If a job faces an interference of 9 units on a uniprocessor system the processor availability is as it has been depicted in the figure. But when this interference is scheduled on a multiprocessor system, it gets divided by the number of cores to form the upper bound on the interference. This can be considered as the upper bound because if the interference was not evenly spread and one processor had more than 3 units of interference, then there will definitely be another processor that has less than 3 units of interference scheduled on it. Hence, interference divided by the number of cores m gives an upper bound on the interference on a multicore platform.

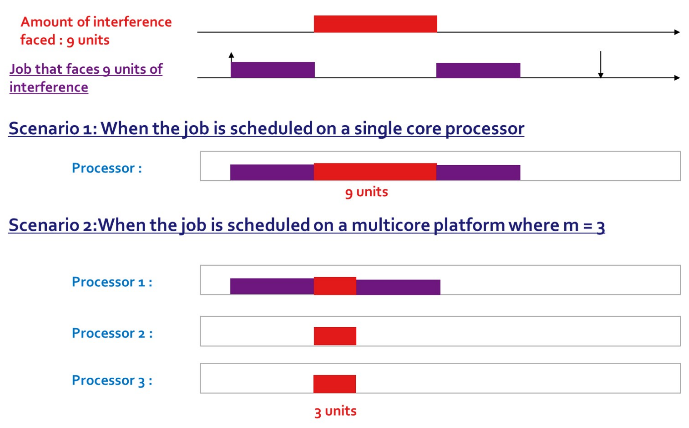

Amount of interference faced : 9 units
Job that faces 9 units of interference
Scenario 1: When the job is scheduled on a single core processor
Processor :
9 units
Scenario 2:When the job is scheduled on a multicore platform where m = 3
Processor 1 :
Processor 2 :
Processor 3 :
3 units

Figure 6.1: **Visualization of the interference on a single core platform and a multicore platform**

6.2 Graph generation algorithm on a multiprocessor platform

Now that the worst case interference has been calculated, the extension of the solution to multiprocessor systems is hence simplified. The algorithm to generate a graph for multiprocessor systems is given by Algorithm 3. When calculating the latest finish time in the algorithm, the modified worst case interference as in Equations 6.1 and 6.2 are used instead.

6.3 Evaluation

In order to understand how effectively the extended solution for multiprocessor performs against the state of the art, experiments were conducted. Algorithm 3 was implemented as a python program and this analysis was tested against the state of the art using synthetically generated tasksets.

6.3.1 Comparison baseline selection

The solutions that use finite state machines to build their analysis have not been evaluated. This is because, the finite state machine solutions are not scalable and even the smallest tasksets generated for the experiments would have been too large for those tools to handle. Hence, the experiments in this section compare how well our solution performs when compared to a solution that extends the response time analysis by Audsley et al. [2]. As seen in Section 3.2, there have been many solutions that extend the work of Audsley et al. [2], and since all the solutions perform very similarly and have very slight variation within their results (refer Sun et al.[27]), testing against just one of those solutions will help us understand how the solution developed in this thesis performs. The chosen baseline was the solution by Guan et al. [16] as this solution has been regarded as the state-of-the-art in Sun et al [27] for global fixed priority scheduling.

Algorithm 3: Graph generation algorithm on a multiprocessor platform

```
   Input: Jobset $\mathcal{J}$
   Output: Graph $G$
 1 sort($\mathcal{J}$) based on $r_i^{min}$
 2 Initialize $G$ by adding $v_0 = \emptyset$
 3 for $J_i$ in $\mathcal{J}$ do
 4     Create new state $v_{x+1}$
 5     State $v_x$ is the last state that as added to $G$
 6     if $J_i$ is the highest priority job in $v_x$ then
 7         $LFT_i(v_{x+1}) = r_i^{max} + C_i max$
 8     else
 9         $I_i(v_{x+1}) \leftarrow$ Equation 6.1
10         $LFT_i(v_{x+1}) = r_i^{max} + I_i(v_{x+1}) + C_i^{max}$
11     Add $J_i$ to state $v_{x+1}$
12     for all $J_j$ in state $v_x$ do
           /* If the job in the previous state has a smaller latest finish
              time than the earliest release of the newly added job        */
13         if $LFT_j(v_x) < r_i^{min}$ then
14             $J_j$ is not carried forward to $v_{x+1}$
           /* If the job in the previous state has a larger latest finish
              time than the earliest release of the newly added job        */
15         else
               /* If the job in the previous state has a higher priority than
                  the newly added job                                      */
16             if $p_i > p_j$ then
17                 Add $J_j$ to $v_{x+1}$
               /* If the job in the previous state has a lower priority than
                  the newly added job                                      */
18             else
19                 $I_{j,i}(v_{x+1}) \leftarrow$ Equation 6.2
20                 $LFT_j(v_{x+1}) = LFT_j(v_x) + I_{j,i}(v_{x+1})$
21                 Add $J_j$ to $v_{x+1}$
22     Add $v_{x+1}$ to $G$
```

6.3.2 Synthetic taskset generation

Similar to the experiments for the single processor solution, tasksets were synthetically generated using the Emberson and Davis tool [13]. The tasksets were generated such that the number of cores m ranged from 2 to 8 cores. The period of the taksets followed a log-uniform distribution and ranged from 10 to 10,000 time units, and the total utilization of the system $U = \left\{\sum_{\tau_i} \in C_i/T_i\right\}/m$ was equal to a predefined value. The number of tasks in a taskset was determined based on the number of cores. For each core, the number of tasksets ranged from 1 to 5 times the number of cores i.e, when $m = 2$ the number of tasks in a taskset were within the set of values $[2, 4, 6, 8, 10]$. When experiences accounting for release offsets were performed, each task was also assigned an offset picked as a random value smaller than or equal to the period of the task. Since Gual et al. [16], does not account for jitter, the generated tasks do not have release time jitter. Since Guan et al.[16] only the worst-case analysis, the execution time variation is also not considered. A rate-monotonic priority assignment, where a shorter period results in a higher priority, was designated to each task as the deadlines of the tasks was assumed to be equal to the period of the task i.e., $D_i = T_i$. For each combination of parameters, 100 different tasksets were generated.

6.3.3 Experiments

Three types of experiments were performed. In the experiment presented in Section 6.3.3.1, the utilization is varied for constant values of the number of tasks in a taskset and number of cores. In the experiments presented in Section 6.3.3.2, the utilization and the number of cores is kept constant while the number fo tasksets is varied. Lastly, in the experiments presented in Section 6.3.3.3, the number of cores is varied while the number of tasks per taskset and utilization is kept constant.

6.3.3.1 Experiments by varying total utilization U

We plot the schedulability ratio against varying values of total system utilizations. From Figure 6.2, we see the result of the experiment when the number of cores is set to 4 and the number of tasks in the taskset is set to 5 times the number of cores, i.e., 20 tasks per taskset. We see that Guan et al.[16] performs slightly better than our solution. Both solutions are able to easily identify schedulable tasksets that have a utilization less than 0.3, but any schedulable taskset with a utilization above 0.45 is never found.

6.3.3.2 Experiments by varying the number of tasks in a taskset

We plot the schedulability ratio against varying values of number of tasks in a taskset. From Figure 6.3, we see the result of the experiment when the number of cores is set to 5 and the utilization is 0.25. Here, too we see that Guan et al.[16] performs slightly better than our solution. The schedulabilty ratios of both the solutions start to drop when the number of tasks in a taskset increases.

6.3.3.3 Experiments by varying the number of cores

We plot the schedulability ratio against varying values of number of cores. From Figure 6.4, we see the result of the experiment when the number of tasks in a taskset is twice the number of cores and the utilization is 0.35. Similar to the results in Figure 6.2 and 6.3, we see that Guan et al.[16] performs slightly better than our solution. Both solutions are able to easily identify schedulable tasksets when there are only a few number of cores, but any schedulable taskset scheduled on a platform with around 8 cores is never found.

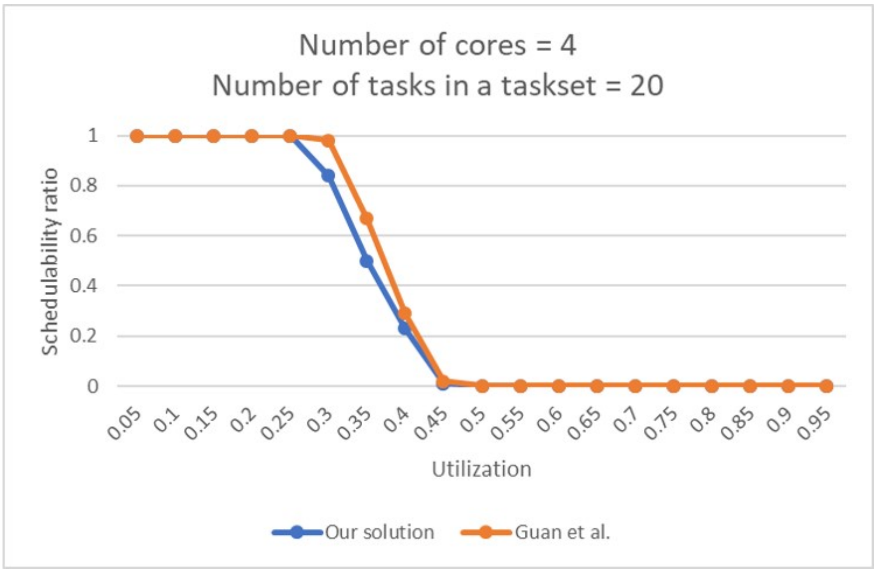


Figure 6.2: **Evaluation results of solutions where the utilization is varied over a fixed number of tasks in a taskset and a fixed number of cores**

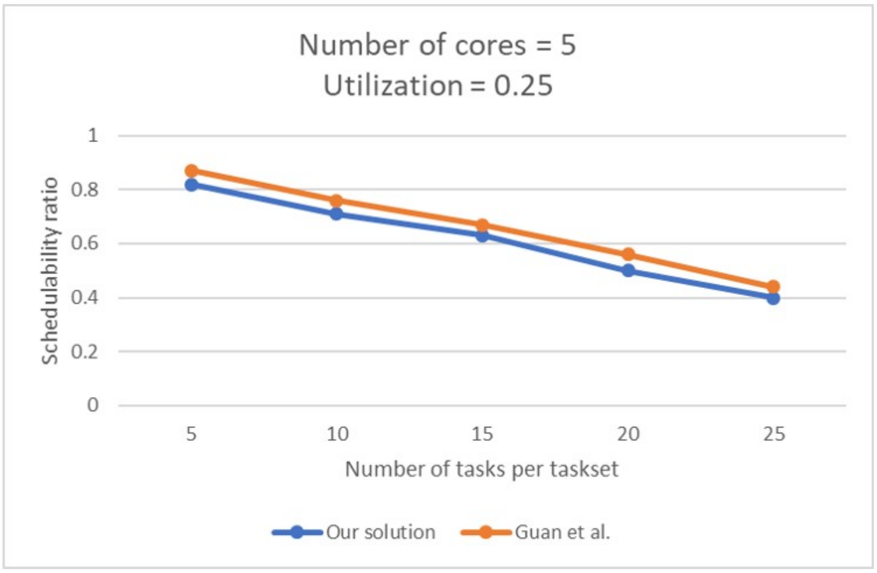


Figure 6.3: **Evaluation results of solutions where the number of tasks in a taskset is varied over a fixed utilization and a fixed number of cores**

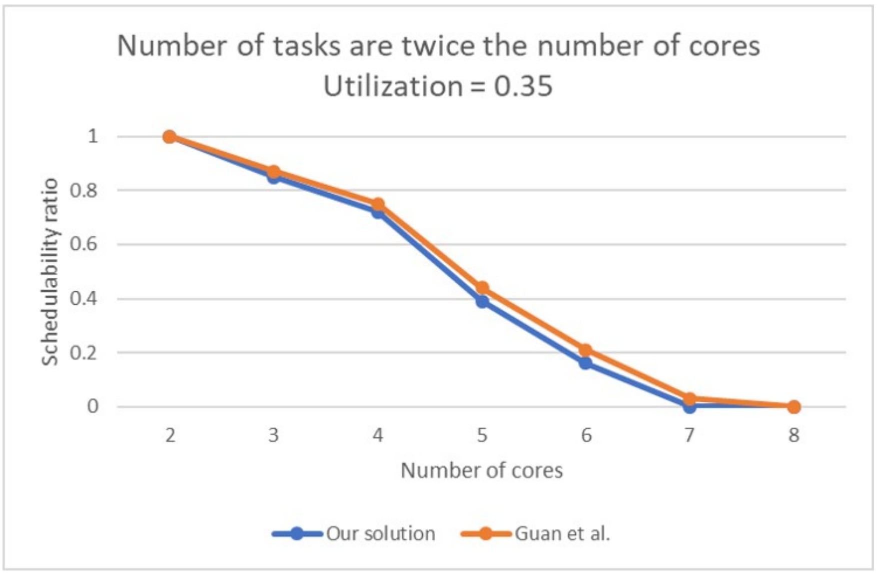


Figure 6.4: **Evaluation results of solutions where the number of cores is varied over a fixed utilization and a fixed number of tasks in a taskset**

Chapter 7

Conclusions

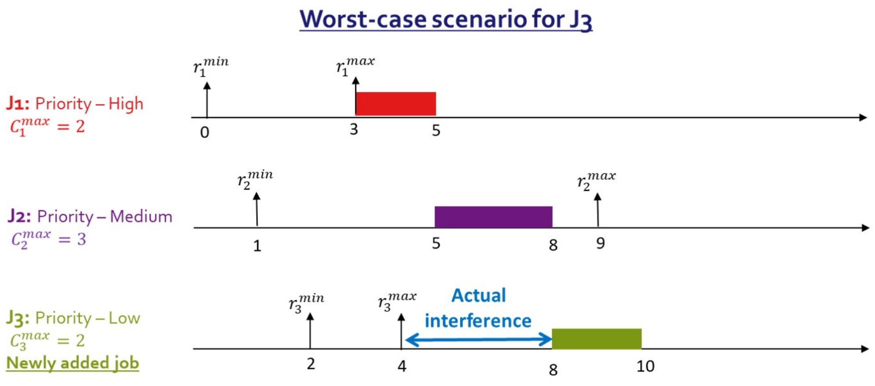
This thesis has introduced a new schedule abstraction graph analysis technique that allows analysing preemptive tasks scheduled under global *job-level fixed-priority*[JLFP] scheduling policies. This solution was built initially for a uniprocessor platform and then was extended to work with global scheduling. The effectiveness of the solution was analyzed by checking the accuracy of the solution and comparing this to the state of the art. The accuracy of the solution was determined by obtaining the schedulability ratio of tasksets with various combination of parameters. From Section 5.3.3, we can see that we perform similar to the state-of-the-art for single core platforms. However, and unfortunately, our solution is slightly more pessimistic than the state-of-the-art for multiprocessor systems. Nevertheless, we have hopes that the accuracy of the analysis for multiprocessor systems can be improved in the future. Indeed, the analysis proposed in this work is of a completely different nature than the related work. The solution proposed in this thesis is the first of its kind and it thus has a lot of space for improvement yet.

7.1 Discussions

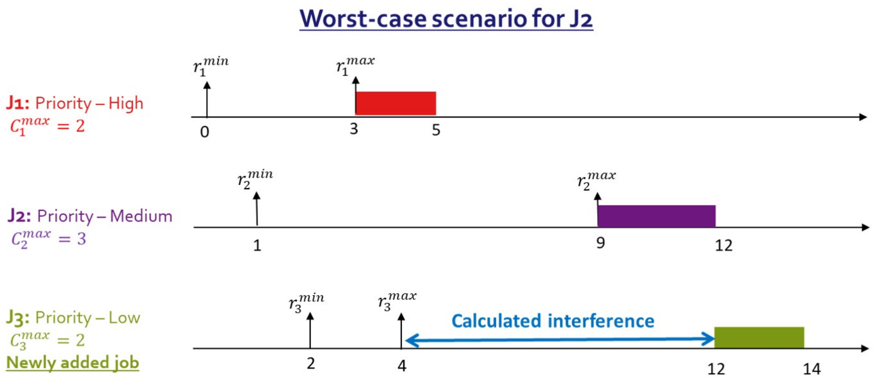
The schedule abstraction graph technique that was developed to analyse preemptive tasks scheduled on a uniprocessor platform performed just as well as the state of the art solutions. This provides a solution that has a lot of scope for an extension as it is highly accurate while completely eradicating the state space explosion problem. However, when extending this solution to multiprocessor systems, we introduce pessimism because the calculation of the interference caused by the higher priority tasks is over approximated. This over-approximation was introduced so that we could keep the extension of the solution very similar to the original solution on uniprocessor platforms. even though this makes the multiprocessor system more pessimistic, we still face pessimism in the uniprocessor solution itself.

7.1.1 Sources of pessimism

The first source of pessimism is found in Equation 5.3. As proven in Lemmas 5.2.1 and 5.2.2, the first and second cases of Equation 5.3 return exact values for the worst-case interference suffered by the newly added job J_a. However, the third case (i.e., when there are two higher priority active jobs J_i and $J_j \in HP(J_a, v_x)$ such that $r_i^{max} < r_a^{max} < r_j^{max}$), computes only an upper bound on the worst-case interference. That is, there might be scenarios where the returned value is pessimistic. A scenario where such an over estimation occurs is depicted in Figure 7.1. Here, J_3 is newly added to a node v_{x+1}. In Figure 7.1(a), we see a schedule that shows one of the possible execution scenarios where J_3 faces the largest possible interference. The release pattern of the jobs in $HP(J_3, v_x)$ i.e., jobs J_1 and J_2, determine the largest interference that J_3 could face. As proven in Theorem 5.2.4, the following two properties determine a possible schedule that leads to the worst case interference of J_3. **1.** Jobs like J_1 that have their latest release time earlier than or equal to the latest release time of J_3, i.e., $r_1^{max} \leq r_3^{max}$, release at their latest release time. **2.** Jobs like J_2 that have their latest release time later than the latest release time of J_3, i.e., $r_2^{max} > r_3^{max}$, are released at r_3^{max}. Hence, according to those properties and referring to Figure 7.1(a), the actual worst-case interference suffered by J_3 is 4 units of time.



(a) Schedule which is the worst-case for J_3



(b) Schedule which is the worst-case for J_2

Figure 7.1: **Overestimation of the interference as computed by Equation 5.3.**

However, according to Equation 5.3 and Lemma 5.2.3, while calculating the interference for J_3, we consider the latest finish time of J_2. This is because if J_3 is the newly added job J_a, then according to Equation 5.4, J_b is J_2. The worst-case scenario for J_2 is when it releases at r_2^{max}. As can be seen in Figure 7.1(b), if J_2 releases at $r_2^{max} = 9$, then it finishes its execution at time 12. Hence the interference calculated according to Equation 5.3, is 8 units of time. Hence, Equation 5.3 assumes that J_3 can only start its execution at time 12, leaving the processor idle from time 5 to 9. In fact, that schedule can never occur in a real system as the scheduler is work-conserving as described in Section 2.2 and will never leave a processor idle when there is a ready job. Hence, Equation 5.3 over-estimates the worst-case interference.

This over-approximation can be reduced by analysing the interference caused to J_a by each higher priority job instead of looking at just the latest finish time of J_b. This would allow for a tighter bound on the upper bound for the interference. This pessimism is more visible when the release times of the jobs are far apart, i.e., the jobs have a large release jitter. Hence, in our experiments where we had a jitter of 5%, this pessimism does not manifest itself as the jitter was small.

7.2 Future work

The obvious main goal of future work is to improve the accuracy of the analysis for multiprocessor systems. In order to do so, the graph could be modified in order to incorporate information about each core and how each core performs independently instead of finding an upperbound on all cores as currently done. This extension, however, is of a very complex nature, since no one so far could find typical execution patterns that yield the WCRT of a job under global multiprocessor scheduling. It is an open problem for the last 30 years and, to the best of our knowledge, no research team seems to be close to finding a solution yet.

Another extension is to introduce the notion of precedence constraints can be added to the system. The system currently assumes that the jobs in the analysis do not have any precedence constraints i.e., jobs are independent of each other and can hence, have any possible job ordering. With precedence constraints, the jobs in a task might have to execute in a particular order. We expect that the solution proposed in this dissertation can easily be extended to support tasks to contain precedence constraints as the precedence constraints only affects the release time of each job (i.e., a job may only be released once all its predecessors are completed).

Bibliography

[1] N Audsley, A Burns, M Richardson, K Tindell, and A J Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[2] N C Audsley, A Burns, R I Davis, K W Tindell, and A J Wellings. Fixed priority scheduling an historical perspective. *Real-Time Systems*, 8(2-3):173–198, March 1995.

[3] T. P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 120–129, 2003.

[4] Theodore P. Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *Proceedings of the 11th International Conference on Principles of Distributed Systems*, OPODIS'07, page 62–75, 2007.

[5] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, page 119–128, 2007.

[6] Geoffrey Nelissen Sebastian Altmeyer Robert I. Davis Benny Akesson, Mitra Nasri. A survey of industry practice in real-time systems. In *IEEE Real-Time Systems Symposium (RTSS), 2020*, 2020 (to appear).

[7] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 149–160, 2007.

[8] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. volume 2005, pages 209– 218, 2005.

[9] Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. *Algorithmica*, 63(4):763–780, August 2012.

[10] Artem Burmyakov, Enrico Bini, and Eduardo Tovar. An exact schedulability test for global fp using state space pruning. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, page 225–234, 2015.

[11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer Publishing Company, Incorporated, 3rd edition, 2011.

[12] Robert I. Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[13] P. Emberson, R. Stafford, and R.I. Davis. Techniques for the synthesis of multiprocessor tasksets. *WATERS'10*, 2010.

[14] Anam Farrukh and Richard West. smartflight: An environmentally-aware adaptive real-time flight management system. In *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, pages 24:1–24:22, 2020.

[15] Gilles Geeraerts, Joël Goossens, and Markus Lindström. Multiprocessor schedulability of arbitrary-deadline sporadic tasks: complexity and antichain algorithm. *Real-Time Systems*, 49(2):171–218, 2013.

[16] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *2009 30th IEEE Real-Time Systems Symposium*, pages 387–397, 2009.

[17] C-. Han and H-. Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Proceedings Real-Time Systems Symposium*, pages 36–45, 1997.

[18] Goossens Joël. Scheduling of hard real-time periodic systems with various kinds of deadline and offset constraints. *PhD thesis, Universite Libre De Bruxelles*, 1999.

[19] Jinkyu Lee and Insik Shin. Limited carry-in technique for real-time multicore scheduling. *Journal of Systems Architecture*, 59(7):372 – 375, 2013.

[20] Robert Leibinger. Software architectures for advanced driver assistance systems (adas). 2015.

[21] 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.

[22] Mitra Nasri and Bjorn Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. pages 12–23, 2017.

[23] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. A Response-Time Analysis for Non-preemptive Job Sets under Global Scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS 2018)*, pages 9:1–9:23, 2018.

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

[25] Moonju Park and Heemin Park. An efficient test method for rate monotonic schedulability. *IEEE Transactions on Computers*, 63:1–1, 2014.

[26] O. Redell and Martin Törngren. Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 164– 172, 2002.

[27] Youcheng Sun and Marco Di Natale. Assessing the pessimism of current multicore global fixed-priority schedulability analysis. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 575–583, 2018.

[28] Youcheng Sun, Giuseppe Lipari, Nan Guan, and Wang yi. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *RTCSA 2014 - 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2014.