Delft University of Technology
Master of Science Thesis in Embedded Systems

Schedulability analysis of limited-preemptive moldable gang tasks

Joan Marcè i Igual

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

Embedded Networked Systems

TU Delft Delft University of Technology

Schedulability analysis of limited-preemptive moldable gang tasks

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

Joan Marcè i Igual

23rd of August, 2020

Author
Joan Marcè i Igual

Title
Schedulability analysis of limited-preemptive moldable gang tasks
MSc Presentation Date
27th of August, 2020

Graduation Committee

Prof. dr. ir. Koen Langendoen (chairman)	Delft University of Technology
Dr. Mitra Nasri (co-supervisor)	Delft University of Technology
Dr. ir. Geoffrey Nelissen (supervisor)	Eindhoven University of Technology

The work presented in this thesis has lead to a paper which ahs been submitted to the 2020 Real-Time Systems Symposium (RTSS) for publication, pending peer-review.

Abstract

Gang scheduling, has long been adopted by the high-performance computing community as a way to reduce the synchronization overhead between related threads. Gang schedulling allows for several threads to execute in lock steps without suffering from long busy-wait periods or be penalised by large context-switch overheads. If several threads use the same data, it also reduces the number of memory transactions by allowing the program to load those data only once for all threads rather than once per thread. To avoid reloading large amount of data after each preemption and hence incur large execution-time overheads, in this work, we assume that the tasks adhere to a limited-preemptive execution model. We further assume that each gang task is moldable, that is, it has a minimum and a maximum number of cores on which it may be executed. The actual execution time of a job depends then on the number of cores allocated by the scheduler at run-tiem. In this work, we consider the case for which tasks are scheduled according to a global job-level fixed priority scheduling algorithm, and present a worst-case response time analysis for limited-preemptive moldable gang tasks. Additionally, we propose a new scheduling policy to improve the schedulability of moldable gang tasks.

“Blessed are those who find wisdom, those who gain understanding, for she is more profitable than silver and yields better returns than gold.” — Proverbs 3:13–14

Preface

First, I would like to thank my direct supervisor Dr. ir. Geoffrey Nelissen from TU/e. His professional advice and technical knowledge helped me walk through all the steps of this thesis, he also reviewed my work and patiently answered all my questions, sometimes even late during the night. I would also like to thank my co-supervisor Dr. Mitra Nasri from TU/e, who reviewed all my findings and pointed me in the right direction when I was lost. I must express express my very profound gratitude to my family for providing me with support and continuous encouragement through my years of study and through the process of writing this thesis. Finally I would like to thank God for making me a curious being and providing me with this opportunity.

Joan Marcè i Igual

Delft, The Netherlands
23rd of August, 2020

Contents

Chapter 1

Introduction

Nowadays, many safety-critical real-time systems found in the aviation, railway or automotive industry are controlled by the use of computers. The safety of such real-time systems depends not only on the logical or functional correctness but also on temporal correctness, i.e., the ability to satisfy all timing requirements, which are typically described by deadlines, of the tasks in the system during the system's lifetime. [1].

As finding an optimal schedule of such tasks has been shown to be computationally intractable [1]. Multiple scheduling policies have been proposed in order to find simpler solutions to the scheduling problem. The job-level fixed-priority (JLFP) scheduling policy allows to define a different priority to every job that a task can release. This policy can model other scheduling policies such as earliest-deadline first (EDF) or deadline monotonic (DM).

In the last decade, multi-core processors have been gaining a lot of attention in real-time systems since they deliver more computing power with lower power consumption. There are three main execution models to run a parallel real-time application on a multi-core platform [2, 3]: thread-based, gang scheduling and federated scheduling models. The later two models are special cases of thread-based execution. In the gang model, multiple parallel threads are grouped and executed together as a "gang". That is, when a gang task is executed, a given set of cores is reserved exclusively to execute the task hence allowing multiple threads to execute simultaneously. Federated scheduling is a particular case of gang scheduling where the cores are dedicated to the threads during the system's lifetime, an example of the different models can be seen in Figure 1.1. While thread-based execution and federated scheduling have their use cases, thread-based execution can have a lot of variability in the execution time when precedence constraints are involved and federated scheduling can underuse the platform due to its constraint of reserving cores forever. That is why in this thesis, we will focus on the gang scheduling model.

Gang scheduling was initially adopted by the high-performance computing (HPC) community in the early 80s [2]. The idea was to reduce the overhead caused by the synchronisation of multiple related threads and to optimise the access to shared data in data-intensive computations [4]. As all the threads start

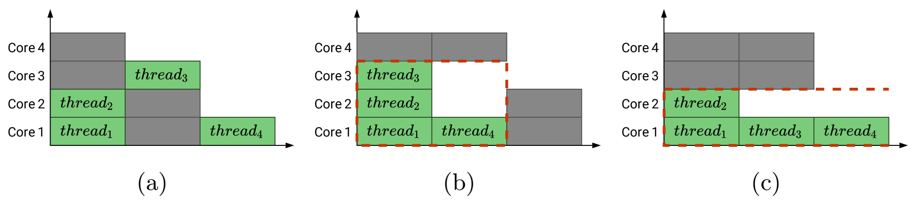


Figure 1.1: **Different thread-based execution models. Four threads have been scheduled among other threads in the system (in grey), threads 1 to 3 have to finish before thread 4 can start. (a) shows the system scheduled with a plain thread-based model, (b) shows the same system scheduled under a gang task and (c) shows it scheduled under a federated scheduling model.**

running simultaneously, it avoids long-busy wait periods while waiting for other threads. Furthermore, the number of memory transactions can be reduced by allowing the application to load the data once for all the threads instead of once per thread. This is really useful as gang tasks usually process large amounts of data. The number of memory transactions can be further reduced by using a non-preemptive[1] execution model. This property results in smaller execution times as it avoids reloading data into and from memory after each preemption.

Gang scheduling can be classified depending on when the number of cores allocated to the execution of a gang task is determined [5]. With *Rigid* gang scheduling, each task requires a fixed number of cores that is defined at design time. This means that a job of a rigid gang task cannot start until at least the number of required cores is available. *Moldable* gang scheduling is a model where each task has a minimum and a maximum number of cores on which it may execute. Thus, the scheduler decides the number of cores that is being allocated to a job when it dispatches it for execution. The number of cores allocated by the scheduler must be within the minimum and maximum number of cores defined for the task. So, the actual execution time of the job will depend on the number of cores allocated by the scheduler when dispatching that job. Note that rigid gang is a particular case of moldable gang where the minimum and maximum number of cores on which the job may execute are exactly the same. Finally, the *malleable* gang model is an extension of the moldable model but where the number of cores can also change during the execution of a job. However this model in practice is hard to implement due to the difficulty of dynamically changing the number of cores allocated to an application.

Currently, we can find examples of the rigid and malleable gang scheduling model in the scheduling of graphics processor units (GPUs) [6] where each kernel thread-block needs a fixed number of cores before execution may start. Another application can be found in the scheduling of (hardware) tasks on field programmable gate arrays (FPGAs) [7]. FPGAs are divided into multiple regions and a task can request a certain number of regions in order to execute.

[1]Preemptive execution allows the scheduler to suspend some workload and restore it later.

1.1 Limitations of the state-of-the-art

Gang tasks where introduced by Ousterhout et al. [2] in 1982 with high-performance computing in mind. However, it wasn't until 2008 that the first work in real-time scheduling theory was proposed [8]. Since then multiple tests have been proposed for *preemptive* scheduling. For rigid gang, two schedulability tests [5, 9] and one optimal scheduling policy [10] have been introduced; for moldable gang, there's a schedulability test [11, 12] and a scheduling policy [13], and for malleable gang there's a feasibility utilization bound proposed by Collette et al. [8]. Additionally, the bundled scheduling model [14] has been introduced as a way to model precedence constraints between rigid gang tasks under a preemptive execution.

Regarding the non-preemptive execution model, Dong et al. [15] introduced a utilization-based test for rigid gang tasks, while this test is fast it is pessimistic in nature. This is the closest to our work as it considers a non-preemptive gang model.

To the best of our knowledge, no analysis has been designed to obtain a safe upper bound on the worst-case response time (WCRT) of moldable gang tasks under *non-preemptive* or *limited-preemptive* scheduling, where in the latter a job of a gang task may be preempted only at well-defined preemption points. However, a recent introduced technique is the schedule-abstraction graph (SAG) proposed by Nasri and Brandenburg [16]. It provides a relatively fast and accurate response-time analysis for for global scheduling under a JLFP scheduling policy with precedence constraints [17] but it doesn't have support yet for gang tasks.

1.2 Research questions and our approach

Our main research focus in this work is in limited-preemptive moldable gang tasks. We want to answer the following questions:

1. To which extend can we improve the accuracy of schedulability analysis for gang tasks compared to the current state-of-the-art?
2. How does the moldable property of gang tasks affect schedulability of the JLFP scheduling policy?
3. Can a non-work-conserving scheduling policy improve schedulability of moldable gang tasks? If so, to what extent?

To do so, in this project we propose an extension of the notion of SAG proposed by [16] in order to derive the best-case response time (BCRT) and WCRT from a set of rigid/moldable gang tasks scheduled by a limited-preemptive JLFP scheduling policy. Moreover, we propose a new scheduling policy in order to evaluate how the schedulability of moldable gang tasks can be improved.

1.3 Organisation

The rest of this document is organised as follows: we explain the different approaches and limitations of other solutions to our problem in Chapter 2. Then, we specify our problem by detailing and defining the system model in Chapter 3. Afterwards we explain the extensions and changes that have to be made in the SAG in order to add support for limited-preemptive moldable gang tasks. The changes are divided in two steps. We start by explaining the changes for the non-preemptive execution model in Chapter 4 and we further extend it in Chapter 5 by adding support for the limited-preemptive execution model. Additionally, a new non-work conserving policy is presented in Chapter 6. Finally, an empirical evaluation of the proposed analysis and the new scheduling policy is detailed in Chapter 7. We conclude in Chapter 8.

Chapter 2

State-of-the-art

This chapter presents the current state-of-the-art regarding limited-preemptive moldable gang scheduling. The introduction of gang scheduling for HPC is explained in Section 2.1. Then regarding real-time theory the current work on rigid gang is shown in Section 2.2 and on moldable and malleable in Section 2.3. To show the differences with the limited-preemptive definition used in this project, the bundled scheduling model is presented in Section 2.4. Finally, Section 2.5 explains how SAG has already been extended.

2.1 Gang in high-performance computing

As previously mentioned, gang tasks, also called "coscheduled threads", were introduced by Ousterhout et al. [2] in 1982. Moreover, since the late 80s, the optimal scheduling of non-preemptive gang tasks has been know to be a NP-complete problem, as it is equivalent to solve the bin packing problem [18]. The concept was refined by comparing it against other thread synchronization mechanisms [4] and showing that it has an overall better system utilization [19].

Further research focused on improving the average-case response time [20]. Moreover, fragmentation can occur if there are not enough idle cores to start executing a ready job so Feitelson et al. [21] designed some packing schemes to tackle this problem. Additionally, Wiseman et al. [22] paired tasks that used different types of resources and scheduled them together to improve schedulability of compatible tasks. However, none of these methods contemplated the restrictions needed for real-time scheduling.

2.2 Real-time rigid gang

Regarding preemptive real-time rigid gang tasks, Goossens et al. [5] show that scheduling gang under a JLFP scheduling policy is not *sustainable* [23] w.r.t. execution time variation. They also propose an exact schedulability test for the task-level fixed-priority (TLFP) scheduler. Moreover, another utilization-based test has been introduced by Dong et al. [9] for gang tasks under the EDF scheduler. Finally a new optimal scheduling policy has been introduced by Goossens et al. [10] but it requires a high number of preemptions.

For the non-preemptive model, only one utilization-based test has been proposed by Dong et al. [15] designed for sporadic rigid gang tasks. This analysis is the closest to our work but, unfortunately, as it is designed for rigid gang tasks it is difficult to compare it to the scheduling of moldable gang tasks.

2.3 Real-time moldable and malleable gang

As previously said, moldable gang tasks can have a minimum and a maximum number of cores on which they may execute. Thus, the scheduling policy has to decide how many cores are actually given to the task at dispatch time. Kato et al. [11] and Richard et al. [12] proposed sufficient schedulability tests for preemptive moldable gang tasks under global EDF. Moreover, Berten et al. [13] have introduced a greedy scheduling algorithm that decides the number of cores assigned to a job based on whether the job will be able to meet the deadline with such assignment.

For malleable tasks, where a job may change its level of parallelism during its execution, Collette et al. [8] have presented a feasibility test together with an optimal scheduling policy, in terms of number of cores. They also show that the EDF scheduler is not optimal in terms of number of cores for malleable gang tasks. However, all these results are also for preemptive tasks and cannot be adapted to non-preemptive scheduling.

2.4 Bundle scheduling

As a way to improve schedulability of gang tasks and reduce some core idle time, Wasly et al. [14] have proposed the bundled task model (BTM), that extends the rigid gang task model. They also provide a sufficient schedulability test. It models tasks as a succession of "bundles" with precedence constraints between them. Then, each bundle is scheduled following a preemptive rigid gang model. Each bundle can request a different number of cores than the other bundles, thus allowing the task to have different levels of parallelism during the execution that can be changed at certain points. Nonetheless, the tests for the BTM model are designed for preemptive execution and cannot be used in the non-preemptive case.

The *limited-preemptive* definition of this project is an extension of the BTM model where each task can have precedence constraints with other tasks. The

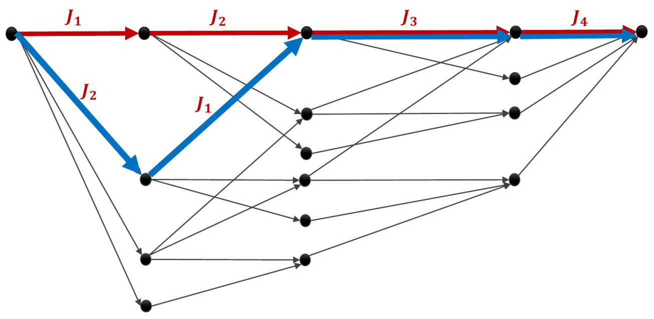


Figure 2.1: **Example of SAG, each node is a different system state and each edge connecting two nodes is a different scheduling decision leading to new multiple states. Every possible path from the first node to the last represents a possible execution scenario.**

difference comes with the fact that they are moldable gang tasks, rather than rigid, and are scheduled non-preemptively instead. This effectively is a non-preemptive model but that allows preemptions at certain specified points in the task, just between two bundles.

2.5 Schedule-abstraction analyses

The response-time analysis of this project is based on the concept of schedule-abstraction (SA). This is a new type of analysis that provides relatively fast but highly accurate schedulability results [24]. An SA-based analysis derives the WCRT and BCRT of task by building a schedule-abstraction graph that contains all possible schedules generated by the scheduling policy of a provided job set in an observation window such as the hyperperiod of the tasks. Additionally, it is able to combine *similar* schedules in order to reduce the size of the SAG. An example of SAG expansion can be seen in Figure 2.1.

It was first introduced in 2017 by Nasri and Brandenburg [16] and since then, it has been extended to various response-time analysis problems for non-preemptive tasks on single-core platforms [16]. On multi-core platforms, it has been extended for global JLFP policies [25] and for tasks with precedence constraints [17].

Chapter 3

System model

3.1 Platform and task model

The platform assumed in this project is made of m identical cores on which we execute a set of n limited-preemptive moldable gang tasks. To model the limited-preemptive execution aspect, each task τ_k ($1 \leq k \leq m$) is modelled by a directed acyclic graph (DAG), $\mathcal{G}_k = \langle V_k, E_k \rangle$ such that V_k is a set of execution segments and E_k is a set of precedence constraints between the segments in V_k.

For each arrival of a task τ_k, each execution segment of V_k releases a *job*. Each job released by an execution segment behaves like a non-preemptive moldable gang job. That is, a job J_i is defined by a minimum ($m_i^{\min}$) and a maximum ($m_i^{\max}$) number of cores on which it may be executed. The actual number of cores allocated to job J_i is determined by the scheduler at runtime according to the scheduling policy described in Section 3.2. For each possible number of cores p ($m_i^{\min} \leq p \leq m_i^{\max}$) that may be allocated to J_i, we assume that J_i will execute for a minimum of $C_i^{\min}(p)$ and a maximum of $C_i^{\max}(p)$ time units before completing its execution. Because J_i was released by a task τ_k whose structure is defined by the DAG $\mathcal{G}_k$, we defined $\mathit{pred}(J_i)$ as the set of predecessors of J_i. That is, $\mathit{pred}(J_i)$ contains all the jobs released by execution segments that are predecessors of J_i in the DAG $\mathcal{G}_k$. Since $\mathcal{G}_k$ models precedence constraints between execution segments, J_i may start to execute only if all jobs in $\mathit{pred}(J_i)$ completed their executions.

Since it is assumed that tasks release jobs according to a known arrival pattern, their schedulability analysis is equivalent to analysing the schedulability of a finite set of jobs $\mathcal{J}$ released in an observation window whose length can be computed beforehand [16]. For instance, for periodic tasks with synchronous releases, and constrained deadlines the observation window length is equal to the hyper-period of the system (i.e., the least common multiple of all tasks' periods) [10].

In sum, each job $J_i \in \mathcal{J}$ is defined by the tuple $([r_i^{\min}, r_i^{\max}], [m_i^{\min}, m_i^{\max}], \overline{C_i}^{\min}, \overline{C_i}^{\max}, d_i)$ where $r_i^{\min}$ and $r_i^{\max}$ are the earliest and latest release times of J_i respectively; $m_i^{\min}$ and $m_i^{\max}$ are the minimum and maximum number of cores on which J_i may execute; $\overline{C}_i^{\min}$ and $\overline{C}_i^{\max}$ are vectors such that each entry $C_i^{\min}(p)$ and $C_i^{\max}(p)$ contain the best-case execution time (BCET) and worst-case execution time (WCET) of J_i on p (for $m_i^{\min} \leq p \leq m_i^{\max}$), respectively; and d_i is the deadline by which J_i must complete its execution.

Without any loss of generality we assume that $m_i^{\min} \geq 1$ and $m_i^{\max} \leq m$. That is, J_i cannot execute on less than one core and cannot request more cores than the number of cores in the platform.

Note that moldable gang is more generic than rigid gang. Hence, if a job J_i has its minimum and maximum number of cores equal (i.e., $m_i^{\min} = m_i^{\max}$) then the job is said to be rigid. If all jobs released by a task are rigid, then the task is rigid.

3.2 Scheduler Model

Jobs are scheduled non-preemptively using a work-conserving job-level fixed-priority (JLFP) algorithm that is assumed to follow the following set of rules:

Rule 1: A job J_i is considered *ready* at time t if and only if it is released at or before t, it is not yet completed at t, it is not already executing at t, and all predecessors of J_i completed their execution by t.

Rule 2: A job J_i is considered *eligible* to be dispatched at time t if and only if it is ready at time t and there are at least $m_i^{\min}$ cores available at time t.

Rule 3: The scheduler is invoked whenever a job is released or a job completes

Rule 4: At every invocation of the scheduler, the highest priority *eligible* job is chosen to be dispatched.

Rule 5: The dispatched job is assigned a number of cores that is the minimum between $m_i^{\max}$ and the number of free cores at the time at which is dispatched (i.e., it always executes on as many cores as possible so as to maximize its parallelism).

Rule 6: The number of cores allocated to a job cannot change during its execution.

Rule 7: The execution of a job cannot be preempted once it started.

Rule 8: No core may remain idle as long as there are eligible jobs to be dispatched.

Since we assume a JLFP scheduling algorithm, we use the notations hp_i and lp_i to refer to the set of higher and lower priority jobs of J_i respectively.

3.3 Particular cases

The system model presented in Sections 3.1 and 3.2 covers a broad set of application models. In order to illustrate its generality, we point out a few particular cases that may be modelled and hence analysed with the results presented in Chapter 7.

1. *Limited-preemptive DAG tasks*: If all execution segments have a parallelism requirement $m_i^{\min} = m_i^{\max} = 1$, then the task model reduces to the usual DAG task model.
2. *Non-preemptive moldable gang tasks*: If all tasks are composed of a single execution segment then tasks are non-preemptive tasks.
3. *Limited-preemptive sequential tasks*: If each execution segment has at most one predecessor and at most one successor in the DAG of each tasks, then each task is composed of a sequence of non-preemptive regions as in the fixed-preemption point limited-preemptive model.
4. *Rigid gang*: If a job has its minimum and maximum number of cores equal (i.e., $\forall J_i, m_i^{\min} = m_i^{\max}$) then the job is said to be rigid gang. If all jobs released by a task are rigid, then the task is rigid.

Any combination of the above is also covered by the model assumed in this work.

Chapter 4

Non-preemptive Worst-Case Response-Time analysis

To check the schedulability of a non-preemptive task set, we compute the worst-case response time that may be experienced by any job released by each task. If the WCRT of every job released in the observation window is smaller than or equal to its deadline, then the task set is deemed schedulable.

In this chapter we start defining the analysis for the non-preemptive execution model. We do so in order to show a more step-by-step thought process that it is later extended on Chapter 5 to handle a limited-preemptive execution model. We thus simplify the assumptions made in this chapter such that for all tasks, the DAG $\mathcal{G}_k$ contains a single execution segment, i.e., $|V_k| = 1$. Note that due to space constraints, we provide the proof of Lemmas 1 to 8, Corollaries 1 to 4, and Theorems 1 and 2 in Appendix A.

4.1 Schedule Abstraction

In this work, we use the concept of *schedule abstraction* introduced by Nasri and Brandenburg[16] to compute the WCRT of every job in a job set $\mathcal{J}$.

The idea of schedule abstraction consists in encoding all possible schedules of a job set $\mathcal{J}$ with a directed acyclic graph $\mathcal{G} = \langle V, E \rangle$ where V is the set of vertices (referred to as nodes) and E is the set of edges connecting any two nodes in V. A path in the graph $\mathcal{G}$ represents a possible order of scheduling decisions taken by the scheduler, and each node $v \in V$ represents the set of all possible system states that may result from the scheduling decisions encoded on the paths that reach to V.

A direct edge connecting a node v to a node v' in $\mathcal{G}$ represents a scheduling decision taken by the scheduler that brings the set of system states represented by v to a subset of the system states represented by v'. In the context of this work, this scheduling decision consists in (1) the selection of the job that must be dispatched next on the platform, and (2) the number of cores allocated to that job. Since we schedule jobs non-preemptively, the best- and worst-case completion time of a job J_i, can be calculated at the same time as it is dispatched. This information is thus also recorded in the edge of the graph representing the dispatch of J_i. Note that because of the potential uncertainty on the actual release time and execution time of each job, the scheduling decision involving a job J_i may appear in different places in the schedule abstraction graph (i.e., after different sequences of scheduling decisions). Therefore, the worst-case response time of a job is given by the largest completion time recorded on all edges referencing that job in the schedule abstraction graph.

According to the semantic of the schedule abstraction graph $\mathcal{G}$ given above, one may conclude that, in the very particular case of a fully deterministic system (i.e., with known job arrival times, no release jitter, no execution time variation, deterministic scheduler), there is only a single possible schedule for the task set. Thus, the scheduler may take only a single sequence of scheduling decision. Therefore, the schedule abstraction graph $\mathcal{G}$ will be made of a single path recording those decisions. Hence, building the graph becomes equivalent to making a time simulation of the schedule within the observation window. However, if any system property exhibits some level of uncertainty (e.g., varying execution times or release jitter), then the number of possible schedules becomes rapidly intractable, and so would the number of paths and nodes in $\mathcal{G}$. Thus, the main challenges associated to designing a good schedule abstraction-based analysis is to find the right level of abstraction to represent system states, and to find techniques to efficiently prune branches of the graph, merge nodes together and encode a set of system states in a single node V. We cover those different aspects of the analysis in the rest of this document.

4.2 System state representation

Two of the clear differences between this work and previous works on the schedule abstraction is that (1) jobs may need more than one core to start executing, hence cores may remain idle even when there is pending workload, and (2) a single job may release more than one core simultaneously. Those two particularities imply that the time at which different cores become available to execute new workload is somewhat synchronised. Thus, we must be able to encode and keep track of such synchronization in the system state abstraction.

Therefore, we develop a new abstraction that encodes a system state using three pieces of information:

i) The set $\mathcal{S}$ of all jobs that have already been dispatched to reach the system state represented by node v.

ii) The set of all possible instants at which cores may become available to execute new workload.

iii) How many cores may be freed at the exact same time and when (remember, a job executes on p parallel cores and thus releases p cores when it completes).

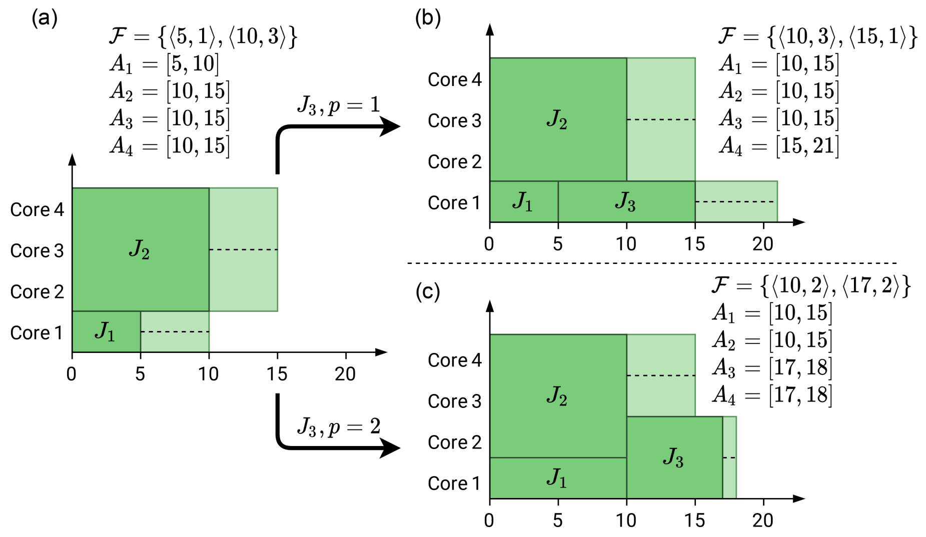


Figure 4.1: **Example of two possible execution scenarios for J_3 and their resulting system states. (a) initial state, (b) J_3 scheduled with $p = 1$, (c) J_3 scheduled with $p = 2$**

To encode (ii), we use m intervals. Each interval $A_k(v) = [A_k^{\min}(v), A_k^{\max}(v)]$ (with $1 \le k \le m$) encloses all the time instants at which k cores may become available to execute new workload after the scheduling decisions that led to node v in $\mathcal{G}$. That is, $A_k^{\min}(v)$ is the time until which there are certainly *less than* k cores available, and $A_k^{\max}(v)$ is the time by which *at least* k cores are certainly available to execute new jobs. We call $A_k^{\min}(v)$ and $A_k^{\max}(v)$ the earliest and latest availability time of k cores for system state v, and we call $A_k(v)$ the availability interval of k cores in state v. In the following, when there is no ambiguity, we do not explicitly write the system state v when referring to A_k, $A_k^{\min}$ and $A_k^{\max}$.

To encode (iii) and thus know how many cores may be freed simultaneously by a single job and at what time, we store a set of pairs of values $\mathcal{F} = \big\{\langle f_1(v), M_1(v)\rangle, \langle f_2(v), M_2(v)\rangle, \dots\big\}$ such that each pair $F_l(v) = \langle f_l(v), M_l(v)\rangle$ has the following meaning: at least $M_l(v)$ cores will be freed by a single job no earlier than time $f_l(v)$. By definition, we have that the total number of cores that may be freed is equal to m, i.e., $\sum_{l>0} M_l(v) = m$, and the earliest time $f_l(v)$ at which a group of cores may be freed must also correspond to the earliest time at which some core may become available, i.e., $\forall l, \exists k$ s.t. $f_l(v) = A_k^{\min}(v)$.

Example 4.1. *Figure (4.1a) shows a system with $m = 4$ cores where two jobs have been scheduled: J_1 on one cores must finish within the interval $[5,10]$, and J_2 on three cores must finish within $[10,15]$. In this system, one core becomes possibly available at time 5 and three additional cores become possibly available simultaneously at time 10. Similarly, one core is certainly available at time 15. Thus $\mathcal{F} = \{\langle 5,1\rangle, \langle 10,3\rangle\}$ and $A_1 = [5,10]$, $A_2 = [10,15]$, $A_3 = [10,15]$, $A_4 = [10,15]$.*

Now, assume that a job J_3 is released at time 1 with $m_3^{\min} = 1$, $m_3^{\max} = 2$,

Algorithm 1: Algorithm to generate a schedule abstraction graph.

```
input  : Job set $\mathcal{J}$
output : Bounds on the BCRT and WCRT of every job in $\mathcal{J}$
1  $\forall J_i \in \mathcal{J}, BCRT_i \leftarrow \infty, WCRT_i \leftarrow 0$;
2  initialize $\mathcal{G}$ with a root node $v_1$ with $\mathcal{S}(v_1) = \emptyset$, $A_k(v_1) = [0, 0], \forall 1 \leq k \leq m$,
     and $\mathcal{F}(v_1) = \{(0, m)\}$;
3  while $\exists$ a leaf node $v$ s.t. $\mathcal{S}(v) \neq \mathcal{J}$ do
4      $P \leftarrow$ the shortest path from $v_1$ to a leaf node $v$;
5      $v \leftarrow$ the leaf vertex of $P$;
6      for each job $J_i \in \{\mathcal{J} \setminus \mathcal{S}(v)\}$ do
7          for $\forall p \mid m_i^{\min} \leq p \leq m_i^{\max}$ do
8              if $J_i$ may be dispatched next on $p$ cores then
9                  Compute the earliest and latest finish time $EFT_i^p(v)$ and
                     $LFT_i^p(v)$ of $J_i$ on $p$ cores;
10                 $BCRT_i \leftarrow \min\{EFT_i^p(v) - r_i^{\min}, BCRT_i\}$;
11                 $WCRT_i \leftarrow \max\{LFT_i^p(v) - r_i^{\min}, WCRT_i\}$;
12                 Build the next states using Alg. 2;
13                 Try to merge the new system states with other nodes in $\mathcal{G}$
                     (Sec. 4.5);
14             end
15         end
16     end
17 end
```

$\overline{C}_3^{\min} = \{10, 7\}$ *and* $\overline{C}_3^{\max} = \{11, 8\}$*. Two execution scenarios are possible, hence two new system states are created.*

If J_1 finishes before J_2 then one core will be freed and J_3 will be scheduled with $p = 1$. This means that J_3 starts executing at the earliest at time 5 and at the latest at time 10. For $p = 1$ we know that the BCET and WCET of J_3 is 10 and 11, respectively. Therefore, the finish time interval of J_3 is $[15, 21]$. Then, as shown in Figure (4.1b), three cores become possibly available at time 10 and one additional core becomes possibly available at time 15. Therefore, we have $\mathcal{F} = \{\langle 10, 3\rangle, \langle 15, 1\rangle\}$, and the availability intervals become $A_1 = [10, 15]$, $A_2 = [10, 15]$, $A_3 = [10, 15]$, and $A_4 = [15, 20]$.

However, in another execution scenario where J_1 and J_2 finish at the same time, J_3 will be dispatched on $p = 2$ cores. This can only happen at time 10. For $p = 2$ the execution time interval of J_3 is $[7, 8]$, leading to the finish time interval $[17, 18]$. Thus, the new availability intervals are $A_1 = [10, 15]$, $A_2 = [10, 15]$, $A_3 = [17, 18]$, and $A_4 = [17, 18]$ and $\mathcal{F} = \{\langle 10, 2\rangle, \langle 17, 2\rangle\}$.

4.3 Building the schedule-abstraction graph

The SAG for a job set $\mathcal{J}$ is built according to Algorithm 1.

The algorithm starts (Line 2) by building an initial node v_1 representing the state of the system when no job has started to execute yet. Therefore, v_1 is initialized with an empty set of scheduled jobs ($\mathcal{S}_{v_1} = \emptyset$), with all cores potentially and certainly available at time 0 (i.e., $A_k(v_1) = [0, 0] \quad \forall k | 1 \leq k \leq m$) and with all m cores being freed simultaneously at time 0 (i.e., $\mathcal{F}(v_1) = \{\langle 0, m \rangle\}$).

Then, for each node in the graph that has not been analysed yet (Line 3), the algorithm checks which jobs that have not been scheduled yet may be dispatched next by the scheduler and on how many cores they may be executed (Lines 6 to 16). For each such job J_i and number of cores p, the earliest and latest completion times of the job are computed (Line 9). If the computed completion times result in larger (smaller, respectively) worst-case (best-case, respectively) response times for J_i than those computed on other paths of the graph (i.e., for other sequences of scheduling decisions), then it updates the recorded values for their $WCRT$ and/or $BCRT$ (Lines 10 and 11). Finally, Algorithm 1 uses Algorithm 2 presented in Section 4.4.3 to build *all* system states that may result from scheduling J_i on p cores in state v (Line 12) and hence expand the graph.

To defer the potential state explosion as long as possible, Algorithm 1 tries to merge the newly created nodes with existing nodes and hence reduce the number of branches in the graph (Line 13).

The algorithm stops when all nodes in the schedule-abstraction graph have been visited and all leaf nodes correspond to system states in which all jobs have been scheduled (i.e., $\mathcal{S}(v) = \mathcal{J}$).

4.4 Expansion phase

The expansion phase is divided in the following steps:

1. For each job J_i that was not dispatched yet (i.e., $J_i \notin \mathcal{S}(v)$) and for each possible number of cores $p \in [m_i^{\min}, m_i^{\max}]$ check whether J_i may be the next job dispatched by the scheduler on exactly p cores in state v.
2. If J_i may be dispatched next, compute the earliest and latest finish times of J_i.
3. Finally, build the new system states resulting from the scheduler dispatching J_i on p cores in state v. We discuss each of those steps in Sections 4.4.1 to 4.4.3.

4.4.1 Dispatch condition

To check whether J_i may be the next job dispatched by the scheduler on p cores in system state v, we first compute the earliest time $EST_i^p(v)$ at which that job would be starting to execute on p cores if it was the only job left to execute. Then, we compute the latest time $LST_i^p(v)$ at which it must have started in order to be the first job dispatched by the scheduler considering all the other pending jobs in the system. If $LST_i^p(v)$ is larger than or equal to $EST_i^p(v)$, then there exists an execution scenario in which J_i may be the next job dispatched on p by the scheduler. Otherwise, if $LST_i^p(v) < EST_i^p(v)$, then either J_i cannot be dispatched on p or there will always be another job dispatched before J_i.

4.4.1.1 Earliest Start Time

The earliest start time $EST_i^p(v)$ of J_i on p cores ($m_i^{\min} lep \leq m_i^{\max}$) can be computed the following properties:

i) By rule 1, J_i cannot start before it is released (i.e., $EST_i^p(v) \geq r_i^{\min}$).

ii) By rule 2, there must be at least p available to start to execute J_i on p (i.e., $EST_i^p(v) \geq A_p^{\min}(v)$).

iii) By rule 5, if $p < m_i^{\max}$, no more than p may be available when J_i is dispatched (Otherwise, by rule 5, it would be dispatched on more than p cores).

In order to encode property (iii) we define $A_p^{exact}(v)$ as the earliest time at which *exactly* p cores may become available. Note that $A_p^{exact}(v)$ is different from $A_p^{\min}(v)$ in the sense that $A_p^{\min}(v)$ gives the time at which at least (but not at most) p cores are available while $A_p^{exact}(v)$ denotes the time at which exactly p cores become available. We explain how to compute $A_p^{exact}(v)$ later in Section 4.4.1.2. Then properties (ii) and (iii) are encoded through $t_{gang}^p(v)$ defined in Lemma 1.

Lemma 1. *Job J_i cannot start executing with p cores before $t_{gang}^p(v)$ defined as*

$$t_{gang}^p(v) = \begin{cases} A_p^{\min}(v) & \text{if } p = m_i^{\min}, \\ A_p^{exact}(v) & \text{otherwise} \end{cases} \tag{4.1}$$

Corollary 1. *A job J_i cannot start executing on exactly p cores before time $EST_i^p(v)$, defined as*

$$EST_i^p(v) = \max\{r_i^{\min}, t_{gang}^p(v)\} \tag{4.2}$$

4.4.1.2 Computing $A_p^{exact}(v)$

The earliest time $A_p^{exact}(v)$ at which exactly p cores may become available can be computed from the information available in $\mathcal{F}(v)$. Specifically, we must find a subset $\mathcal{F}' \subseteq \mathcal{F}(v)$ such that $\sum_{F_l \in \mathcal{F}'} M_l = p$ and for which the time at which the latest core is freed (i.e., the time given by $\max_{F_l \in \mathcal{F}'}\{f_l\}$) is minimum.

The earliest time $A_p^{exact}(v)$ at which exactly p cores may become available is then equal to the time at which the last core in $\mathcal{F}'$ is freed, i.e., $A_p^{exact}(v) = \max_{F_l \in \mathcal{F}'}\{f_l\}$.

If there is no subset $\mathcal{F}' \subseteq \mathcal{F}(v)$ such that $\sum_{F_l \in \mathcal{F}'} M_l = p$, then there is no possibility for exactly p cores to become simultaneously available in system state v, i.e., there will always be more cores or less cores available at any time. Hence $A_p^{exact}(v) = +\infty$.

Note that to avoid computing all combinations of values in $\mathcal{F}(v)$, one can use text-book solutions for solving the subset-sum problem that have at most a quadratic complexity with respect to the number of cores m[26][1].

[1]Dynamic programming allows to solve the subset-sum problem with a complexity $O(sN)$ where s is the maximum sum to find and N the number of elements in the set $\mathcal{F}$. In our case, both s and the size of $\mathcal{F}$ are upper-bounded by the number of cores m

4.4.1.3 Latest Start Time

The latest start time $LST_i^p(v)$ at which job J_i may start to execute on p cores assuming that it is the next job that is dispatched by the scheduler depends on three factors:

i) The time $t^p_{avail}(v)$ at which more than p become available, since the scheduler would then dispatch J_i on more than p cores if $m_i^{\max} > p$.

ii) The time $t_{wc}(v)$ by which another job than J_i certainly becomes eligible for execution, since the scheduler will then dispatch another job before J_i if J_i did not start by then.

iii) The time $t^p_{high}(v)$ by which a higher priority job may become eligible, since the scheduler will then dispatch that other job instead of J_i.

We discuss how to compute bounds on the three time instants $t^p_{avail}(v)$, $t_{wc}(v)$, and $t^p_{high}(v)$ next.

First, according to rule 5, if J_i starts to execute on p cores at time $LST_i^p(v)$, then either p is the maximum number of cores on which J_i may execute, i.e., $p = m_i^{\max}$, or there are no more than p cores available at time $LST_i^p(v)$, since $A^{\max}_{p+1}(v)$ denotes the time by which $p+1$ cores will certainly become available, we have that

$$LST_i^p(v) \leq t^p_{avail}(v) \tag{4.3}$$

where

$$t^p_{avail}(v) = \begin{cases} A^{\max}_{p+1}(v) - 1 & \text{if } p < m_i^{\max}, \\ +\infty & \text{otherwise} \end{cases} \tag{4.4}$$

Second, if J_i is the first job dispatched by the scheduler until time $LST_i^p(v)$, then according to rules 3 and 8 there must be no other job that was eligible to be dispatched before $LST_i^p(v)$. Since by rule 2 a generic job J_j is eligible only if it is ready and there are at least $m_j^{\min}$ cores available, we must have

$$LST_i^p(v) \leq t_{wc}(v) \tag{4.5}$$

with

$$t_{wc}(v) = \min_{J_j \notin \mathcal{S}(v)} \left\{ \max \left\{ r_j^{\max}, A^{\max}_{m_j^{\min}}(v) \right\} \right\} \tag{4.6}$$

where $r_j^{\max}$ is the latest time at which a job J_j that was not scheduled yet (i.e., $J_j \notin \mathcal{S}(v)$) may be released, and $A^{\max}_{m_j^{\min}}(v)$ is the latest time by which the minimum number of cores $m_j^{\min}$ requested by J_j will be available to execute J_j.

Third, according to rule 4, if job J_i is dispatched at time $LST_i^p(v)$ and it is the first job dispatched by the scheduler in any system state v, then J_i must be the highest priority eligible job until time $LST_i^p(v)$. That is,

$$LST_i^p(v) < t^p_{high}(v) \tag{4.7}$$

where $t^p_{high}(v)$ is computed as in Lemma 2.

Lemma 2. *J_i will not be the first job dispatched in the system state v or will not be dispatched on exactly p cores if it did not start to execute before time*

$t^p_{high}(v)$ *defined as*[2]

$$t^p_{high}(v) = \min^{\infty}_{J_j \in \{\mathrm{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}} \left\{ t^p_h(J_i, J_j) \right\} \tag{4.8}$$

where

$$t^p_h(J_i, J_j) = \begin{cases} r^{\max}_j & \textit{if } m^{\min}_j \leq p \\ \max\{r^{\max}_j, A^{\max}_{m^{\min}_j}\} & \textit{otherwise} \end{cases} \tag{4.9}$$

Corollary 2. *Job J_i cannot be dispatched on p cores and be the first job dispatched in state v later than*

$$LST^p_i(v) = \min\{t^p_{avail}(v), t_{wc}(v), t^p_{high}(v) - 1\} \tag{4.10}$$

4.4.1.4 Dispatch Condition

A job J_i may be dispatched on p cores (with $m^{\min}_i \leq p \leq m^{\max}_i$) and may be the first job dispatched by the scheduler in a system state v only if the earliest time at which it may be dispatched on p cores is no later than the latest time at which it may be the first job to be dispatched. That is, it must respect the following inequality:

$$EST^p_i(v) \leq LST^p_i(v) \tag{4.11}$$

Theorem 1. *A job J_i may be dispatched on p cores and be the first job dispatched by the scheduler in system state v only if $EST^p_i(v) < \infty$ and inequality 4.11 is respected.*

4.4.2 Job finish times

The earliest time at which a job J_i may complete its execution when dispatched on p cores is when it starts at the earliest (i.e., at $EST^p_i(v)$) and executes for its best-case execution time on p cores (i.e., for $C^{\min}_i(p)$). That is,

$$EFT^p_i(v) = EST^p_i(v) + C^{\min}_i(p) \tag{4.12}$$

Similarly, the latest time at which a job J_i may complete its execution when it is the next job dispatched and it is dispatched on p cores is when it starts as late as possible (i.e., at $LST^p_i(v)$) and it runs for its WCET on p cores (i.e., for $C^{\max}_i(p)$). That is,

$$LFT^p_i(v) = LST^p_i(v) + C^{\max}_i(p) \tag{4.13}$$

[2] $\min^{\infty}_{x \in S}\{x\} = +\infty$ if $S = \emptyset$. Otherwise, $\min^{\infty}_{x \in S}\{x\} = \min_{x \in S}\{x\}$

4.4.3 Building new system states

If job J_i passes the dispatch condition for p cores in state v, then there are different execution scenarios in which the scheduler may dispatch J_i on p in system state v. For each such scenario, we build a new node v' representing the system state resulting from scheduling J_i on p cores. Apart from adding J_i to the set of scheduled jobs $\mathcal{S}_{v'}$, there are two data structures that must be updated. The set of availability intervals, and the set of earliest simultaneous cores releases $\mathcal{F}(v')$. We discuss both in the following sub-sections.

4.4.3.1 New set of earliest simultaneous core releases $\mathcal{F}(v')$

We divide this discussion in two parts. We first cover the case where the number of cores p assigned to J_i is smaller than its maximum parallelism $m_i^{\max}$, and then cover the case where $p = m_i^{\max}$.

Case $\boldsymbol{p < m_i^{\max}}$ If $p < m_i^{\max}$, then exactly p cores must be available when J_i starts to execute (rule 5). Yet, any combination of simultaneously released cores that sum to p and are possibly released between the earliest and latest start time of J_i may be used to execute J_i. Because there may be more than one such combination, we first identify every subset $\mathcal{F}_k^{=p}$ of elements in $\mathcal{F}(v)$ such that $\sum_{F_l \in \mathcal{F}_k^{=p}} M_l(v) = p$ and $\forall F_l \in \mathcal{F}_k^{=p}, f_l(v) \leq LST_i^p(v)$. Then for each subset $\mathcal{F}_k^{=p} \subseteq \mathcal{F}(v)$ that meets those conditions, we create a new node v'_k in the graph that represents the system state resulting from dispatching J_i on the specific p cores contained in $\mathcal{F}_k^{=p}$. The new set of earliest simultaneous core releases $\mathcal{F}(v'_k)$ in the new state v'_k is then built according to Lemma 3.

Lemma 3. *Let node v'_k results from executing J_i on the p cores in $\mathcal{F}_k^{=p}$, then the set of earliest simultaneous core releases is*

$$\mathcal{F}_{v'_k} = \Big\{\langle EFT_i^p(v), p\rangle\Big\} \cup \Big\{\mathcal{F}(v) \setminus \mathcal{F}_k^{=p}\Big\} \tag{4.14}$$

Case $\boldsymbol{p = m_i^{\max}}$ In the particular case where the number of cores p assigned to J_i is equal to its maximum parallelism $m_i^{\max}$, there must be at least p but also potentially more than p available when J_i starts to execute. Thus, differently from the case covered above, we identify every subset $\mathcal{F}_k^{\geq p}$ of $\mathcal{F}(v)$ whose elements sum up to *at least* p. That is, $\sum_{F_l \in \mathcal{F}_k^{\geq p}} M_l(v) \geq p$ and $\forall F_l \in \mathcal{F}_k^{\geq p}, f_l(v) \leq LST_i^p(v)$. As before, for each subset $\mathcal{F}_k^{\geq p}$, we create a new node v'_k whose set of earliest simultaneous core releases $\mathcal{F}_{v'_k}$ is computed according to Lemmas 4 and 5.

Lemma 4. *If all the cores in $\mathcal{F}_k^{\geq p}$ are released when J_i starts to execute, then J_i starts no earlier than $t_k = \max_{F_l \in \mathcal{F}_k^{\geq p}}\{f_l(v)\}$.*

Lemma 5. *Let node v'_k result from executing J_i on p of the cores in $\mathcal{F}_k^{\geq p}$, then the set of earliest simultaneous core releases is*

$$\mathcal{F}_{v'_k} = \Big\{\langle EFT_i^p(v), p\rangle\Big\} \cup \Big\{\langle t_k, (s-p)\rangle\Big\} \cup \Big\{\mathcal{F}(v) \setminus \mathcal{F}_k^{\geq p}\Big\} \tag{4.15}$$

Algorithm 2: Build all system states resulting from dispatching J_i on p cores in v.

```
1 for $\forall \mathcal{F}_k^p \subseteq \mathcal{F}(v)$ s.t. conditions of Section 4.4.3.1 are respected do
2     Add a node $v_k'$ to the schedule-abstraction graph $\mathcal{G}$;
3     $\mathcal{S}_{v_k'} \leftarrow \mathcal{S}(v) \cup \{J_i\}$;
4     Compute $PA$ and $CA$ according to Lemmas 6 and 7;
5     Sort $PA$ and $CA$ in non-decreasing order ;
6     $\forall x \,|\, 1 \leq x \leq m,\; A_k^{v_k'} = [PA_x,\, CA_x]$;
7     Compute $\mathcal{F}_{v_k'}$ according to Lemmas 3 and 5;
8     Connect $v$ to $v_k'$ with an edge;
9 end
```

where s is the number of cores in $\mathcal{F}_k^{\geq p}$, i.e., $s = \sum_{F_l \in \mathcal{F}_k^{\geq p}} M_l(v)$.

4.4.3.2 New availability intervals

To construct the availability intervals $A_x(v_k')$ $(1 \leq x \leq m)$ of a system state v_k' reachable from v, we build the set PA of all instants at which each core may *potentially* be available, and the set CA of the latest possible times at which each core will certainly become available after dispatching J_i on p cores in $\mathcal{F}_k^{=p}$ or $\mathcal{F}_k^{\geq p}$ (depending on whether $p < m_i^{\max}$ or $p = m_i^{\max}$ as discussed above). We do so using Lemmas 6 and 7.

Lemma 6. *A set of lower bounds on the time instants at which each core may potentially become available to execute new workload in v_k' is given by*

$$PA = \Big\{p \times \{EFT_i^p(v)\}\Big\} \cup \Big\{ \max\{A_x^{\min}(v), t_k\} \big| p < x \leq m\Big\} \tag{4.16}$$

Corollary 3. *A lower bound on the time at which x cores are potentially available to execute new workload in v_k' (i.e., $A_x^{\min}(v)$) is given by the x^{th} element in the non-decreasingly ordered set PA.*

Lemma 7. *A set of upper bounds on the time instants at which each core will certainly become available to execute new workload in v_k' is given by*

$$CA = \Big\{p \times \{LFT_i^p(v)\}\Big\} \cup \Big\{ \max\{A_x^{\max}(v), t_k\} \big| p < x \leq m\Big\} \tag{4.17}$$

Corollary 4. *An upper bound on the time at which x cores are certainly available to execute new workload in v_k' (i.e., $A_x^{\max}(v)$) is given by the x^{th} element in the non-decreasingly ordered set CA.*

The complete procedure presented in this section to build the system states resulting from dispatching J_i on p cores in state v is summarized in Algorithm 2.

4.5 Merge phase

To slow down the growth of the SAG and defer a potential state space explosion when building the graph of all possible scheduling decisions, Algorithm 1 tries to merge as many nodes as possible. Two system states v_k and v_q are merged in a new system state v_z according to the following rule:

Rule 9: If v_k and v_q are two nodes such that $\mathcal{S}_{v_k} = \mathcal{S}_{v_q}$ and $\forall x,\ 1 \leq x \leq m,\ A_x^{v_k} \cap A_x^{v_q} \neq \emptyset$, then v_k and v_q are merged into a single state v_z.

The availability intervals of the merged state v_z are then computed so that they enclose the availability intervals of both states v_k and v_q. That is, $\forall x,\ 1 \leq x \leq m$:

$$A_x^{v_z} = \Big[\min\{A_x^{\min}(v_k), A_x^{\min}(v_q)\}, \max\{A_x^{\max}(v_k),\ A_x^{\max}(v_q)\}\Big] \tag{4.18}$$

This way, all possible combinations of instants at which cores become available in either state v_k or v_q is also possible in v_z.

The set of earliest simultaneous core releases $\mathcal{F}(v_z)$ of the merged state is computed using Algorithm 3 which performs the following procedure. For both initial states v_k and v_q, it sorts the groups of cores that are simultaneously released in a non-decreasing order with respect to the time at which they are released. It then breaks the groups of simultaneously released cores in smaller ones so that the size of the groups match in both states (Lines 3 to 10), i.e., after the transformation we have $|\mathcal{F}'(v_k)| = |\mathcal{F}'(v_q)|$ and $\forall x \,|\, 1 \leq x \leq |\mathcal{F}'(v_k)|,\ |M'(v_k)| = |M'(v_q)|$. It then keeps the groups of cores that are released the earliest and assigns them to $\mathcal{F}(v_z)$ (Lines 12 to 16), i.e., $|\mathcal{F}(v_z)| = |\mathcal{F}'(v_k)| = |\mathcal{F}'(v_q)|$ and $\forall x \,|\, 1 \leq x \leq |\mathcal{F}'(v_k)|,\ M_x(v_z) = M'_x(v_k) = M'_x(v_q)$ and $f_x(v_z) = \min\{f'_x(v_k), f'_x(v_q)\}$.

We now prove that all simultaneous core release patterns that are possible in one of the two initial states v_k or v_q is also possible in the new merged state v_z.

Lemma 8. *If exactly p cores may be available at time t in either v_k or v_q, then exactly p cores may be available at time t in v_z.*

4.6 Proof of correctness

After the algorithm to build the SAG has been presented, we prove that the analysis covers all possible execution scenarios and hence returns safe bounds on the BCRT and WCRT of each job in the analysed job set $\mathcal{J}$.

Theorem 2. *For any possible execution scenario such that J_i executes on p cores and finishes at t, there is a path $\langle v_1, \ldots, v_k \rangle$ in the schedule-abstraction graph such that J_i passes the dispatch condition on p cores in v_k and $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$.*

Algorithm 3: Merge of $\mathcal{F}(v_k)$ and $\mathcal{F}(v_q)$ into $\mathcal{F}(v_z)$

```
input  : $\mathcal{F}(v_k)$ and $\mathcal{F}(v_q)$
output : $\mathcal{F}(v_z)$
1  $\mathcal{F}'(v_k) = \mathcal{F}'(v_q) = \emptyset$;
2  while $\mathcal{F}(v_k) \neq \emptyset \wedge \mathcal{F}(v_q) \neq \emptyset$ do
3      Extract the pair $\langle f_K, M_K \rangle$ such that $f_K$ is the minimum value in $\mathcal{F}(v_k)$
         and, in case of tie, $M_K$ is the minimum among the tying values. For
         $\mathcal{F}(v_q)$ extract $\langle f_Q, M_Q \rangle$ using the same rule;
4      $M_{new} \leftarrow \min\{M_K, M_Q\}$;
5      Add $\langle f_K, M_{new} \rangle$ to $\mathcal{F}'(v_k)$;
6      Add $\langle f_Q, M_{new} \rangle$ to $\mathcal{F}'(v_q)$;
7      $M_K \leftarrow M_K - M_{new}$;
8      $M_Q \leftarrow M_Q - M_{new}$;
9      Add $\langle f_K, M_K \rangle$ to $\mathcal{F}(v_k)$ if $M_K > 0$ ;
10     Add $\langle f_Q, M_Q \rangle$ to $\mathcal{F}(v_q)$ if $M_Q > 0$ ;
11 end
12 forall $1 \leq x \leq |\mathcal{F}'(v_k)|$ do
13     $f_x(v_z) = \min\{f'_x(v_k), f'_x(v_q)\}$;
14     $M_x(v_z) = M'_x(v_k)$;
15     Add $\langle f_x(v_z), M_x(v_z) \rangle$ to $\mathcal{F}(v_z)$;
16 end
17 If either $\mathcal{F}(v_k)$ or $\mathcal{F}(v_q)$ is not empty, add the remaining values to $\mathcal{F}(v_z)$.
18 return $\mathcal{F}(v_z)$;
```

Chapter 5

Limited-preemptive analysis extension

The limited-preemptive execution model analysis is an extension of the non-preemptive analysis detailed in Chapter 4. Some changes and additions are required to handle the precedence constraints between jobs. In Section 5.1 we explain the additional information required in the system state to track precedence constraints. Then, in Section 5.2, we show the changes needed in the dispatch condition and how to build the new system state. Finally, there are also some additions in the merge phase, explained in Section 5.3. Note that due to space constraints, we provide the proof of Lemmas 9 to 16, Lemma 2-bis, Corollaries 5 and 6, and Corollary 1-bis in Appendix B.

5.1 System state representation

The system state representation needed for the limited-preemptive model is an extension of the one introduced in Section 4.2. By rule 1, a job cannot be considered ready if there's a predecessor that has not completed its execution yet. While at first, it would seem possible to know this information by checking whether all the predecessors of a job J_i are in the set of dispatched jobs $\mathcal{S}(v)$ (i.e., $pred(J_i) \subseteq \mathcal{S}(v)$) this does not account for the event where all the predecessors of a job have been dispatched but some of them have not finished their execution yet.

Therefore, we need to keep track of running jobs to prevent their successors from being dispatched while they are still running. We do so by encoding the set $\mathcal{X}(v)$ of jobs for which we are certain that they are still running. Note that it is a subset of the jobs that have already been dispatched until reaching state v, that is $\mathcal{X}(v) \subseteq \mathcal{S}(v)$. For each job $J_x \in \mathcal{X}(v)$, three bounds are saved in the system state v:

- A lower-bound $EFT_x(v)$ on the *earliest finish time* of J_x
- An upper-bound $LFT_x(v)$ on the *latest finish time* of J_x

- A lower-bound $p_x(v)$ on the number of cores on which J_x is running. Remember, that each job is moldable and its execution may be parallelised on a variable number of cores depending on the scheduler's decisions encoded in different paths leading to v.

5.2 Expansion phase

The changes required for the expansion phase are focused in adding some new equations to the dispatch condition of a job J_i.

5.2.1 Ready jobs

Previously the ready time of a job J_i was defined by its release time interval $[r_i^{\min}, r_i^{\max}]$. However, as a job cannot be ready until all of its predecessors have completed their execution, this is no longer the case.

So, similar to [17], we define the set $\mathcal{R}(v)$ of jobs that are *potentially ready* to be scheduled next in the system state v as the set of jobs that have not been dispatched yet and have had all their predecessors already dispatched, i.e.,

$$\mathcal{R}(v) = \Big\{ J_i | J_i \in \{\mathcal{J} \setminus \mathcal{S}(v)\} \wedge \mathit{pred}(J_i) \subseteq \mathcal{S}(v) \Big\} \tag{5.1}$$

Furthermore, according to rule 1 a job is ready to execute if and only if:

i) It is released.

ii) It was not yet dispatched.

iii) All its predecessors have completed.

Therefore, we define the earliest time $R_i^{\min}(v)$ at which a job $J_i \in \mathcal{R}(v)$ may be potentially ready as the maximum of $r_i^{\min}$ and the earliest time at which all predecessors of J_i have possibly completed, i.e.

$$R_i^{\min}(v) = \max \Big\{ r_i^{\min}, \max_0 \{ EFT_x^*(v) | J_x \in \mathit{pred}(J_i) \} \Big\} \tag{5.2}$$

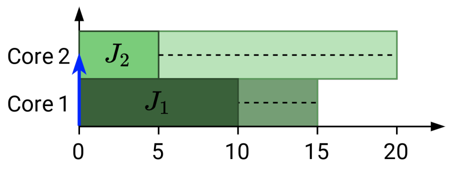
where $EFT_x^*(v)$ is a safe lower bound (defined next) on the earliest finish time of J_x for all execution scenarios that lead to the system states encoded in node v.

Similarly, the latest time $R_i^{\max}(v)$ at which $J_i \in \mathcal{R}(v)$ is certainly ready is defined as the maximum of $r_i^{\max}$ and the time at which all predecessors of J_i have certainly completed, i.e.,

$$R_i^{\max}(v) = \max \Big\{ r_i^{\max}, \max_0 \{ LFT_x^*(v) | J_x \in \mathit{pred}(J_i) \} \Big\} \tag{5.3}$$

where $LFT^*(v)$ is a safe upper bound on the latest finish time of J_x in state v.

To compute safe bounds on the earliest and latest finish time of a job $J_x \in \mathit{pred}(J_i)$ until reaching a given system state v, we consider two different cases depending on whether J_x is in the set of certainly running jobs $\mathcal{X}(v)$ at v or not:



(a) **System state**

J_i	Priority	$pred(J_i)$ Case A	$pred(J_i)$ Case B
J_3	High	$\{J_1\}$	$\{J_2\}$
J_4	Low	$\emptyset$	$\emptyset$

(b) **Jobs' specification**

Figure 5.1: **Representation of system state and jobs of Example 5.1. Two jobs can be scheduled now, where J_3 has a higher priority and a predecessor. We analyze two cases.**

1. For the predecessors of J_i that are *certainly running* in system state v, i.e., any job $J_x \in \{pred(i) \cap \mathcal{X}(v)\}$, the bounds $EFT_x^*(v)$ and $LFT_x^*(v)$ can safely assume the values $EFT_x(v)$ and $LFT_x(v)$ saved for that job in state v as explained in Section 5.1.
2. For predecessors of J_i that are not certainly running in state v, i.e., any job $J_x \in pred(J_i)$ that is not in $\mathcal{X}(v)$, there is no bound on $EFT_x(v)$ and $LFT_x(v)$ saved in v. Note that the decision of not saving those values in each node in the graph is an intentional optimization to limit the amount of memory required by the algorithm. Therefore, we instead use the current values of $BCRT_x$ and $WCRT_x$ computed so far by Algorithm 1 as they are safe bounds on the EFT and LFT of J_x for all system states explored until reaching this point of the schedule-abstraction graph, which then also includes v.

5.2.2 Dispatch condition

When updating the analysis to support the limited-preemptive execution model, some changes have to be done in the way the $EST_i^p(v)$ and $LST_i^p(v)$ of a job J_i are computed.

5.2.2.1 Minimum start time with higher-priority jobs

There is one extra condition that needs to be accounted for when we compute the $EST_i^p(v)$ of a job J_i under the limited-preemptive execution model. If a lower-priority job J_i is ready to be dispatched at time t but every allocation of cores for J_i will use cores just freed by the certainly running predecessors of a higher-priority job J_k then this higher-priority job can also be eligible to be dispatched instead of J_i if there are at least $m_k^{\min}$ cores available.

In the non-preemptive analysis, it may seem that $t_{high}^p(v)$, in Equation (4.8), already encodes this condition but that is not the case as it does not account for possible allocations of cores that ensure that the predecessors of a higher-priority job have finished their execution. An example of the limitations of $t_{high}^p(v)$ can be seen in Example 5.1.

Example 5.1. *Let's suppose that we have a system such that $m = 2$ with two jobs that have been scheduled already, (shown in Figure 5.1a) where J_1 and J_2*

have the finish time intervals $[10, 15]$ *and* $[5, 20]$ *respectively. There are two jobs ready:* J_3 *and* J_4*,* J_3 *has the highest priority and* J_4 *the lowest. We further assume that* J_3 *has a single predecessor whilst* J_4 *does not have any. We analyze two different cases where we evaluate the possibility of dispatching* J_4 *depending on which job is the predecessor of* J_3*:*

Case A *The predecessor of* J_3 *is* J_1*. In this case* J_4 *can start at the earliest at time 5 right after* J_2 *finishes its execution. As* J_3 *has to wait for its predecessor (*J_1*) to finish its execution,* J_4 *can start at the latest right before* J_3 *certainly does, which happens at time 15. So we have* $EST_4^p(v) = 5$ *and* $LST_4^p(v) = 14$.

Case B *The predecessor of* J_3 *is* J_2*. Now in this case, it would seem possible that* J_4 *can start at the earliest at time 5. However, that is not the case because, if* J_4 *were to start at time 5, this would imply that* J_2 *has finished its execution and thus* J_3 *can be dispatched. We can see how the earliest time at which* J_4 *can be dispatched is at time 10 after* J_1 *finishes its execution; and the latest time is at time 15 where due to rule 8 the scheduler is certain to dispatch* J_4*, thus obtaining* $EST_4^p(v) = 10$ *and* $LST_4^p(v) = 15$.

So, we need to compute the earliest time $t_i^{pred}(v)$ such that we are certain that if J_i can be dispatched, there's no other higher-priority job that has also become eligible at the same time. Note that even if we perform the analysis without the lower bound provided by $t_i^{pred}(v)$ the analysis is still valid but more pessimistic as we end up exploring impossible execution scenarios.

To compute this lower bound we have to check whether dispatching J_i at time $t_i^{pred}(v)$ would certainly use cores freed by predecessors of a released higher-priority job that was just waiting for them. If that is the case, the current $t_i^{pred}(v)$ is not valid and we have to find another one. This means that computing $t_i^{pred}(v)$ is an iterative process where we have to check the times at which J_i could possibly start. These are all the possible times at which the scheduler could be invoked as defined by rule 3. We define our solution step by step.

Lemma 9. *The set of certainly running predecessors* $\mathcal{X}_x^{pred}(v)$ *of a job* J_x *in state* v *is defined as*

$$\mathcal{X}_x^{pred}(v) = \{pred(x) \cap \mathcal{X}(v)\} \tag{5.4}$$

Lemma 10. *The number of cores possibly available at a time* t *in state* v *can be computed as*

$$q^{\min}(v, t) = \max_{1 \le k \le m} \{k | t \ge A_k^{\min}(v)\} \tag{5.5}$$

Lemma 11. *At time* t *in state* v*, the set of waiting jobs* $J_k \in \mathcal{Q}_i(v, t)$ *that have to be checked because* J_i *being dispatched could mean that the predecessors of* J_k *have finished their execution is defined as*

$$\mathcal{Q}_i(v, t) = \left\{ J_k \middle| \underbrace{\text{can\_start}\,(J_k, v, t)}_{J_k \textit{ can start at time } t} \wedge \underbrace{\mathcal{X}_k^{pred}(v) \neq \emptyset}_{\substack{\textit{Has a certainly} \\ \textit{running predecessor}}} \wedge J_k \in \text{hp}_i \right\} \tag{5.6}$$

where

$$\text{can\_start}\,(J_x, v, t) = \Big(t \geq \max\Big\{r_x^{\max}, A^{\min}_{m_x^{\min}}(v)\Big\}\Big) \wedge \Big(pred(J_x) \subseteq \mathcal{S}(v)\Big) \quad (5.7)$$

Jobs $J_k \in \mathcal{Q}_i(v,t)$ are higher-priority jobs with a certainly running predecessor that can start if J_i does but, since they have precedence constraints, these constraints may prevent J_k from starting. Thus, we have to check whether J_i being dispatched at time t means that these constraints are *certainly* fulfilled in which case, job J_k is dispatched instead of J_i. To do so, we check whether the scheduler will allocate the cores that were previously being used by the predecessors of a higher-priority job $J_k \in \mathcal{Q}_i(v,t)$ to job J_i. If (i) that's the case, (ii) cores from *all* the certainly running predecessors of job $J_k \in \mathcal{Q}_i(v,t)$ are needed and (iii) no other allocation is possible, J_k will certainly be able to start at time t if J_i does.

Lemma 12. *If job J_i can start at time t in state v a higher-priority job $J_k \in \mathcal{Q}_i(v,t)$ will also certainly be able to start if all possible allocations of cores for J_i require for each of the certainly running predecessors $J_j \in \mathcal{X}_k^{pred}(v)$ at least one core of the cores previously being used by J_j.*

Now, in order to check whether J_i uses at least one core from each of the cores freed by the certainly running predecessors of a job J_k we know that J_i can use the possibly available cores $q^{\min}(v,t)$ minus the number of cores that would *ensure* that J_i is using cores from all the predecessors.

Lemma 13. *The minimum number of cores $p_k^{pred}(v)$ that ensures that all possible allocations of cores for J_i certainly use at least one core from each of the cores freed by certainly running predecessors of a job $J_k \in \mathcal{Q}_i(v,t)$ is*

$$p_k^{pred}(v) = \min_{J_j \in \mathcal{X}_k^{pred}(v)} p_j \quad (5.8)$$

Corollary 5. *The job $J_k^{pred}(v) \in \mathcal{X}_k^{pred}(v)$ from the certainly running predecessors of a higher-priority job $J_k \in \mathcal{Q}_i(v,t)$ that has the minimum number of simultaneously freed cores where J_i cannot be scheduled is*

$$J_k^{pred}(v) = \operatorname*{arg\,min}_{J_j \in \mathcal{X}_k^{pred}(v)} p_j \quad (5.9)$$

Lemma 14. *The set of jobs $\mathcal{Q}_i^{pred}(v,t)$ whose number of cores cannot be used by J_i is defined as*

$$\mathcal{Q}_i^{pred}(v,t) = \Big\{J_k^{pred}(v) \Big| J_k \in \mathcal{Q}_i(v,t)\Big\} \quad (5.10)$$

Corollary 6. *The total number of cores that cannot be used by J_i if it can be scheduled at the earliest at time t is*

$$q_i^{pred}(v,t) = \sum_{J_j \in \mathcal{Q}_i^{pred}(v,t)} p_j \quad (5.11)$$

Algorithm 4: Computation of lower bound for $t_i^{pred}(v)$

```
input  : State $v$ and job $J_i$
output : $t_i^{pred}(v)$
1 $t_i^{pred}(v) := R_i^{\min}(v)$;
2 foreach $A_k^{\min}(v) \in \{A_p^{\min}(v), \ldots, A_m^{\min}(v)\}$ do
3     $t_i^{pred}(v) := \max\{t_i^{pred}(v), A_k^{\min}(v)\}$;
4     if inequality 5.12 holds with $t = t_i^{pred}(v)$ then
5         return $t_i^{pred}(v)$;
6     end
7 end
8 return $+\infty$;
```

Lemma 15. *J_i may be scheduled at time t at the earliest on p cores if it matches the following condition:*

$$q^{\min}(v,t) - q_i^{pred}(v,t) \geq p \tag{5.12}$$

Finally in order to compute the lower bound for $t_i^{pred}(v)$ we use Algorithm 4 that checks all the times at which J_i can be possibly dispatched. If none of the evaluated times matches the condition then the algorithm returns $+\infty$ as this means that J_i will not be dispatched in system state v.

5.2.2.2 Earliest Start Time

So now, the earliest time at which job J_i may start to execute on p cores depends on the following factors:

i) The time $R_i^{\min}(v)$ at which the job can possibly start and all its predecessors may have finished. This effectively replaces $r_i^{\min}$ in the previous Equation (4.2).

ii) The time $t_{gang}^p(v)$ at which we know that executing J_i on p cores is possible.

iii) The time $t_i^{pred}(v)$ by which we are certain that J_i can start but other released higher-priority jobs are still waiting for their predecessors to finish their execution.

Corollary 1-bis. *A job J_i in a limited-preemptive execution model cannot start on exactly p cores before time $EST_i^p(v)$, defined as*

$$EST_i^p(v) = \max\{R_i^{\min}(v), t_{gang}^p(v), t_i^{pred}(v)\} \tag{5.13}$$

5.2.2.3 Latest Start Time

The $LST_i^p(v)$ of a limited-preemptive job J_i is computed in a similar way as previously explained in Section 4.4.1.3. However, we must update the definitions of $t_{wc}(v)$ and $t_{high}^p(v)$ in order to handle jobs with precedence constraints ($t_{avail}^p(v)$ does not get affected and can be used as before).

As previously explained, if J_i is the first job dispatched by the scheduler until $LST_i^p(v)$, then according to rules 3 and 8 there must be no other job that was eligible to be dispatched before $LST_i^p(v)$. Since by rule 2 a generic job J_j is eligible only if it is ready and, by rule rule 1, it is only ready if all its predecessors have completed execution (i.e., after $R_i^{\max}(v)$) we must have

$$LST_i^p(v) \leq t_{wc}(v) \tag{5.14}$$

where

$$t_{wc}(v) = \min_{J_j \in \mathcal{R}(v)} \left\{ \max \left\{ R_j^{\max}(v), A_{m_j^{\min}}^{\max}(v) \right\} \right\} \tag{5.15}$$

where $R_j^{\max}(v)$ is the latest time at which $J_j \in \mathcal{R}(v)$ may be ready, and $A_{m_j^{\max}}^{\max}$ is the latest time by which $m_j^{\max}$ cores may be available to execute new workload in all system states covered by v.

Similarly, we update $t_{high}^p(v)$ to account for cases where the higher priority jobs are still waiting for their predecessors to finish execution. Thus we redefine Lemma 2 in Lemma 2-bis.

Lemma 2-bis. *J_i will not be the first job dispatched in the system state v or will not be dispatched on exactly p cores if it did not start to execute before time $t_{high}^p(v)$ defined as*

$$t_{high}^p(v) = \overset{\infty}{\min_{J_j \in \{\mathrm{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}}} \left\{ \max \left\{ t_h^p(J_i, J_j), t_{h,pred}(J_i, J_j) \right\} \right\} \tag{5.16}$$

where

$$t_h^p(J_i, J_j) = \begin{cases} r_j^{\max} & \text{if } m_j^{\min} \leq p \\ \max\{r_j^{\max}, A_{m_j^{\min}}^{\max}\} & \text{otherwise} \end{cases} \tag{5.17}$$

and

$$t_{h,pred}(J_i, J_j) = \overset{0}{\max}\{LFT_y^*(v) | J_y \in pred(J_j) \setminus pred(J_i)\} \tag{5.18}$$

Finally after these changes, $LST_i^p(v)$ can be computed as previously stated in Equation (4.10) of Corollary 2:

$$LST_i^p(v) = \min\{t_{avail}^p(v), t_{wc}(v), t_{high}^p(v) - 1\} \tag{5.19}$$

5.2.2.4 Dispatch condition

Now that we have updated the definitions of $EST_i^p(v)$ and $LST_i^p(v)$ we can use the same dispatch condition as stated in Section 4.4.1.4, i.e., making sure that J_i respects inequality 4.11.

5.2.3 Building a new system state

As explained in Section 4.4.3, after dispatching J_i we generate multiple new states v'_k. Now, as we are analysing the limited-preemptive execution model, we also have to update the set of certainly running jobs $\mathcal{X}(v'_k)$ to represent the jobs that are certainly running in the new state v'_k. We update this set using a similar method as the one used by Nasri et. al. in [17].

Lemma 16. *The set of certainly running jobs $\mathcal{X}(v'_k)$ of the new state v'_k is comprised of*

$$\mathcal{X}(v'_k) = \Big\{J_i\Big\} \cup \Big\{J_j | J_j \in \mathcal{X}(v) \wedge EFT_j(v) > LST_i^p(v)\Big\} \tag{5.20}$$

5.3 Merge phase

Finally the last part of the analysis that needs to be updated in order to account for a limited-preemptive execution model is the merge phase. As we explained in Section 4.5, two system states v_k and v_q can be merged in a new system state v_z. We can use the same method as previously used but we also must merge the two sets of certainly running jobs $\mathcal{X}(v_k)$ and $\mathcal{X}(v_q)$ into $\mathcal{X}(v_z)$.

To do so, we extend the method described in section 4.6 in [17] where the set of certainly running jobs was updated as

$$\mathcal{X}(v_z) = \Big\{J_x \Big| J_x \in \mathcal{X}(v_k) \cap \mathcal{X}(v_q)\Big\} \tag{5.21}$$

Additionally, we update the number of cores p_x on which each job J_x is certainly running in the resulting state v_z. We consider the bounds that were previously defined for all execution scenarios leading to either v_k or v_q, and hence also to the merged state v_z. Therefore we have that

$$p_x(v_z) = \min\big\{p_x(v_k), p_x(v_q)\big\} \tag{5.22}$$

Chapter 6

A non-work conserving scheduling policy

As explained in Section 1.2 we want to evaluate whether a new scheduling policy can improve the schedulability ratio of moldable gang tasks. We do so, by proposing a new non-work scheduling policy that can solve the limitations of the JLFP scheduler. Moreover, we also propose an extension to our scheduling policy that can do a better assignment of cores when multiple jobs are dispatched at the same time.

6.1 Limitations of the job-level fixed-priority scheduling policy

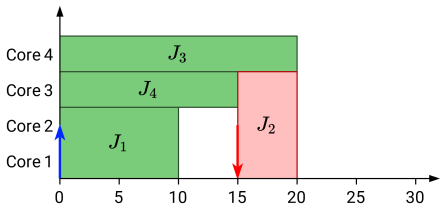
Even if the JLFP scheduling policy performs reasonably well, there are still some limitations that appear when dealing with moldable gang tasks. We explore a limitation caused by the work-conserving property (rule 8) and another limitation caused by how the number of cores are assigned to a job, defined in rule 5.

6.1.1 Work-conserving policy

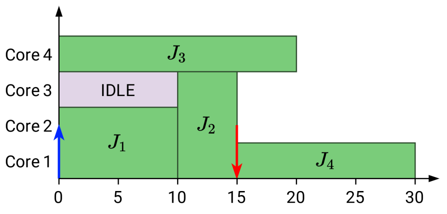
By rule 8 the JLFP scheduler uses a work-conserving policy. However, this may cause some priority inversion issues when scheduling gang tasks. Example 6.1 shows a case with rigid gang jobs where this happens and it is caused by a job requiring more cores than the ones that are available at that moment, causing a deadline miss later on.

Example 6.1. *Figure 6.1c defines some jobs that have to be scheduled in a system with $m = 4$ cores. J_1 is the one with the highest priority and J_4 is the one with the lowest, additionally J_2 has a deadline at time 15.*

The result of scheduling this system with a JLFP scheduling policy can be seen in Figure 6.1a. With such policy, J_1 is dispatched first at time 0 but J_2



(a) **Job set scheduled under JLFP**



(b) **Job set with no deadline misses**

J_i	Priority	Cores	Execution time	Deadline
J_1	High	2	10	$+\infty$
J_2	Mid-high	3	5	15
J_3	Mid-low	1	20	$+\infty$
J_4	Low	1	15	$+\infty$

(c) **Jobs' specification**

Figure 6.1: **Example of priority inversion that can happen with gang jobs. (a) shows the result of scheduling the jobs with the JLFP scheduling policy and (b) shows a possible solution without deadline misses.**

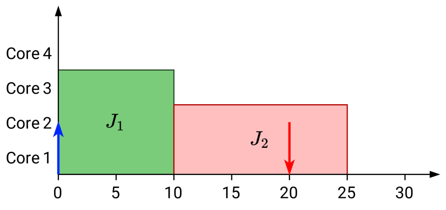
cannot be dispatched at time 0 as at this moment there are only 2 cores free and J_2 requires 3 cores to execute. This allows both J_3 and J_4 to be dispatched and thus retarding the execution of J_2 until time 15 where it causes a deadline miss. However, this job set could have been scheduled without deadline misses as shown in Figure 6.1b by leaving one core idle that would have allowed J_2 to start running at time 10.

6.1.2 Cores assigned to moldable gang tasks

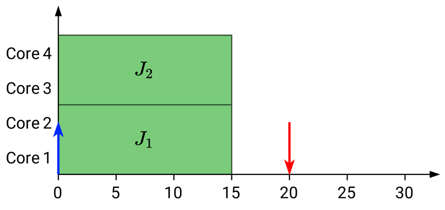
Additionally, as defined in rule 5, the number of cores allocated to a job J_i is the minimum between $m_i^{\max}$ and the number of free cores at the time at which is dispatched. This means that if there are multiple jobs dispatched at the same time, there can be an unbalance between the number of cores assigned.

Example 6.2. *Figure 6.2c defines some jobs that have to be scheduled in a system with $m = 4$ cores. The result of scheduling these jobs under the JLFP scheduler can be seen in Figure 6.2a. The deadline miss is caused by the fact that J_1 is assigned 3 cores and thus J_2 is not eligible to be scheduled as there are not enough cores to dispatch it at time 0. However, we know that if we had a better scheduling policy, the job set is actually schedulable as it can be seen in Figure 6.2b.*

So as it can be seen in Example 6.2 it would be possible to reduce the number of deadline misses by having a scheduling policy that performs a better assignment of the number of cores allocated to a job.



(a) **Job set scheduled under JLFP**



(b) **Job set with no deadline misses**

J_i	Priority	Cores min	Cores max	Execution time	Deadline
J_1	High	2	3	10	$+\infty$
J_2	Low	2	2	5	15

(c) **Jobs' specification**

Figure 6.2: **Example of deadline miss caused by the scheduler allocating too many cores to the highest-priority job. (a) shows the result of scheduling the jobs under the JLFP scheduling policy and (b) shows a possibly solution without deadline misses.**

6.2 Reservation-based non-work conserving scheduling policy

In order to prevent priority inversions and deadline misses explained in Section 6.1 we introduce a new reservation-based non-work conserving scheduling policy that attempts to solve these issues. This new policy is based on the JLFP scheduler so it still uses priorities assigned to jobs, we call it the reservation-based gang (ResG) scheduling policy.

6.2.1 Basic idea

One of the limitations shown in previous sections is that the work-conserving property allows lower-priority jobs to be dispatched before higher-priority jobs if there are not enough cores for the higher-priority jobs to become eligible. This in turn can further delay the dispatch time of the higher-priority job.

In order to solve this, we allow a higher-priority job J_i, to reserve the minimum number of cores that they need in order to be eligible (i.e., m_i^{min}). Once these cores are reserved, we make sure that dispatching a new job in the system does not increase the availability times of the number of cores reserved. By doing this, a lower-priority job is allowed to start before a higher-priority job with reservations if the lower-priority job is guaranteed to finish before the availability time of the cores required by the higher-priority job. Note that, to compute the availability times, we only use the WCET of the job (i.e., $C_i^{\text{max}}(p_i)$).

Example 6.3. *If we take the job set previously defined in Figure 6.1c and we apply this new rule we can see the result in Figure 6.3. This result was achieved with the following steps:*

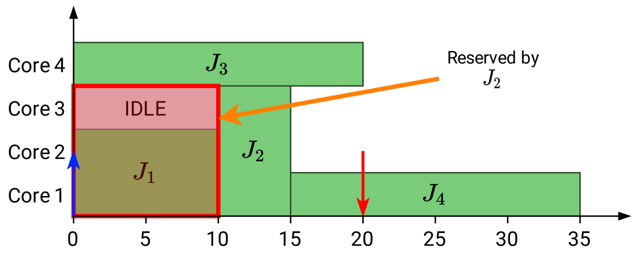


Figure 6.3: **System of Figure 6.1c scheduled under the ResG scheduler**

1. *At time 0, J_1 has the highest priority, so it is scheduled first with $p = 2$ cores.*
2. *Then, J_2 has $m_2^{\min} = 3$ but only 2 cores are free at this moment, so it reserves the 3 cores leaving only 1 virtually free core where other jobs can be scheduled. The availability times are $A = \{0, 0, 10, 10\}$ since 1 core is available at time 0, 2 cores are available at time 0, 3 cores are available at time 10 and 4 cores are available at time 10.*
3. *As J_3 has $m_3^{\min} = 1$ and can start without increasing the availability of 3 cores (i.e., $A_3 = 10$) so it is also dispatched at time 0. The availability times are $A = \{0, 10, 10, 20\}$.*
4. *J_4 has $m_4^{\min} = 1$ but dispatching it at time 0 would increase the availability time of 3 cores from time 10 to time 20 so J_4 is not dispatched and instead reserves the remaining core that could be reserved at time 0.*
5. *At time 10, after J_1 finishes its execution, the scheduler is called again. There are now 3 cores free and none of them are reserved at time 10, so J_2 is dispatched.*
6. *Finally, at time 15, J_4 gets dispatched.*

What happened here is that J_1, as it had the highest priority, was scheduled with $p = 2$ cores leaving two cores idle. Then J_2 required 3 cores to be eligible but as only 2 cores were free it reserved *3 cores (2 cores from already running jobs and 1 core leaving it idle).*

Deciding the number of cores that will be assigned to a job is divided in two steps. The first step happens when deciding if a job will be dispatched or not and we always use the minimum number of cores p such that we are sure that the deadline is always met and p falls within the bounds of the job (i.e., $m_i^{\min} \leq p \leq m_i^{\max}$), which can be expressed as

$$\min_{m_i^{\min} \leq p \leq m_i^{\max}} t + C_i^{\max}(p) \leq d_i \tag{6.1}$$

In the second step, once we know all the jobs that will be dispatched at a specific time t, we decide to assign more cores to jobs that can still use more (i.e., $p_i < m_i^{\max}$). How we decide which jobs get more cores and how many cores is detailed in Section 6.2.3.

6.2.2 Scheduler model

We propose the following set of rules to model the ResG scheduling policy:

Rule 1: A job J_i is considered *ready* at time t if and only if it is released at or before t, it is not yet completed at t, it is not already executing at t, and all predecessors of J_i completed their execution by t.

Rule 2: A job J_i is considered *eligible* to be dispatched at time t if and only if it is ready at time t and it belongs to the eligible set of jobs $\mathcal{J}^{eligible}$.

Rule 3: The scheduler is invoked whenever a job is released or a job completes.

Rule 4: At every invocation of the scheduler, the sets $\mathcal{J}^{eligible}$ and $\mathcal{M}^{eligible}$ are computed with Algorithm 5 and $\mathcal{M}^{moldable}$ is computed with Algorithm 6.

Rule 5: The dispatched job J_i is assigned $p_i \in \mathcal{M}^{moldable}$ cores. That is, the number of cores defined in the set $\mathcal{M}^{moldable}$ for job J_i.

Rule 6: The number of cores allocated to a job cannot change during its execution.

Rule 7: The execution of a job cannot be preempted once it started.

Algorithm 5 computes both the set $\mathcal{J}^{eligible}$ of jobs that are eligible and will be dispatched, and also the set $\mathcal{M}^{eligible}$ of cores assigned to each job in $\mathcal{J}^{eligible}$. Each job J_i is checked in order of priority, then at Line 7 the minimum number of cores needed to meet the deadline is selected (this can be computed with Equation (6.1) as previously explained). Then Line 8 checks if enough free cores are available to dispatch J_i. There are multiple options that can happen:

- **There are enough cores** (i.e., $p_i \leq (m - m^{busy})$). Then, Line 9 temporarily computes the new availability times of dispatching J_i with p_i cores. Line 10 checks if dispatching J_i will delay the availability times of any of the values of the reserved cores, this condition allows lower-priority jobs to use these cores if they are guaranteed to finish before the number of reserved cores becomes available and thus, allowing a more efficient use of resources. If the condition is met, Lines 11 to 14 add the job to the eligible set with its respective cores and update the availability times to reflect the future dispatch of the job.
- **There are not enough cores but cores can still be reserved** (i.e., $p_i > (m - m^{busy}) \land p_i \leq (m - \sum_{q \in \mathcal{M}^{reserved}} q)$). In this case, $p_i + \sum_{q \in \mathcal{M}^{reserved}} q$ cores are reserved in Line 18. We use this value instead of p_i because otherwise we would be blocking valid executions when multiple jobs reserve cores.
- **There are not enough cores and cores cannot be reserved**. In this case, we stop looking for more jobs to schedule as if we continued trying to dispatch jobs we could have a priority-inversion issue (Line 20).

Algorithm 5: ResG eligible jobs selection algorithm.

```
input  : Set $\mathcal{R}$ of ready jobs
output : Set $\mathcal{J}^{eligible}$ of eligible jobs, set $\mathcal{M}^{eligible}$ of minimum number of
         cores assigned to jobs requried to meet the deadline and $\mathcal{M}^{reserved}$
         of reserved cores.
$m$ := total cores in the system;
$m^{busy}$ := number of cores executing jobs already;
$\mathcal{M}^{reserved} := \emptyset$ ; // Set of number of cores already reserved
$\mathcal{J}^{eligible} := \emptyset$;
Sort jobs in $\mathcal{R}$ by highest priority first;
foreach $J_i \in \mathcal{R}$ do
    $p_i$ = minimum cores to meet the deadline;
    if $p_i \le (m - m^{busy})$ then
        $A^{temp}$ := availability adding $J_i$ execution with $p_i$ cores to $A$;
        if checkAvailability($A^{temp}$, $A$, $\mathcal{M}^{reserved}$) then
            /* Enough cores are free and adding $J_i$ with $p_i$ cores will not
               increase the availability times of the number of reserved
               cores                                                        */
            $\mathcal{J}^{eligible} := \mathcal{J}^{eligible} \cup \{J_i\}$;
            $\mathcal{M}^{eligible} := \mathcal{M}^{eligible} \cup \{p_i\}$;
            $m^{busy} := m^{busy} + p_i$;
            continue;
        end
    end
    // Reached only if $J_i$ has not been selected in the previous step
    if $p_i > (m - m^{busy}) \wedge (p \le (m - \sum_{q \in \mathcal{M}^{reserved}} q))$ then
        // Too many cores are busy but there's space to reserve more
        $\mathcal{M}^{reserved} := \mathcal{M}^{reserved} \cup \{p_i + \sum_{q \in \mathcal{M}^{reserved}} q\}$;
    else
        /* This jobs cannot be scheduled right now and we cannot even
           reserve $p$ cores so we stop trying to mark more jobs as
           eligible.                                                        */
        break;
    end
end
Function checkAvailability($A^{temp}$, $A$, $\mathcal{M}^{reserved}$)
    foreach $q \in \mathcal{M}^{reserved}$ do
        if $A_q^{temp} > A_q$ then return false ;
    end
    return true
end
```

6.2.3 Moldable gang cores assignment

Because we are using moldable gang tasks, this means that we can decide how many cores the scheduler assigns to a job. In Section 6.2.2 Algorithm 5 computes the jobs that are eligible to be dispatched and also the minimum number of cores that each job needs in order to meet the deadline. However, now we can decide to assign more cores that will heavily benefit of using these extra cores.

For every free core, we select the best candidate job to have it assigned. We make sure that adding one more core to the selected job does not make the availability times of the reserved cores worse. We first search the job such that giving it one more core would make the availability times of the reserved cores better. If no job is found, we then give it to the job such that adding one more core would reduce the execution time defined as

$$C_i^{\max}(p+1) - C_i^{\max}(p) \tag{6.2}$$

So, we select the job that has the highest value when using Equation (6.2). Algorithm 6 shows the algorithm used to compute the set $\mathcal{M}^{moldable}$ of cores assigned to the jobs before being dispatched.

Algorithm 6: Assignment of cores of the ResG scheduling policy

```
input  : Set $\mathcal{J}^{eligible}$ of eligible jobs, $\mathcal{M}^{eligible}$ of cores assigned to the
         eligible jobs and $\mathcal{M}^{reserved}$ of reserved cores.
output : Set $\mathcal{M}^{moldable}$ of cores assigned to the eligible jobs.
$m :=$ total cores in the system;
$m^{free} := m - m^{busy} - \sum_{q \in \mathcal{M}^{eligible}} q$;
$\mathcal{M}^{moldable} = \mathcal{M}^{eligible}$;
while $m^{free} > 0$ do
    $J_i :=$ getBestCandidate($\mathcal{M}^{reserved}$, $\mathcal{J}^{eligible}$, $A$);
    if $J_i = \emptyset$ then
        break;
    end
    Extract $p_i$ from $\mathcal{M}^{moldable}$ and insert back $p_i = p_i + 1$;
    $m^{free} = m^{free} - 1$;
end
Function getBestCandidate($\mathcal{M}^{reserved}$, $\mathcal{J}^{eligible}$, $A$)
    foreach $J_i \in \mathcal{J}^{eligible}$ do
        if giving one extra core to $J_i$ improves any $A_q$, $q \in \mathcal{M}^{reserved}$ then
            return $J_i$;
        end
    end
    Sort jobs in $\mathcal{J}^{eligible}$ by value of Equation (6.2);
    foreach $J_i \in \mathcal{J}^{eligible}$ do
        if $J_i$ does not worsen $A_q$, $\forall q \in \mathcal{M}^{reserved}$ then
            return $J_i$;
        end
    end
    return $\emptyset$;
end
```

Chapter 7

Experimental evaluation

We performed an experimental evaluation to test whether the proposed test improves the accuracy of the schedulability analysis and also to see if the new ResG proposed scheduling policy increases the schedulability ratio when compared against JLFP. We show the results obtained for the non-preemptive analysis in Section 7.1, then we show the results for the limited-preemptive analysis in Section 7.2 and finally we compare the two scheduling policies in Section 7.3.

7.1 Non-preemptive analysis

When performing the evaluation of the non-preemptive analysis we wanted to test:

i) Whether the proposed test improves the accuracy of the schedulability analysis in comparison to the related work.

ii) Understand the influence of $m_i^{\min}$ and $m_i^{\max}$ on the schedulability of moldable gang tasks.

iii) Check whether the run-time of the analysis is practical.

The experiments were done by applying Algorithm 1 to the analysis of rigid and moldable gang tasks under a non-preemptive JLFP scheduling policy. We compared our results to the only other schedulability analysis for non-preemptive gang (NPGang) [15].

We implemented Algorithm 1 as a C++ program and performed the analysis on a cluster of the Clouding.io cloud computing service company. Each machine is equipped with 32GiB of RAM and AMD EPYC 7502 32-Core processors clocked at 2.5GHz. We report the *CPU time* as the run-time of the analysis.

Experiment	m	n	$m_i^{\min}$	$m_i^{\max}$	$\max U_i$
a-seq-random	8	20	1	$[1, m]$	1
a-seq-divisor				{1, 2, 4, 8}	
a-gang-random			$[1, m]$	$[1, m]$	$m \times U$
a-gang-divisor			{1, 2, 4, 8}	{1, 2, 4, 8}	
b	8	20	$m_i^{\min} = m_i^{\max}$ with {1, 2, 4, 6, 8}		$m \times U$
c	4	10	1	{1, 2, 3, 4}	1
d	8	20	1	{1, 2, 3, 4, 6, 8}	1
e	16	32	1	{4, 8, 16}	1
f	8	8–24	1	{1, 2, 3, 4, 6, 8}	1

Table 7.1: **Specification of experiments performed**

7.1.1 Experiments on synthetic task sets

We generated task sets using the same established method used in prior studies [10, 25, 17]. We randomly generated n utilization values with a total sum of $m \times U$ by using Stafford's RandFixSum algorithm [27] where the utilization U_i of each task falls in the interval $[0.001, m_i^{\min}]$. To avoid cases where the hyperperiod is impractically large due to incompatible task periods, we chose the period values with a log-uniform distribution in the interval $[10000, 100000]$ with a granularity of 5000 (as in [25]). Additionally, we discarded every task set that contains more than 100,000 jobs in its hyperperiod.

To evaluate the impact of $m_i^{\min}$ and $m_i^{\max}$ on the system schedulability, we assigned $m_i^{\min}$ and $m_i^{\max}$ differently for different experiments as detailed in Table 7.1. Experiments (a-seq-random) and (a-gang-random) pick their values randomly from the interval $[1, m]$. On the other hand, experiments (a-seq-divisor) and (a-gang-divisor) randomly pick their values from the set $\{1, 2, 4, 8\}$ composed of divisor of the number of cores $m = 8$. Note that we always ensure that $m_i^{\min} < m_i^{\max}$ when picking random values. Experiment (b) assumes a rigid gang model, so $m_i^{\min} = m_i^{\max}$ for all tasks. Finally, in experiments (c), (d), (e), and (f) all tasks share the same $m_i^{\min}$ and $m_i^{\max}$ values, where $m_i^{\min} = 1$ and $m_i^{\max}$ is varied from 1 to 16.

For each data point in the plots of Figure 7.1, we generated 450 task sets and report the *schedulability ratio* (i.e., the percentage of task sets deemed schedulable by the analysis). Additionally, we report the run-time of the schedulability analysis for each task set tested in experiments (c), (d), and (e) as a function of the number of jobs in their hyperperiod.

7.1.2 Schedulability results

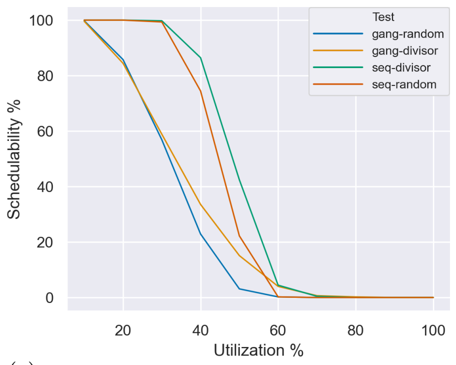
Figure 7.1b shows the schedulability ratio of rigid gang tasks as a function of the total system utilization and with different upper bounds on the maximum parallelism $m_i^{\max}$ of each task. We compare the test of Dong and Liu[15] to the non-preemptive SAG analysis of Chapter 4. The SAG analysis clearly outperforms Dong and Liu's test for any value of $m_i^{\max}$. For instance, Dong and Liu's test does not detect any schedulable task set at $U = 40\%$ while our analysis still detects between 95 and 100% of them depending on the value of the maximum task parallelism. Thus, the SAG analysis is clearly more accurate as it allows to explore all possible sequence of scheduler decisions whilst the test of [15] is only a utilization-based test.

To investigate how the task parallelism configuration affects the schedulability of *moldable* tasks, we compared four different scenarios in Figure 7.1a (see experimental setup). Task sets with $m_i^{\min}$ set to 1 for all tasks ((a-seq-random) and (a-seq-divisor)) have a higher schedulability ratio because tasks can be dispatched as soon as one or more cores become free. If $m_i^{\min} \geq 1$, however ((a-gang-random) and (a-gang-divisor)) jobs may experience longer blocking (waiting for their minimum number of cores to be freed) and frequent priority inversions with lower-priority jobs "stealing" available cores from higher priority ones. Additionally, we compared the difference between choosing $m_i^{\max}$ as a random value between 1 and the number of cores m, or when ensuring that it is a divisor of the number of cores (i.e., equal to 1, 2, 4, or 8). In the later case ((a-seq-divisor) and (a-gang-divisor)), the schedulability ratio slightly improves in comparison to the former. This slight improvement is caused by having a few less scenarios where cores are left idle with pending workload. However, the impact remains rather small.

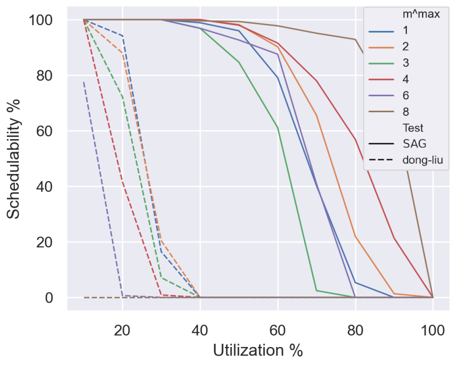
Figure 7.1f shows the effect of the number of *moldable* gang tasks when the total system utilization U is set to 70%. When $m_i^{\max} = 1$, the results are identical to non-preemptive global scheduling. Also, when $m_i^{\max} = 8$, the scheduler described in Section 3.2 will execute all jobs on $p = 8$ cores, which is equivalent to single core scheduling. Thus, as the number of tasks increases, the execution time of the tasks decreases and the results become closer to those of single core preemptive gang scheduling. Similarly, when $m_i^{\max}$ is set to 2 or 4 for all tasks, then the scheduler of Section 3.2 behaves identically toa non-preemptive global scheduler on 4 and 2 cores respectively, hence explaining the typical tendency witnessed for such systems. This experiment thus gives the impression that larger maximum task parallelism is beneficial for the schedulability.

We further explore this property in Figures 7.1c to 7.1e where we compare task sets with $m_i^{\min} = 1$ and a common $m^{\max}$ value for all tasks. It is interesting to see in Figure 7.1c that, in a system with four processors, configuring $m^{\max}$ to 3 causes a significant lower schedulability ratio than with other values. Also, with eight cores (Figure 7.1d), setting $m^{\max}$ to 3 or 6 also yields lower schedulability ratios (the same is true for $m^{\max} = 5$ and $m^{\max} = 7$, even though we do not show the results to avoid clutter). This shows the effect of all tasks sharing a value of $m^{\max}$ that is a divisor of the number of cores. When this is the case, all the jobs in the task set will always be scheduled with $p = m^{\max}$ because, as

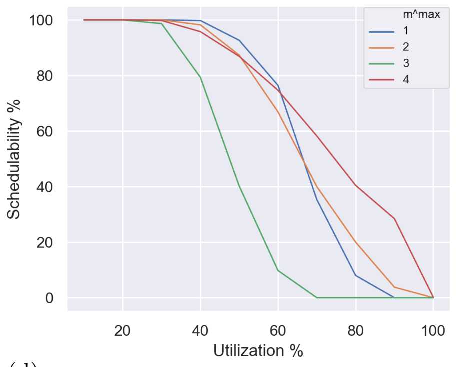
(a)



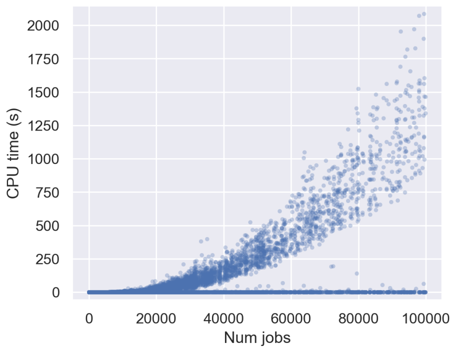
(b)



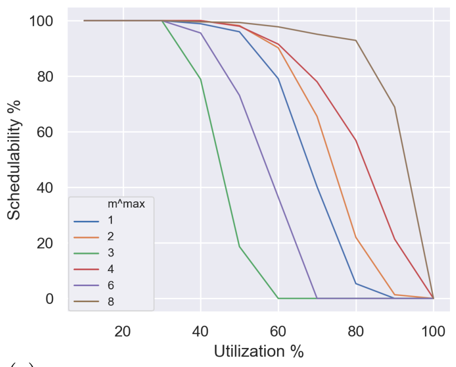
(c)



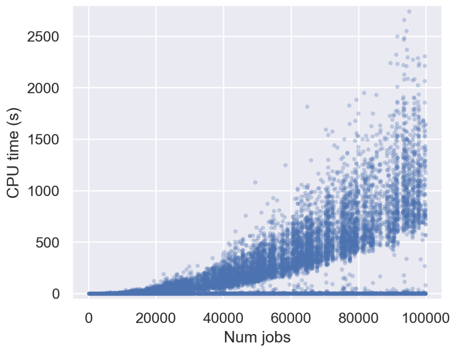
(c-runtime)



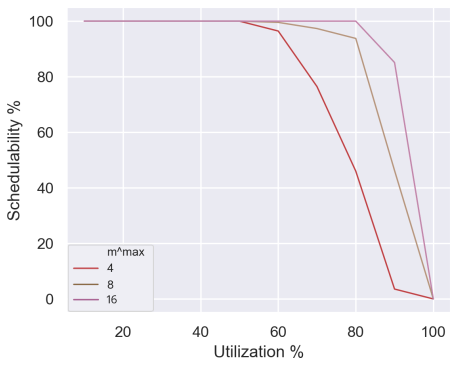
(d)



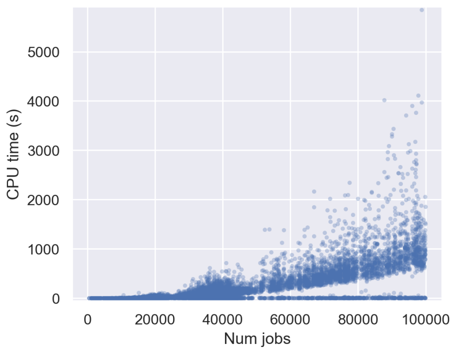
(d-runtime)



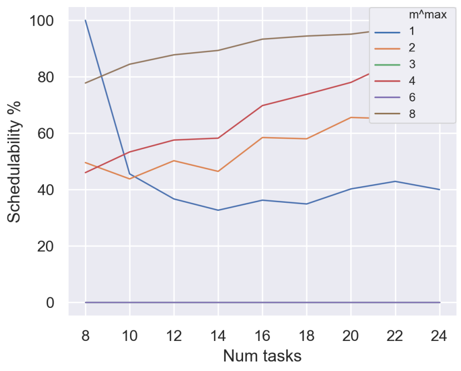
(e)



(e-runtime)



(f)



(f-runtime)

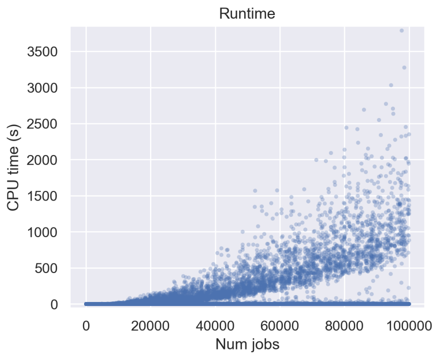


Figure 7.1: **Experimental results. (a) moldable gang tasks with** $m = 8$**, (b) rigid gang tasks with** $m = 8$**, (f) moldable gang tasks with** $m = 8$**,** $U = 0.7$ **and** $m_i^{\min} = 1$**, (c) moldable gang tasks with** $m = 4$ **and** $m_i^{\min} = 1$**, (d) moldable gang tasks with** $m = 8$ **and** $m_i^{\min} = 1$**, (e) moldable gang tasks with** $m = 16$ **and** $m_i^{\min} = 1$**. Runtime information for (c), (d), (e), and (f) is given by (c-runtime), (d-runtime), (e-runtime), and (f-runtime), respectively.**

soon as a job finishes, it will free exactly the same amount of cores as the next job needs to execute with $m^{\max}$ cores. This eliminates the problem of some cores being available but not being used as they are not enough for any pending job to start to execute. However, when $m^{\max}$ is not a divisor of m, some jobs may be executing with $m_i^{\max}$ cores while other may execute with smaller values of p, this causes an imbalance in execution times that leads to more frequent deadline misses.

Figure 7.1 also shows that the run-time of the SAG analysis remains acceptable even for large job sets scheduled on platforms with a large number of cores. The runtime remains well bellow $1{,}000s$ in the vast majority of the cases and the average runtime remains below $500s$.

7.2 Limited-preemptive analysis

When performing the evaluation of the limited-preemptive model we wanted to see whether the proposed test gives meaningful results and we wanted to understand the influence of the number of segments that each task has.

We performed experiments with the limited-preemptive execution model in a similar way as the one used in Section 7.1, using the same platform but applying Algorithm 1 to the analysis of moldable gang tasks under a limited-preemptive JLFP scheduling policy instead.

7.2.1 Experiments on synthetic task sets

We generated synthetic task sets using the same method explained in the previous section. As we wanted to evaluate the impact of the number of segments in a system, experiment (a) varies the number of segments assigned to each task. The task set has $n = 4$ tasks with a total system utilization U of 70% and $m = 4$ cores. Each task also has $m_i^{\min} = 1$ and $m_i^{\max}$ is set the same for all the tasks, then we compare four different values of $m_i^{\max}$ (1, 2, 3, and 4). Then, the number of segments evaluated are 1, 2, 3, 4, 5, 7, 10, 15, 20, and 25. We tried to obtain the same results with $m = 8$ cores and $n = 8$ but the memory requirements and runtime of the analysis became too large to complete.

Additionally, we performed another experiment by varying the system utilization instead of the number of segments. In this experiment, the total number of tasks is also $n = 4$ but we set the number of segments of each task to 5. The system utilization is varied between 10% and 100%.

7.2.2 Schedulability results

Figure 7.2a shows the schedulability ratio of limited-preemptive moldable gang tasks as a function of the number of segments of each task with upper bounds in the maximum parallelism of each task ($m_i^{\max}$). It can be seen how there's a clearly difference between $m_i^{\max} = 4$ and other values.

When the maximum parallelism is allowed, the execution becomes equivalent to a single core execution model. If this is the case, the case the set of certainly running predecessors ($\mathcal{X}_i^{pred}(v)$) always contains the previous scheduled job, except if there's a merge, and thus $t_i^{pred}(v)$ can be computed accurately. However, when the maximum parallelism is lower, it is possible that the set $\mathcal{X}_i^{pred}(v)$ is not totally accurate due to a previous merge and thus there's not enough information to give a more accurate $t_i^{pred}(v)$ value.

Figure 7.2b shows the schedulability ratio for limited-preemptive moldable gang tasks when varying the total system utilization. Again, it can be seen how reducing the maximum level of parallelism has a negative effect regarding schedulability due to pessimism in the analysis. Additionally by looking at the runtime of the analysis in Figure 7.2(b)-runtime it can be seen how some task sets reached the time limit (10000 seconds) when performing the evaluation. This increased runtime is caused by the increased number of segments in the system and more conditions that have to be checked when evaluating precedence constraints.

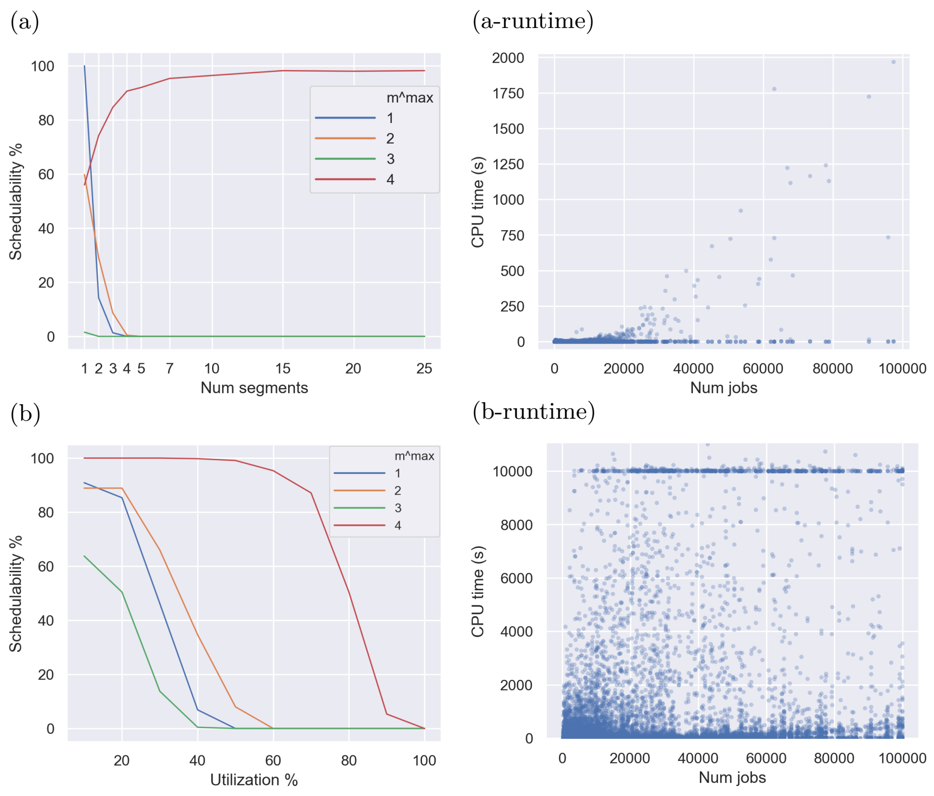


Figure 7.2: **Experimental results for the limited-preemptive execution model. (a) moldable gang tasks varying number of segments with $m = 4$, $U = 0.7$, $m_i^{\min} = 1$ and $n = 4$, (b) moldable gang tasks varying utilization with $m = 4$, $m_i^{\min} = 1$, $n = 4$ and 5 segments per task. Runtime information for (a) is given by (a-runtime).**

7.3 ResG compared with JLFP

We decided to evaluate the new ResG scheduling policy when compared with the JLFP scheduler. We did so by implementing the scheduling rules of JLFP (Section 3.2) and the rules of ResG (Section 6.2.2) in a simulator written in C++. It tests whether a task set will have a deadline miss or not by using the WCET ($C_i^{\max}$) of the jobs. While this is not as accurate as the SAG analysis, due to the schedulability of gang tasks not being sustainable, it still provides us with useful information that helped us to quickly test and develop new ideas that could improve the schedulability of moldable gang tasks.

We evaluated three different schedulers:

JLFP the job-level fixed-priority scheduler as defined in Section 3.2.

ResG-eligible The ResG scheduler but assigning a p value defined in $\mathcal{M}^{eligible}$, this is the minimum number of cores that ensures that a deadline miss will not happen.

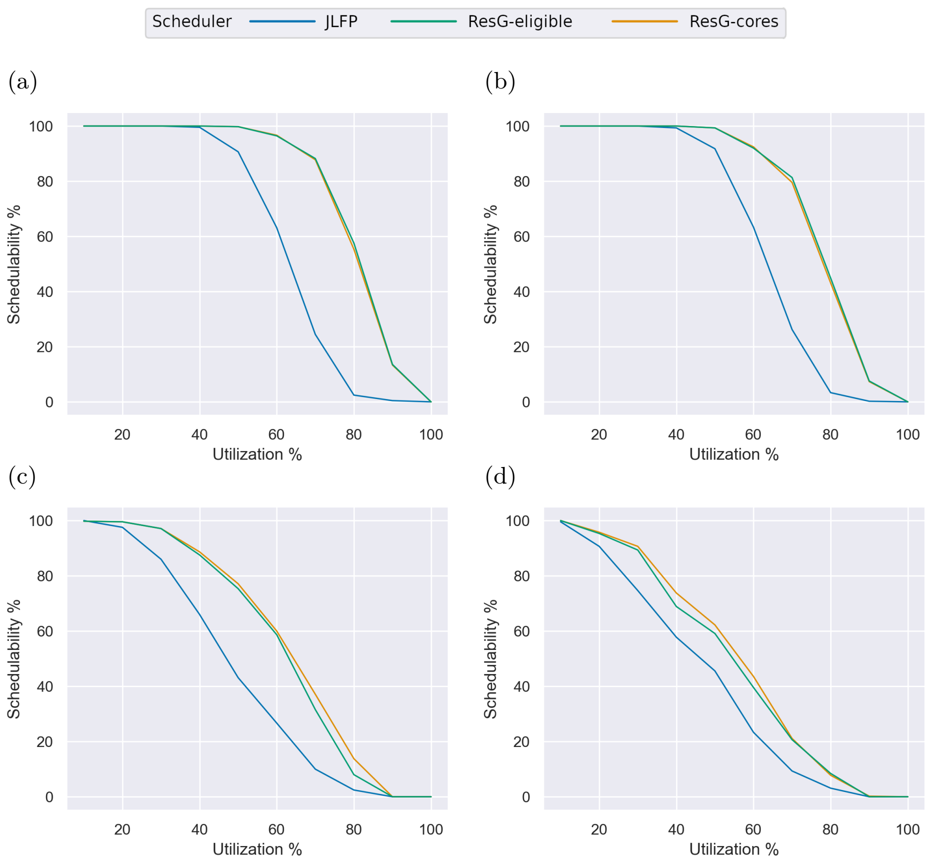


Figure 7.3: **Comparison of JLFP scheduling policy with two variants of ResG. The task sets used are the ones of Section 7.1 experiment (a). Task sets of (a) are (a-seq-random), of (b) are (a-seq-divisor), of (c) are (a-gang-random), and of (d) are (a-gang-divisor).**

ResG-moldable The ResG scheduler assigning a p value defined in $\mathcal{M}^{moldable}$, these are the cores such that all free cores have been assigned to a job.

Figure 7.3 shows the schedulability ratio of the JLFP, ResG-eligible, and ResG-moldable schedulers with four different types of task sets when evaluating them through the simulator that we created. We use the task sets defined in Section 7.1 experiment (a). It can be seen how there's a significant improvement in terms of schedulability ratio when comparing any of the ResG variants with JLFP. In Figure 7.3a the total schedulable task sets by ResG-moldable is a 17.2% higher than the ones schedulable by JLFP, in Figure 7.3b is a 13.8% higher, in Figure 7.3c is a 14.1% higher and in Figure 7.3d is a 9.1% higher.

It can be seen how there's a significative improvement just by using the non-work-conserving property of ResG when compared to JLFP. However, the effect of using $\mathcal{M}^{eligible}$ or $\mathcal{M}^{moldable}$ is not very significative (green and orange curves). This is caused because the non-work-conserving property already allocates much of the cores and already ensures that enough cores are allocated such that deadlines are not missed.

Chapter 8

Conclusions and future work

8.1 Conclusions

As a conclusion of this project we evaluate our initial research questions. We can say that we have significantly improved the accuracy of the schedulability analysis of gang tasks when compared to the current state-of-the-art. Moreover, we also extended the analysis for limited-preemptive moldable gang tasks which is a more generic model than the non-preemptive rigid model used by the state-of-the-art.

Additionally, by doing the analysis we were able to evaluate the impact of the moldable property of gang tasks, specially the impact of a good selection of the minimum and maximum parallelism assigned to each task in the system. We concluded that assigning the same $m_i^{\min}$ value to all tasks yields the best performance, specially when these values are a divisor of the total number of cores in the system.

Finally, we show that a non-work-conserving policy can significantly improve the schedulability ratio of the task sets when compared against the job-level fixed-priority scheduling policy. Moreover, this effect is also combined with a good selection of number of cores so deadlines are always met.

8.2 Future work

We plan on improving our analysis with precedence constraints in order to increase the accuracy when multiple jobs run simultaneously by storing additional information in the system state that would allow us to properly track potentially running jobs, required to do a proper pruning of branches when lower priority jobs are evaluated for a possible dispatch.

Moreover, we would like to improve the ResG scheduling policy by testing more ways of assigning the number of cores allocated to a job. We would like to see if focusing on other objectives when assigning the number of cores, would lead a better schedulability ratio for different types of task sets.

Notation

We provide the following list of common notation used during the thesis.

Description	Symbol
.	Decimal separator
,	Thousands separator
v_i	State i
m	Cores in the system
n	Total tasks in the system
τ_i	Task i
J_i	Job i
$\mathcal{J}$	Job set
$\mathcal{S}(v)$	Set of dispatched jobs in state v
$\mathcal{R}(v)$	Set of ready jobs in state v
$m_i^{\min}$	Minimum cores of job J_i
$m_i^{\max}$	Maximum cores of job J_i
p_i	Cores assigned to job J_i
$C_i^{\min}(p)$	Best-case execution time of job J_i when executed with p cores
$C_i^{\max}(p)$	Worst-case execution time of job J_i when executed with p cores
$r_i^{\min}$	Earliest release time of job J_i
$r_i^{\max}$	Latest release time of job J_i
$R_i^{\min}(v)$	Earliest release time of job J_i with precedence constraints at state v
$R_i^{\max}(v)$	Latest release time of job J_i with precedence constraints at state v
d_i	Deadline of job J_i

hp_i	Set of jobs with higher priority than job J_i
lp_i	Set of jobs with lower priority than job J_i
$pred(J_i)$	Set of predecessors of job J_i
$\mathcal{X}(v)$	Set of certainly running jobs at state v
$\mathcal{X}_i^{pred}(v)$	Set of certainly running predecessors of J_i at state v
$A_k^{\min}(v)$	Possibly available times of k cores at state v
$A_k^{\max}(v)$	Certainly available times of k cores at state v
$A_k^{exact}(v)$	Possibly available times of *exactly* k cores at state v
$\mathcal{F}(v)$	Set of simultaneous core releases at state v
$F_i(v)$	Element i-th in set of simultaneous core releases $\mathcal{F}(v)$ of state v
$f_i(v)$	Earliest availability time of group of cores represented by element $F_i(v)$
$M_i(v)$	Number of cores in group of cores represented by element $F_i(v)$
$EST_i^p(v)$	Earliest start time of J_i with p cores in state v
$LST_i^p(v)$	Latest start time of J_i with p cores in state v
$EFT_i^p(v)$	Earliest finish time of J_i with p cores in state v
$LFT_i^p(v)$	Latest finish time of J_i with p cores in state v

Acronyms

BCET Best-Case Execution Time 10, 16, 51

BCRT Best-Case Response Time 3, 7, 23

BTM Bundled Task Model 6

DAG Directed Acyclic Graph 9, 11, 13

DM Deadline Monotonic 1

EDF Earliest-Deadline First 1, 6

FPGA Field Programmable Gate Array 2

GPU Graphics Processor Unit 2

HPC High-Performance Computing 1, 5

JLFP Job-Level Fixed-Priority 1, 3, 6, 7, 10, 33–35, 41, 45, 47–49

NPGang Non-Preemptive Gang 41

ResG Reservation-Based Gang 35–39, 41, 47–49

SA Schedule-Abstraction 7

SAG Schedule-Abstraction Graph ix, 3–5, 7, 16, 17, 22, 23, 27, 43, 45, 47, 63

TLFP Task-Level Fixed-Priority 6

WCET Worst-Case Execution Time 10, 16, 20, 35, 47, 51

WCRT Worst-Case Response Time 3, 7, 13, 23

Bibliography

[1] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer US, 2010. ISBN: 9781441935786.

[2] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *3rd International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.

[3] Jing Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 85–96, 2014.

[4] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992. ISSN: 07437315. DOI: `10.1016/0743-7315(92)90014-E`.

[5] Joël Goossens and Vandy Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. In *18th International Conference on Real-Time and Network Systems (RTNS)*, pages 189–196, 2010. arXiv: `1006.2617`.

[6] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson and F. Donelson Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017. ISBN: 9781538614143. DOI: `10.1109/RTSS.2017.00017`.

[7] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni and Giorgio Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2016.

[8] Sébastien Collette, Liliana Cucu and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008. ISSN: 00200190. DOI: `10.1016/j.ipl.2007.11.014`.

[9] Zheng Dong and Cong Liu. Analysis techniques for supporting hard real-time sporadic gang task systems. *Real-Time Systems*, 55(3):641–666, 2019. ISSN: 15731383. DOI: `10.1007/s11241-018-9318-7`.

[10] Joël Goossens, Emmanuel Grolleau and Liliana Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-Time Systems*, 52(6):808–832, 2016.

[11] Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In *2009 IEEE Real-Time Systems Symposium (RTSS)*, pages 459–468. IEEE, 2009. ISBN: 9780769538754. DOI: `10.1109/RTSS.2009.42`.

[12] Pascal Richard, Joël Goossens and Shinpei Kato. Comments on "Gang EDF Schedulability Analysis", 2017. arXiv: `1705.05798`.

[13] Vandy Berten, Pierre Courbin and Joël Goossens. Gang fixed priority scheduling of periodic moldable real-time tasks. In *19th International Conference on Real-Time and Network Systems (RTNS)*, pages 9–12, 2011.

[14] Saud Wasly and Rodolfo Pellizzoni. Bundled scheduling of parallel real-time tasks. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, volume 2019-April, pages 130–142. IEEE, 2019. ISBN: 978-1-728-10678-6. DOI: `10.1109/RTAS.2019.00019`.

[15] Zheng Dong and Cong Liu. Work-in-progress: Non-preemptive scheduling of sporadic gang tasks on multiprocessors. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, volume 2019-Decem, pages 512–515. IEEE, 2019. ISBN: 9781728144030. DOI: `10.1109/RTSS46320.2019.00052`.

[16] Mitra Nasri and Björn B. Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, 2017.

[17] Mitra Nasri, Nelissen Geoffrey 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)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 21:1–21:23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. ISBN: 978-3-95977-110-8. DOI: `10.4230/LIPIcs.ECRTS.2019.21`.

[18] Jacek Blazewicz, Mieczyslaw Drabowski and Jan Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, c-35(5):389–393, 1986.

[19] Dror G. Feitelson and Morris A. Jettee. Improved utilization and responsiveness with gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 238–261, Berlin, Heidelberg. Springer Berlin Heidelberg, 1997. ISBN: 978-3-540-69599-8.

[20] Yanyong Zhang, H. Franke, J. Moreira and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, 2003.

[21] Dror G. Feitelson. Packing schemes for gang scheduling. In *Lecture Notes in Computer Science. Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 89–110, 1996. ISBN: 978-3-540-70710-3. DOI: `10.1007/bfb0022289`.

[22] Yair Wiseman and Dror G. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):581–592, 2003. ISSN: 10459219. DOI: `10.1109/TPDS.2003.1206505`.

[23] Felipe Cerqueira, Geoffrey Nelissen and Björn B Brandenburg. On strong and weak sustainability, with an application to self-suspending real-time tasks. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS)*, 26:1–26:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. ISBN: 978-3-95977-075-0. DOI: `10.4230/LIPIcs.ECRTS.2018.26`.

[24] Beyazit Yalcinkaya, Mitra Nasri and Björn B. Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*, pages 1222–1227, 2019.

[25] Mitra Nasri, Geoffrey Nelissen and Björn B. Brandenburg. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*, 9:1–9:23, 2018.

[26] Konstantinos Koiliaris and Chao Xu. Faster Pseudopolynomial Time Algorithms for Subset Sum. *ACM Transactions on Algorithms*, 15(3):1062–1072, 2019. ISSN: 15496333. DOI: `10.1145/3329863`.

[27] Roger Stafford. Random vectors with fixed sum. Technical Report, University of Oxford, 2006. URL: `http://www.mathworks.com/matlabcentral/fileexchange/9700`.

Appendix A

Proofs for non-preemptive analysis

Lemma 1. *Job J_i cannot start executing with p cores before $t^p_{gang}(v)$ defined as*

$$t^p_{gang}(v) = \begin{cases} A^{\min}_p(v) & \textit{if } p = m_i^{\min}, \\ A^{exact}_p(v) & \textit{otherwise} \end{cases} \tag{4.1}$$

Proof of Lemma 1. We analyse two cases:

- If $p = m_i^{\max}$, job J_i cannot start until at least p are available. By definition of availability this happens at $A^{\min}_p(v)$. Thus, proving the claim for the case $p = m_i^{\max}$ in Equation (4.1).
- If $p < m_i^{\max}$, job J_i cannot start executing on p cores until *exactly* p are available (i.e., at $A^{exact}_p(v)$) since by rule 5 it would be dispatched on less, respectively more, than p if there are less, respectively more, than p available. Thus proving the claim for the case $p < m_i^{\max}$.

■

Corollary 1. *A job J_i cannot start executing on exactly p cores before time $EST^p_i(v)$, defined as*

$$EST^p_i(v) = \max\{r_i^{\min}, t^p_{gang}(v)\} \tag{4.2}$$

Proof of Corollary 1. Job J_i cannot start before being released (i.e., before $r_i^{\min}$) and cannot start executing with p cores before $t^p_{gang}(v)$ as proved in Lemma 1. Thus, J_i cannot start before $\max\{r_i^{\min}, t^p_{gang}(v)\}$, thus proving the claim. ■

Lemma 2. *J_i will not be the first job dispatched in the system state v or will not be dispatched on exactly p cores if it did not start to execute before time*

$t^p_{high}(v)$ defined as[1]

$$t^p_{high}(v) = \min^{\infty}_{J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}} \left\{ t^p_h(J_i, J_j) \right\} \tag{4.8}$$

where

$$t^p_h(J_i, J_j) = \begin{cases} r^{\max}_j & \text{if } m^{\min}_j \leq p \\ \max\{r^{\max}_j, A^{\max}_{m^{\min}_j}\} & \text{otherwise} \end{cases} \tag{4.9}$$

Proof of Lemma 2. We prove that a not-yet-scheduled higher-priority job J_j (i.e., $J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}$) will be dispatched before J_i if J_i did not start to execute before $t^p_h(J_i, J_j)$. It then directly follows that a not-yet-scheduled higher-priority job will be dispatched before J_i if J_i did not start to execute before $t^p_{high}(v) = \min^{\infty}_{J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}} \left\{ t^p_h(J_i, J_j) \right\}$ hence proving the lemma.

We consider two cases:

1. If $m^{\min}_j \leq p$, then the higher priority job J_j requires fewer cores than the number of cores requested by job J_i. Thus, if job J_j is released when J_i becomes eligible, then according to rule 2, J_j is also eligible, and because J_j has a higher priority than J_i, the scheduler will dispatch J_j instead of J_i (rule 4). Therefore, J_i cannot be scheduled before J_j on p cores if it did not start to execute before $r^{\max}_j$. This proves that J_j will be dispatched before J_i if J_i did not start to execute before $t^p_h(J_i, J_j)$.
2. If $m^{\min}_j > p$, then, according to rule rule 2, the higher priority job J_j will become eligible when it is released and when $m^{\min}_j$ cores are available. This happens at the latest at time $\max\{r^{\max}_j, A^{\max}_{m^{\min}_j}(v)\}$. Then, because J_j has a higher priority than J_i, the scheduler will dispatch J_j first if J_i did not start to execute before $\max\{r^{\max}_j, A^{\max}_{m^{\min}_j}(v)\}$ (rule 4). Therefore, we proved that J_j will be dispatched before J_i if J_i did not start to execute before $t^p_h(J_i, J_j)$.

■

Corollary 2. *Job J_i cannot be dispatched on p cores and be the first job dispatched in state v later than*

$$LST^p_i(v) = \min\{t^p_{avail}(v), t_{wc}(v), t^p_{high}(v) - 1\} \tag{4.10}$$

Proof of Corollary 2. It directly follows from the combination of Equations (4.3), (4.5) and (4.7). ■

Theorem 1. *A job J_i may be dispatched on p cores and be the first job dispatched by the scheduler in system state v only if $EST^p_i(v) < \infty$ and inequality 4.11 is respected.*

Proof of Theorem 1. It is obvious that the earliest start time $EST^p_i(v)$ of J_i must be smaller than ∞ to ensure that J_i may start to execute in system state v. Hence, we focus on inequality 4.11. By contradiction, assume that:

[1] $\min^{\infty}_{x \in S}\{x\} = +\infty$ if $S = \emptyset$. Otherwise, $\min^{\infty}_{x \in S}\{x\} = \min_{x \in S}\{x\}$

i) A job J_i, is the first job dispatched by the scheduler in system state v.

ii) That, J_i is assigned p cores by the scheduler

iii) That, J_i does not respect inequality 4.11.

Let t_s be the time at which J_i starts executing. By Corollary 1, we have that $t_s \geq estipv$. Thus, by assumption (iii) and the definition of $LST_i^p(v)$ given in Corollary 2, we have $t_s > t_{avail}^p(v)$ or $t_s > t_{wc}(v)$ or $t_s \geq t_{high}^p(v)$. We analyse each case independently.

- $t_s > t_{avail}^p(v)$. Since by Equation (4.4) $t_{avail}^p(v) \geq A_p^{\max}(v)$, at least $p+1$ cores are available at time t_s. Therefore, by rule 5, J_i is dispatched on at least $p+1$ cores. This contradicts the assumption (ii) that J_i is dispatched on p cores.
- $t_s > t_{wc}(v)$. By definition of $t_{wc}(v)$, a job certainly becomes eligible to be dispatched by time $t_{wc}(v)$. Therefore, a job must have been dispatched by the scheduler at or before $t_{wc}(v)$. This contradicts the assumption (i) that J_i is the first job dispatched by the scheduler and J_i is dispatched at time t_s.
- $t_s \geq t_{high}^p(v)$. By Lemma 2, J_i is not the highest eligible priority job at time t_s. Thus, by rule 4 it is not the first job dispatched by the scheduler, hence contradicting the assumption (i).

This proves the claim. ∎

Lemma 3. *Let node v'_k results from executing J_i on the p cores in $\mathcal{F}_k^{=p}$, then the set of earliest simultaneous core releases is*

$$\mathcal{F}_{v'_k} = \left\{ \langle EFT_i^p(v), p \rangle \right\} \cup \left\{ \mathcal{F}(v) \setminus \mathcal{F}_k^{=p} \right\} \tag{4.14}$$

Proof of Lemma 3. Since v'_k considers a system state that result from dispatching job J_i on p cores, p cores will be released simultaneously by J_i when it finishes its execution. This happens no earlier than the earliest finish time $EFT_i^p(v)$ of J_i. Therefore, $\mathcal{F}_{v'_k} \supseteq \{\mathcal{F}(v) \setminus \mathcal{F}_k^{=p}\}$.

Furthermore, because by assumption J_i executes on the cores in $\mathcal{F}_k^{=p}$, the time at which the cores in $\mathcal{F}(v) \setminus \mathcal{F}_k^{=p}$ are released is not impacted by the execution of J_i. Thus, $\mathcal{F}_{v'_k} \supseteq \{\mathcal{F}(v) \setminus \mathcal{F}_k^{=p}\}$. ∎

Lemma 4. *If all the cores in $\mathcal{F}_k^{\geq p}$ are released when J_i starts to execute, then J_i starts no earlier than $t_k = \max_{F_l \in \mathcal{F}_k^{\geq p}} \{f_l(v)\}$.*

Proof of Lemma 4. By definition of F_l, the $M_l(v)$ cores modelled by F_l are all released at the earliest at time $f_l(v)$. Thus, all the cores in $\mathcal{F}_k^{\geq p}$ are available no earlier than $\max_{F_l \in \mathcal{F}_k^{\geq p}} \{f_l(v)\}$. Since J_i starts when all cores in $\mathcal{F}_k^{\geq p}$ are available, this proves the claim. ∎

Lemma 5. *Let node v'_k result from executing J_i on p of the cores in $\mathcal{F}_k^{\geq p}$, then the set of earliest simultaneous core releases is*

$$\mathcal{F}_{v'_k} = \left\{ \langle EFT_i^p(v), p \rangle \right\} \cup \left\{ \langle t_k, (s-p) \rangle \right\} \cup \left\{ \mathcal{F}(v) \setminus \mathcal{F}_k^{\geq p} \right\} \tag{4.15}$$

where s is the number of cores in $\mathcal{F}_k^{\geq p}$, i.e., $s = \sum_{F_l \in \mathcal{F}_k^{\geq p}} M_l(v)$.

Proof of Lemma 5. Since v'_k considers a system state that results from dispatching job J_i on p cores, p cores will be released simultaneously by J_i when it finishes its execution. This happens no earlier than the earliest finish time $EFT_i^p(v)$ of J_i. Therefore, $\mathcal{F}_{v'_k} \supseteq \big\{\langle EFT_i^p(v), p\rangle\big\}$.

Furthermore, by assumption, all cores in $\mathcal{F}_k^{\geq p}$ are free when J_i starts to execute. Therefore, all $(s-p)$ cores in $\mathcal{F}_k^{\geq p}$ on which J_i does not execute are free from J_i's start time onward. By Lemma 4, J_i starts no earlier than t_k. Hence, $\mathcal{F}_{v'_k} \supseteq \big\{\langle t_k, (s-p)\rangle\big\}$.

Finally, because J_i executes on the cores in $\mathcal{F}_k^{\geq p}$, the time at which the cores in $\mathcal{F}(v) \setminus \mathcal{F}_k^{\geq p}$ are released is not impacted by the execution of J_i. Thus, $\mathcal{F}_{v'_k} \supseteq \big\{\mathcal{F}(v) \setminus \mathcal{F}_k^{\geq p}\big\}$. ■

Lemma 6. *A set of lower bounds on the time instants at which each core may potentially become available to execute new workload in v'_k is given by*

$$PA = \Big\{p \times \{EFT_i^p(v)\}\Big\} \cup \Big\{\max\{A_x^{\min}(v), t_k\} \big| p < x \leq m\Big\} \tag{4.16}$$

Proof of Lemma 6. First, since v'_k considers a system state that result form dispatching job J_i on p cores, at least p cores will become available no earlier than the earliest finish time $EFT_i^p(v)$ of J_i. Therefore PA must contain p times $EFT_i^p(v)$.

Second, by rule 8, J_i will always be dispatched on the p first cores that become available. Therefore, the earliest time at which the $(m-p)$ remaining cores may become available is given by the earliest time at which the $(m-p)$ latest cores could become available before dispatching J_i. By definition of the availability intervals, those times are given by $\{A_x^{\min}(v) | p < x \leq m\}$.

Finally, since job J_i is the first job dispatched by the scheduler in state v, and because by Lemma 4 t_k is the earliest time at which J_i is dispatched, cores can start to execute new workload no earlier than t_k in v'_k.

Combining the three facts above, we prove the claim. ■

Corollary 3. *A lower bound on the time at which x cores are potentially available to execute new workload in v'_k (i.e., $A_x^{\min}(v)$) is given by the x^{th} element in the non-decreasingly ordered set PA.*

Proof of Corollary 3. Since PA contains a lower bound on the availability time of every core in system state v'_k, the x^{th} element in the ordered set is a lower bound on the availability time of x cores. ■

Lemma 7. *A set of upper bounds on the time instants at which each core will certainly become available to execute new workload in v'_k is given by*

$$CA = \Big\{p \times \{LFT_i^p(v)\}\Big\} \cup \Big\{\max\{A_x^{\max}(v), t_k\} \big| p < x \leq m\Big\} \tag{4.17}$$

Proof of Lemma 7. Since v'_k represents a system state resulting from dispatching job J_i on p cores, there must be at least p cores that will become available to execute new workload no later than the latest finish time of J_i. That is, there must be p values no smaller than $LFT_i^p(v)$ in CA, i.e., $CA \supseteq \{p \times LFT_i^p(v)\}$.

Furthermore, all $(m-p)$ cores that do not execute J_i will be freed no later than the certain availability time of the $(m-p)$ latest cores that become available in the initial system state v (i.e., the system state before dispatching J_i). Those times are given by $\{A_x^{\max}(v) | p < x \leq m\}$.

Finally, since job J_i is the first job dispatched by the scheduler in v, and because t_k is the earliest time at which J_i is dispatched (Lemma 4), cores can start to execute new workload no earlier than t_k in v'_k.

Combining all the above, we prove the lemma. ■

Corollary 4. *An upper bound on the time at which x cores are certainly available to execute new workload in v'_k (i.e., $A_x^{\max}(v)$) is given by the x^{th} element in the non-decreasingly ordered set CA.*

Proof of Corollary 4. Same proof as Corollary 3, replacing PA with CA. ■

Lemma 8. *If exactly p cores may be available at time t in either v_k or v_q, then exactly p cores may be available at time t in v_z.*

Proof of Lemma 8. Assume that v refers to either v_k or v_q. Each group of cores $F_l(v) \in \mathcal{F}(v)$ is subdivided in one or several smaller groups of cores in $\mathcal{F}(v_z)$ (Lines 3 to 10 in Algorithm 3), that is $\exists \mathcal{F}' \subseteq \mathcal{F}(v_z)$, $\sum_{F_x(v_z) \in \mathcal{F}'} M_x(v_z) = M_l(v)$. Furthermore, each of the group of cores in the subset $\mathcal{F}'$ has an *earliest* release time that is earlier than or at the same time as that of F_l (Lines 12 to 16 in Algorithm 3), i.e., $\forall F_x(v_z) \in \mathcal{F}',\ f_x(v_z) \leq f_l(v)$.

Since for every group of cores that can be simultaneously released at a given time t in v there is a set $\mathcal{F}'$ in v_z composed of the same number of cores, each with an earliest release time no later than t, it then holds that the cores in $\mathcal{F}'$ can also be simultaneously released at t. This proves the lemma. ■

Theorem 2. *For any possible execution scenario such that J_i executes on p cores and finishes at t, there is a path $\langle v_1, \ldots, v_k \rangle$ in the schedule-abstraction graph such that J_i passes the dispatch condition on p cores in v_k and $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$.*

Proof of Theorem 2. Assume that the availability intervals and the set of earliest simultaneous core releases $\mathcal{F}(v_k)$ of state v_k safely model the actual availability times and simultaneous releases of the m cores resulting from the sequence of scheduling decisions encoded in the path $\langle v_1, \ldots, v_k \rangle$.

We prove that $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$, that J_i passes the dispatch condition in v_k and that each state v'_k created by Algorithm 2 because of executing J_i on p cores in v_k, correctly models the actual availability times and simultaneous releases of the cores after executing J_i on p cores.

Under the inductive assumption stated above, Corollaries 1 and 2 prove that $EST_i^p(v_k)$ prove that $EST_i^p(v_k)$ and $LST_i^p(v_k)$ are safe lower- and upper-bounds on the start time of J_i on p cores in v_k, respectively. Furthermore, since gang jobs are non-preemptive,Equations (4.12) and (4.13) are safe lower- and upper-bounds on t (i.e., $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$). Moreover, since $t \in [EFT_i^p(v_k), LFT_i^p(v_k)]$, it must hold that $EFT_i^p(v_k) \leq LFT_i^p(v_k)$, and thus the condition of inequality 4.11 is respected. Then, Lemmas 3 and 5 and Corollaries 3 and 4 prove that the simultaneous releases of cores and their availability is correctly modelled in each newly created state v_k' resulting from scheduling J_i on p cores. Therefore, the inductive assumption is respected for v_k'. Also, according to Lemma 8, potentially merging v_k' with another node in Algorithm 1 maintains the validity of the inductive assumption.

Finally, since all cores are assumed to be free in the initial system state, the inductive assumption (i.e., correct availability intervals and simultaneous core releases) obviously holds for v_1 and thus follows by induction on all the states created by Algorithm 1. ■

Appendix B

Proofs for limited-preemptive analysis

Lemma 9. *The set of certainly running predecessors $\mathcal{X}_x^{pred}(v)$ of a job J_x in state v is defined as*

$$\mathcal{X}_x^{pred}(v) = \{pred(x) \cap \mathcal{X}(v)\} \tag{5.4}$$

Proof of Lemma 9. By definition of $pred(J_x)$ and $\mathcal{X}(v)$ these are the jobs that are predecessors of J_x and that are also certainly running at state v. ■

Lemma 10. *The number of cores possibly available at a time t in state v can be computed as*

$$q^{\min}(v,t) = \max_{1 \le k \le m} \left\{ k | t \ge A_k^{\min}(v) \right\} \tag{5.5}$$

Proof of Lemma 10. By definition of availability the number of possibly available cores at time t is the maximum value k such that k cores have possibly become available at time t, i.e., $t \ge A_k^{\min}(v)$. ■

Lemma 11. *At time t in state v, the set of waiting jobs $J_k \in \mathcal{Q}_i(v,t)$ that have to be checked because J_i being dispatched could mean that the predecessors of J_k have finished their execution is defined as*

$$\mathcal{Q}_i(v,t) = \left\{ J_k \middle| \underbrace{\text{can\_start}\,(J_k, v, t)}_{J_k \textit{ can start at time } t} \wedge \underbrace{\mathcal{X}_k^{pred}(v) \neq \emptyset}_{\substack{\textit{Has a certainly} \\ \textit{running predecessor}}} \wedge\ J_k \in \text{hp}_i \right\} \tag{5.6}$$

where

$$\text{can\_start}\,(J_x, v, t) = \Big(t \ge \max\Big\{r_x^{\max}, A_{m_x^{\min}}^{\min}(v)\Big\}\Big) \wedge \Big(pred(J_x) \subseteq \mathcal{S}(v)\Big) \tag{5.7}$$

Proof of Lemma 11. Jobs $J_k \in \mathcal{Q}_i(v,t)$ are jobs that have a higher priority than J_i and that are waiting for their predecessors to finish execution. Thus, depending on the conditions, J_i being dispatched at time t would mean that they should have been dispatched instead as that would imply that their predecessors have finished execution. Then, the properties of these jobs $J_k \in \mathcal{Q}_i(v,t)$ are:

- If J_i can start, so does J_k. This means that J_k has been released, at least $m_k^{\min}$ cores are available and all the predecessors of J_k have been dispatched already.
- J_k has certainly running predecessors. This means that J_k is only waiting for its predecessors to finish execution.
- J_k has a higher priority than J_i. If that weren't the case, every possible dispatch scenario of J_i would be possible and would never imply that J_k should have been dispatched instead.

Combining the three conditions above that have to be met by a $J_k \in \mathcal{Q}_i(v,t)$ we prove the lemma. ■

Lemma 12. *If job J_i can start at time t in state v a higher-priority job $J_k \in \mathcal{Q}_i(v,t)$ will also certainly be able to start if all possible allocations of cores for J_i require for each of the certainly running predecessors $J_j \in \mathcal{X}_k^{pred}(v)$ at least one core of the cores previously being used by J_j.*

Proof of Lemma 12. By contradiction, let's assume that J_i is dispatched at time t in state v, and in order to do so, all possible core allocations require for each of the certainly running predecessors $J_j \in \mathcal{X}_k^{pred}(v)$ of $J_k \in \mathcal{Q}_i(v,t)$ at least one core previously being used for each of the J_j. If that is the case, this means that all the certainly running predecessors $J_j \in \mathcal{X}_k^{pred}(v)$ have finished their execution and thus, by rule 1, J_k is ready. Moreover, since J_k belongs to the set $\{pred(i) \cap \mathcal{X}(v)\}$ of higher-priority jobs with a certainly running predecessor that can start if J_i can start this means that, by rule 2, J_k is also eligible and, by rule 4, J_k is dispatched before J_i. This contradicts the assumption that J_i is scheduled at t and thus proves the claim. ■

Lemma 13. *The minimum number of cores $p_k^{pred}(v)$ that ensures that all possible allocations of cores for J_i certainly use at least one core from each of the cores freed by certainly running predecessors of a job $J_k \in \mathcal{Q}_i(v,t)$ is*

$$p_k^{pred}(v) = \min_{J_j \in \mathcal{X}_k^{pred}(v)} p_j \tag{5.8}$$

Proof of Lemma 13. By contradiction, let's assume that $p_k^{pred}(v)$ is not the minimum among all the certainly running predecessors $\mathcal{X}_k^{pred}(v)$ of J_k. This means that J_i cannot be scheduled with more than $q^{\min}(v,t) - p_k^{pred}(v)$ cores without using cores from all the certainly running predecessors of J_k. As we assumed that $p_k^{pred}(v)$ is not the minimum among all the jobs J_j that are certainly running predecessors of J_k, this means that there's a job J_x such that $p_x < p_k^{pred}(v) \wedge J_x \in \mathcal{X}_k^{pred}(v)$. This means that it is actually possible to scheduled J_i using $p_k^{pred}(v) - p_x$ without using cores from the group of cores of J_x. This contradicts the assumption that $p_k^{pred}(v)$ is not the minimum among the number of cores of groups of cores freed by certainly running predecessors $J_j \in \mathcal{X}_k^{pred}(v)$ of job $J_k \in \mathcal{Q}_i(v,t)$ and thus proves our claim. ■

Corollary 5. *The job $J_k^{pred}(v) \in \mathcal{X}_k^{pred}(v)$ from the certainly running predecessors of a higher-priority job $J_k \in \mathcal{Q}_i(v,t)$ that has the minimum number of simultaneously freed cores where J_i cannot be scheduled is*

$$J_k^{pred}(v) = \underset{J_j \in \mathcal{X}_k^{pred}(v)}{\arg\min}\ p_j \tag{5.9}$$

Proof of Corollary 5. The proof is the same as in Lemma 13 but now we take the argument of the minimum instead of the minimum itself. ■

Lemma 14. *The set of jobs $\mathcal{Q}_i^{pred}(v,t)$ whose number of cores cannot be used by J_i is defined as*

$$\mathcal{Q}_i^{pred}(v,t) = \left\{ J_k^{pred}(v) \middle| J_k \in \mathcal{Q}_i(v,t) \right\} \tag{5.10}$$

Proof of Lemma 14. This proof follows from Corollary 5. As there are multiple higher-priority jobs J_k each of them has a job $J_k^{pred}(v)$ with a minimum number of cores that cannot be used by the lower priority job J_i. Note that $\mathcal{Q}_i^{pred}(v,t)$ is a set (i.e., no repetitions) so if multiple $J_k \in \mathcal{Q}_i(v,t)$ have the same minimum predecessor the condition is not counted twice. ■

Corollary 6. *The total number of cores that cannot be used by J_i if it can be scheduled at the earliest at time t is*

$$q_i^{pred}(v,t) = \sum_{J_j \in \mathcal{Q}_i^{pred}(v,t)} p_j \tag{5.11}$$

Proof of Corollary 6. By definition of $\mathcal{Q}_i^{pred}(v,t)$, each job $J_i \in \mathcal{Q}_i^{pred}(v,t)$ contains a number of cores that cannot be used when allocating cores for J_i. These are non-overlapping cores as they belong to different jobs. So, J_i cannot use any of the cores that are being used by $J_i \in \mathcal{Q}_i^{pred}(v,t)$. ■

Lemma 15. *J_i may be scheduled at time t at the earliest on p cores if it matches the following condition:*

$$q^{\min}(v,t) - q_i^{pred}(v,t) \geq p \tag{5.12}$$

Proof of Lemma 15. By combining the number of available cores at time t minus the cores that cannot be used by J_i at time t from Corollary 6 we can see how in order for J_i to be scheduled on p cores, the inequality must hold. ■

Corollary 1-bis. *A job J_i in a limited-preemptive execution model cannot start on exactly p cores before time $EST_i^p(v)$, defined as*

$$EST_i^p(v) = \max\{R_i^{\min}(v), t_{gang}^p(v), t_i^{pred}(v)\} \tag{5.13}$$

Proof of Corollary 1-bis. Similarly to the proof of Corollary 1, job J_i cannot start executing before being released and before all its predecessors have completed (i.e., before $R_i^{\min}(v)$). It also cannot start executing with exactly p cores before $t_{gang}^p(v)$ as already proved in Lemma 1. Finally, according to Lemma 15, the earliest time at which it can start has to be greater or equal than the time at which we know that dispatching J_i will mean that a waiting higher-priority job should have been dispatched instead as its predecessors just finished at the same time as J_i has been dispatched (i.e., $EST_i^p(v) \geq t_i^{pred}(v)$). Thus, J_i cannot start before $\max\{R_i^{\min}(v), t_{gang}^p(v), t_i^{pred}(v)\}$, thus proving the claim. ■

Lemma 2-bis. *J_i will not be the first job dispatched in the system state v or will not be dispatched on exactly p cores if it did not start to execute before time $t_{high}^p(v)$ defined as*

$$t_{high}^p(v) = \overset{\infty}{\min_{J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}}} \Big\{ \max \Big\{ t_h^p(J_i, J_j), t_{h,pred}(J_i, J_j) \Big\} \Big\} \tag{5.16}$$

where

$$t_h^p(J_i, J_j) = \begin{cases} r_j^{\max} & \textit{if } m_j^{\min} \leq p \\ \max\{r_j^{\max}, A_{m_j^{\min}}^{\max}\} & \textit{otherwise} \end{cases} \tag{5.17}$$

and

$$t_{h,pred}(J_i, J_j) = \overset{0}{\max}\{LFT_y^*(v) | J_y \in pred(J_j) \setminus pred(J_i)\} \tag{5.18}$$

Proof of Lemma 2-bis. We prove that a not-yet-scheduled higher-priority job J_j (i.e., $J_j \in \{\text{hp}_i \cap \{\mathcal{J} \setminus \mathcal{S}(v)\}\}$) will be dispatched before J_i if J_i did not start to execute before $t_{high}^p(v)$. We use the fact that $t_h^p(J_i, J_j)$ was already previously proven in Lemma 2 by showing how a lower priority job has to start before a higher priority job is released and has enough cores to become eligible.

$t_{h,pred}(J_i, J_j)$ shows the latest time at which a higher-priority job waiting for its predecessors can be ready. As J_j is waiting for its predecessors we have to look for the latest finish time of all of the predecessors of J_j (i.e. $\max^0\{LFT_y^*(v) | J_y \in pred(J_j)\}$). Moreover, since we are evaluating the time at which J_i is dispatched, the predecessors of J_i have finished their execution. Thus, we have to check the finish time of the predecessors of J_j that haven't finished their execution, these are $pred(J_j) \setminus pred(J_i)$. Thus we obtain $t_{h,pred}(J_i, J_j) = \max^0\{LFT_y^*(v) | J_y \in pred(J_j) \setminus pred(J_i)\}$

Then, if J_i has not started to execute before $t_{high}^p(v)$ this means that the predecessors of J_j have finished (i.e, $t_{high} \geq t_{h,pred}(J_i, J_j)$) and that J_j has enough cores and has been released (i.e., $t_{high}^p(v) \geq t_h^p(J_i, J_j)$). Combining both facts we obtain $\max\{t_h^p(J_i, J_j), t_{h,pred}(J_i, J_j)\}$ thus proving our claim. ■

Lemma 16. *The set of certainly running jobs $\mathcal{X}(v_k')$ of the new state v_k' is comprised of*

$$\mathcal{X}(v_k') = \Big\{ J_i \Big\} \cup \Big\{ J_j | J_j \in \mathcal{X}(v) \land EFT_j(v) > LST_i^p(v) \Big\} \tag{5.20}$$

Proof of Lemma 16. Since the system state v'_k results from dispatching J_i, job J_i is certainly running in system state v'_k. Hence $\mathcal{X}(v'_k) \supseteq \{J_i\}$.

Furthermore, all jobs that were certainly running just before dispatching J_i, i.e., those in $\mathcal{X}(v)$, and that complete their execution at the earliest after the time at which job J_i starts at the latest (i.e., after time $LST_i^p(v)$) are certainly running concurrently to J_i in every system state resulting from dispatching J_i. That is, $\mathcal{X}(v) \supseteq \{J_j | J_j \in \mathcal{X}(v) \wedge EFT_j(v) > LST_i^p(v)\}$. This proves our claim. ■