Delft University of Technology
Master of Science Thesis in Embedded Systems

Response-Time Analysis for Non-Preemptive Global Scheduling with Spin Locks

Suhail Taha Saeed Nogd
Supervised by: Dr. ir. Geoffrey Nelissen
Co-supervised by: Dr. Mitra Nasri

Embedded
Networked
Systems

TU Delft Delft University of Technology

Response-Time Analysis for Non-Preemptive Global Scheduling with Spin Locks

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

Suhail Taha Saeed Nogd
S.T.S.Nogd@student.tudelft.nl
suhailnogd@hotmail.de

25th of August 2020

Author
Suhail Taha Saeed Nogd (S.T.S.Nogd@student.tudelft.nl)
(suhailnogd@hotmail.de)
Title
Response-Time Analysis for Non-Preemptive Global Scheduling with Spin Locks
MSc Presentation Date
25th of August 2020

Graduation Committee

Prof. dr. K.G. Langendoen	Delft University of Technology
Dr. Mitra Nasri	Delft University of Technology & Eindhoven University of Technology
Dr. ir. Geoffrey Nelissen	Eindhoven University of Technology

A part of the work introduced in this master thesis has been presented at the workshop for *sCalable And PrecIse Timing AnaLysis for multicore platforms* (CAPITAL) in Brussels, Belgium in February 2020.
Furthermore, a paper has been written covering a part of the contributions presented in this thesis and has been submitted to the *Real-Time Systems Symposium* (RTSS), the premier conference in the field of real-time systems.
Finally, since the first paper did not comprise all the contributions presented in this thesis, it is planned to write a second paper based on the remaining contributions and submit it to the next possible conference or as a journal paper.

Abstract

With the proliferation of multicore platforms, the embedded systems world has shifted more and more towards multiprocessing to make use of high computing power and increased cyber functionalities. Although today multiprocessor platforms have been extensively adopted by real-time embedded systems, there exists a need for tools and techniques that can accurately assess the temporal correctness of a system. In terms of multiprocessor systems, this is coupled with fundamental challenges, since these systems, as we find them today, make use of complex hardware components, resource sharing and memory architectures, which negatively affect the timing predictability of such systems. Spin-based locking protocols, which are used to ensure mutual exclusion when sharing resources in a system, and a non-preemptive execution model have been found to help mitigate the adverse effect on the timing predictability, since they allow for less interruptions, which results in reduced cache evictions and a better estimate of worst-case execution times. While this improves the overall timing predictability of such systems, to date, there exists no response-time analysis that can analyze multiprocessor systems that globally execute non-preemptive tasks sharing resources protected by spin locks.

Motivated by the lack of analysis tools for systems that consider non-preemptive global scheduling and the access to shared resources, this work provides the first analysis for global job-level fixed-priority (JLFP) scheduling policies and FIFO- or priority-ordered spin locks. To do so, it extends the family of schedule-abstraction-based analysis to model the access to shared resources in a highly accurate manner. The proposed analysis computes response-time bounds for a set of resource-sharing jobs subject to release jitter and execution-time uncertainties by implicitly exploring all possible execution scenarios using state-abstraction and state-pruning techniques. A large-scale empirical evaluation of the proposed analysis shows it to be substantially less pessimistic than simple execution-time inflation methods (i.e., a straightforward extension of existing response-time analysis tools for non-preemptive tasks that *do not* share resources), thanks to the explicit modeling of contention for shared resources and a scenario-aware blocking analysis.

Preface

This thesis marks the finish line of my 2-year journey as an Embedded Systems master student at TU Delft. While it certainly was not an easy journey, I believe it was a time full of learning and developing. My student life at TU Delft has shown me the true meaning of Carpe Diem.

I would like to thank my supervisors, Dr. ir. Geoffrey Nelissen and Dr. Mitra Nasri, for their invaluable guidance and for letting me find solutions by myself, rather than providing them. The work with you allowed me to change my way of thinking about problems and find opportunities where I would normally see obstacles. During my work with you, I believe I have learned to think more critically and take a stance like a researcher. I would also like to thank Dr. Björn Brandenburg for providing open source tools and having an answer to all my questions.

My deepest gratitude goes towards my family. I do not know where I would be without your help and your advice. While physically I was in a different country, your phone calls always made me feel at home. Even during times of the lockdown, when the work environment was not ideal, you were always there to discuss and motivate me. Shukran!

Finally, to all my friends in Germany, I would like to say Dankeschön! for always listening to my complaints and supporting me during tough times.

Suhail Taha Saeed Nogd

Delft, The Netherlands
23rd August 2020

Contents

List of Figures

Abbreviations

BCRT best-case response time. 22

DAG directed acyclic graph. 14

DM deadline monotonic. 7

EDF earliest deadline first. 3, 6, 7, 11

FIFO first-in first-out. 2, 3, 8, 10–13, 15, 17, 26, 28, 30, 32, 33, 43, 53

FMLP Flexible Multiprocessor Locking Protocol. 11–13, 15

FP fixed-priority. 3, 7, 11

JLFP job-level fixed-priority. 3, 6, 7, 16, 21, 23, 24, 26, 28, 53

LP linear programming. 12, 13

MILP Mixed Integer Linear Programming. 12

MPCP Multiprocessor Priority Ceiling Protocol. 13

MSRP Multiprocessor Stack Resource Policy. 11

OMLP Optimal Locking Protocol. 12, 13, 15

Pfair optimal proportional fair. 10

PIP Priority Inheritance Protocol. 13, 15

RM rate monotonic. 7

SAG schedule-abstraction graph. 19, 36, 41

WCET worst-case execution time. 2, 14

WCRT worst-case response time. 2, 3, 22, 53

Chapter 1

Introduction

Over the past decade, multicore processing platforms have been extensively adopted by the real-time embedded systems industry in order to provide high computing power and deliver more *cyber* functionalities than ever before. These platforms allow a parallel execution of applications, which improves their response time while reducing the power consumption of the system. For example, in the automotive industry multicore processing platforms have been adopted for more than six years already [33].

For such multicore platforms, we use multiprocessor scheduling to coordinate how the workload is distributed over the available cores. We generally differentiate between three approaches namely *global scheduling*, *partitioned scheduling* and *hybrid scheduling*. In this work we focus on global scheduling, which is known to have a more efficient utilization of the platform resources [28]. Global scheduling is more flexible compared to the alternatives and can adapt better to dynamic changes (e.g. varying execution times), which are not rare in real applications found in the industry.

Generally, the industry deals with critical and complex applications that are divided into several components, which share data between each other. Hence, an essential capability provided by virtually all multitasking real-time operating systems (or runtime environments) is the ability to share data, software or hardware resources without compromising functional correctness (e.g., avoiding race conditions). In real-time systems, this is usually done by using locking protocols. Essentially, there are two types of locking protocols for this purpose: *suspension-based* and *spin-based* protocols. Using suspension-based locks, an executing system functionality (a.k.a *a task*) self-suspends when it is not granted access to the resource, meaning that it yields the processor making it available for other tasks to execute on. Spin-based locks, on the other hand, allow the task to spin (busy-wait) on the core and to keep it occupied until the task is granted the access to the resource. Each approach offers different trade-offs. Conceptually, suspension can be more efficient, since tasks yield the processor, instead of busy-waiting (spinning), such that the core can be utilized for useful computation of another task. However, suspension comes with increased analysis complexity, for example, it is notoriously difficult to predict cache contents after suspensions and usually it is pessimistically assumed that any suspension results in a complete loss of cache affinity. On the other hand, spinning is relatively simple to implement and analyze, requires virtually

no OS support and incurs significantly lower runtime overheads (compared to suspension-based locking protocols) [11]. This means that if the code section of a task that requires to access a shared resource (a.k.a the *critical section*) is relatively short, the cost of suspending and resuming the task can easily outweigh the cost of busy-waiting. This makes spinning more efficient and an attractive choice when critical sections are short, which they ideally are in well designed multiprocessor systems, in order to prevent significant delays when tasks share resources between each other. Throughout this thesis, we will focus on systems in which spin locks are used to enforce mutual exclusion.

Common multiprocessor systems, as we find them today, use complex interconnected hardware and multi-level caches to ensure a high performance [4]. However, this has been proven to negatively affect the predictability of the system, due to an increased interference coming from the access to shared hardware resources. An effective approach to make the hardware platform more predictable is by utilizing a *non-preemptive* execution model. A non-preemptive execution model forbids that any instance of a task (a.k.a *a job*) is preempted (interrupted), which eliminates job migration overheads and avoids that cache contents regarding an unfinished job are lost due to the execution of a different job on the core (intra-core cache interference). Essentially, this allows for a more accurate estimation of the worst-case execution time (WCET), which improves the overall timing predictability of the application.

Timing predictability is a significant requirement for real-time safety-critical applications, i.e., in such systems, the worst-case response time of the task must be bounded despite uncertainties that may happen in the system at runtime. To ensure timing predictability, a real-time system typically undergoes a *response-time analysis*, whose goal is to determine whether or not the worst-case response time (WCRT) of a workload complies with its timing constraints.

While the literature provides ample analyses of both, suspension- and spin-based approaches for globally scheduled *preemptive* tasks (e.g., see a recent survey [11]), to the best of our knowledge there exists no *response-time analysis* for globally scheduled *non-preemptive* tasks that share resources protected by spin locks.

1.1 Research Question

Despite favorable timing analysis properties that come with a non-preemptive execution model and spin-based locking protocols (i.e., less interruptions, less cache evictions, better worst-case execution time estimates), to date, no analysis has been proposed in the literature that addresses the problem of analyzing workloads that **share resources** protected by **spin locks** and that are scheduled on a **multicore** platform using a **global non-preemptive** scheduler.

Motivated by this lack of analysis tools for such systems, our work deals with the following key research question:

How to find highly accurate upper (and lower) bounds on the worst-case (and best-case) response time of jobs scheduled by a global non-preemptive scheduling policy upon multiprocessor platforms where the access to shared resources is managed by first-in first-out (FIFO)- or priority-ordered spin locks?

1.2 Solution Approach and Contributions

In this work, we propose a new sufficient response-time analysis for periodic tasks and other workloads with a repeating pattern of job releases (such as periodic, burtsy or multi-frame tasks) based on the notion of schedule abstraction, a concept initially introduced by Nasri et al. [31–33]. However, whereas Nasri et al.'s prior analyses do not consider locking-induced delays, we assume—and model in detail—that tasks coordinate mutually exclusive access to shared resources by means of *FIFO- or priority-ordered spin locks*. Our analysis is generic in nature and covers all work-conserving (i.e., no core is left idle if there is a pending job that is waiting to be executed) global non-preemptive *job-level fixed-priority (JLFP)* scheduling policies. JLFP policies are a wide range of scheduling policies, in which each job has a fixed priority and the scheduler always executes the highest priority jobs first. They include well-known policies such as *earliest deadline first (EDF)* and *fixed-priority (FP)* scheduling.

Our proposed analysis implicitly explores all possible orders of job start times as well as their accesses to shared resources in a *schedule-abstraction graph* [31–33]. The efficiency (in terms of runtime and memory footprint) and accuracy of this exploration depends on the level of abstraction used to encode the system states and the input tasks. Hence, to design an efficient and scalable analysis, we propose a completely *new system-state abstraction* that models shared resource accesses by means of FIFO- or priority-ordered spin locks in a highly accurate manner. This new system state representation requires to design (and a proof of soundness of) a whole new set of expansion and merging rules, which are used to build the schedule-abstraction graph. Ultimately, our analysis yields a safe *worst-case response time* bound for each job that reflects the worst-case blocking possible.

The proposed analysis has shown significant accuracy gains compared to the state of the art and has been competitive with (or even superior to) solutions designed for preemptive systems (see Section 6.3). Furthermore, our work has highlighted the pessimism that we find in currently existing blocking analyses and has shown that using a schedule-abstraction graph is a promising approach to mitigate this pessimism.

Finally, we present two extensions for our work. The first covers partial order reduction techniques to smartly prune the schedule-abstraction graph in order to delay a state space explosion and improve the scalability of the analysis. Lastly, the analysis has been extended by making it compatible with multi-unit resources and k-exclusion locking protocols (see Section 7.2), which is motivated by the fact that in real-time systems it is common to find multiple similar units of a resource which are handled by k-exclusion locking protocols.

1.3 Summary of Achievements

The key idea of a schedule-abstraction-based response-time analysis considering the access to shared resources protected by FIFO-ordered spin locks, as introduced in this master thesis, has been presented at the workshop for *sCalable And PrecIse Timing AnaLysis for multicore platforms* (CAPITAL) in Brussels, Belgium in February 2020.

Furthermore, a paper has been written covering the response-time analysis

considering FIFO-ordered spin locks and has been submitted to the *Real-Time Systems Symposium* (RTSS), the premier conference in the field of real-time systems. Since this paper did not comprise all the contributions presented in this thesis, it is planned to write a second paper based on the remaining contributions, namely the response-time analysis considering priority-ordered spin locks as well as the extensions considering k-exclusion protocols and partial order reduction techniques. It is planned to submit these to the next possible conference or as a journal paper.

1.4 Thesis Organization

In Chapter 2, we introduce the necessary background and concepts needed to be able to follow this research and the design of the analysis. Chapter 3 provides an overview of the closest related work to our research topic. In Chapter 4, the system model and the problem are defined and all underlying assumptions are stated. Subsequently, Chapter 5 proposes our schedulability analysis. Here, we introduce all the notions and definitions related to the schedule-abstraction graph. Furthermore, the generation of the graph is explained by describing sound expansion and merge rules in detail. Towards the end of Chapter 5, we prove the correctness of the analysis. In Chapter 6 we showcase an empirical evaluation of the work. We extend our work in Chapter 7 by considering multi-unit resources and we introduce partial order reduction techniques to reduce the number of system states that are explored during the analysis. Finally, we draw our conclusion and provide an insight into the future work in Chapter 8.

Chapter 2

Background

This chapter provides a description of the general concepts that will be used throughout this thesis. However, it does not define the problem, assumptions, or the system model. These will be discussed in Chapter 4.

2.1 Definitions

In this work we consider real-time systems. A real-time system is a system whose correctness depends not only on the correctness of logical results, but also on the time at which these results are produced [14]. We find such systems as part of many applications for example in fields like the automotive, industrial automation, medical or aircraft industry. In the following, we will introduce some of the common terms and explain the concepts behind them.

2.1.1 Tasks and Jobs

In real-time systems, a *task* is defined as a process that represents one of the system functionalities. For a system to operate correctly such tasks need to be processed by considering that no timing requirements are violated. To know if this condition is met, each instance of a task (a.k.a a *job*) has a number of properties. Each job has a *release time*, that defines the time at which it becomes available for execution in the system. Furthermore, a job has a computation load, called *execution time*, which tells how much time is needed to process it. Another property is the *deadline*, which specifies the time by which the processing of the job should be finished in order to ensure temporal correctness. A job can also have additional properties for example a *priority* that decides how important it is. Finally, real world systems usually deal with a collection of many tasks and jobs, which is called *task set* [14].

The concept of a task can be visualized in the example shown in Fig. 2.1. In the example, we see a task, called Task 1, that releases three jobs. A new job of the task is released regularly every 4 time units, which means that the *period* of the task is 4. Furthermore, we see that the deadlines of any job aligns with the release of the next job. We call these implicit deadlines. Every job needs to finish its workload before the next instance of the task is released. In this example every job has an execution time of 2 time units.

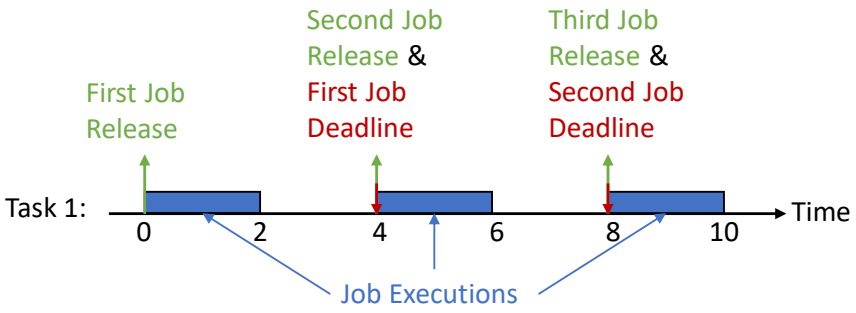


Figure 2.1: **Example of a task.**

2.1.2 Scheduler and Execution Model

In real-time systems, the entity that decides when and where a job can start executing or needs to be interrupted is called the *scheduler*. Schedulers make these decisions by following scheduling policies. These policies decide which of the released jobs is chosen to execute next considering the underlying execution model. In terms of execution models, we mainly differentiate between a *preemptive*, a *non-preemptive* and a *limited-preemptive* model. A preemptive execution model allows the scheduler to interrupt a running process on a core and schedule a different job while the current job has not completed its execution yet. If we consider a non-preemptive execution, a job that has started to execute on a core will certainly run to completion without being interrupted by the scheduler. Finally, there exists the limited-preemptive model, which, as the name implies, allows preemptions by the scheduler only at (usually predetermined) specific points in the execution of a job or when the job has executed for at least a minimum amount of time [14].

2.2 Global Multiprocessor Scheduling

In global scheduling, we consider one ready queue which accommodates all jobs of every task that have arrived to the system and that can start to execute (ready jobs). Jobs are scheduled according to a global scheduler during runtime, which considers all available cores. Global scheduling allows for migration between the cores, meaning that tasks are not limited to execute on preassigned cores, but can use the full set of available processors to execute their workload. Hence, there is no task-to-core binding (usually known as core affinity) and jobs that have started running on a specific processor can even finish their execution on a different core if the system is preemptive. A global scheduler for multiprocessors picks the m-highest priority jobs and schedules them on the m available cores [2]. Then, each dispatched job starts to execute on the core it was dispatched to. The concept of global scheduling can be visualised in Fig. 2.2.

2.2.1 Global Job-Level Fixed-Priority Scheduling

Job-level fixed-priority scheduling covers a wide range of scheduling policies in which each job has a fixed priority. This includes policies such as *earliest dead-*

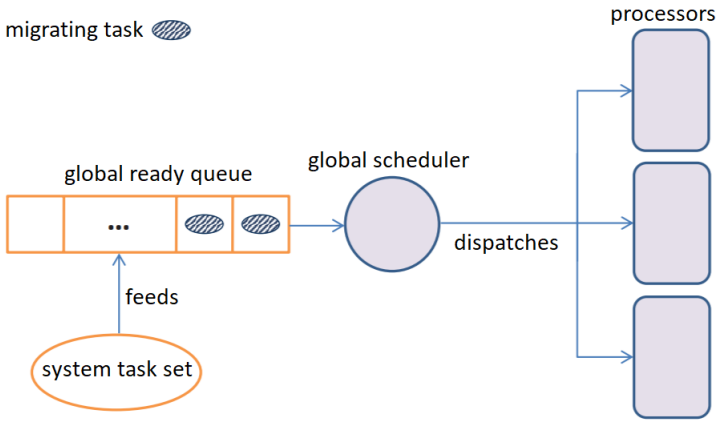


Figure 2.2: **Example of global scheduling.**
[2]

line first (EDF) and *fixed-priority (FP)* scheduling, which are among the most widely implemented policies in real-time operating systems. At any decision point, a global JLFP scheduler picks the m-highest priority jobs and dispatches them to execute next on the m-available processors.

In fixed-priority scheduling, priorities are usually assigned following a priority assignment method. Common methods are *rate monotonic (RM)* and *deadline monotonic (DM)*. In rate monotonic, priorities are assigned according to the period of the task, where a smaller period equates to a higher priority.

On the other hand, deadline monotonic assigns priorities according to *relative* deadlines, where an earlier deadline leads to a higher priority. If we deal with implicit deadlines, meaning that the relative deadline is equal to the period, RM and DM fixed-priority scheduling produce the same schedule.

While fixed-priority scheduling is a task-level fixed-priority scheduler, earliest deadline first is a task-level dynamic-priority scheduling policy. This means that different jobs of the same task can have different priorities. EDF scheduling assigns higher priorities to the jobs with an earlier *absolute* deadline.

As discussed next, one can use schedulability analyses to check whether a system respects its timing constraints under a given scheduling policy.

2.3 Schedulability Analysis

A schedulability analysis for real-time systems is an analysis that checks whether a given workload respects its timing constraints under any valid execution scenario that may happen in the system at runtime. If the analysis finds that the timing constraints are met, we say the workload is schedulable. If we predict that a job deadline may be missed, the workload is deemed non-schedulable.

Schedulability analyses are of great importance as they help us predict the worst-case timing behavior of a system, which allows us to confirm the temporal requirements of an application. As real-time systems handle critical and sometimes even vital tasks, it is of utmost importance to be able to understand and

predict the behavior of the application and to ensure a correct working. Therefore, a schedulability analysis is conducted at design time (offline), because it is typically required by the certification authorities in the process of certifying the system.

Generally we can categorize any schedulability analysis into one of the following three types: exact, necessary or sufficient analysis.

If an exact schedulability analysis accepts the task set, the task set is certainly schedulable by the underlying policy. If a task set is rejected by an exact test, the task set is certainly not schedulable by the given algorithm. Therefore, an exact analysis can explicitly identify if a workload fulfills its timing requirements or not.

On the other hand, if a necessary test accepts a task set, the workload is not necessarily schedulable under the underlying policy. However, if a task set does not pass a necessary test, we are certain that it is not schedulable. As the name implies, the necessary test does not provide us with information about schedulable workloads, because it does not safely identify them. It is essentially used to detect which task sets are not schedulable.

Finally, we have sufficient analyses. Task sets that pass a sufficient test are certainly schedulable under the given scheduling policy. However, if a workload does not pass the analysis it is not necessarily unschedulable. Therefore, sufficient analyses do not provide us with information about certainly non-schedulable task sets, but are used to safely identify workloads that meet their timing requirements.

Scheduling workloads can be affected by many factors, which need to be accounted for by an accurate schedulability analysis. An example of such a factor is the access to shared resources, which can introduce additional delays that might lead to timing requirements not being met. In the following we will dive deeper into shared resources and related design choices that can affect the schedulability of a workload.

2.4 Shared Resources

A *shared resource* is a resource that is used by two or more tasks. Such a resource can be software (e.g., data structure or a set of global variables) but it can also be hardware (e.g., a timer or a display unit). Resources that do not allow for concurrent accesses need to be protected to ensure mutual exclusion and a correct working of the application. Shared resources are protected using so-called locks, which means that a job has to acquire a lock first, in order to use the resource. The workload of a job that is executed under mutual exclusion when utilizing a shared resource is called *critical section*. The use of shared resources in a system comes with several design choices. These choices will be discussed further in the following.

First, a serialization order has to be specified, meaning that rules have to be defined that determine, which job is allowed to access the resource next in case two or more requests are simultaneously issued (conflicting requests). The most common design choices are a *FIFO queue* and a *priority queue*. While a FIFO queue focuses on fairness (i.e., the access of lower-priority jobs is not continuously delayed with the arrival of higher-priority jobs), a priority queue allows control over the amount of blocking experienced by higher-priority jobs

because of lower-priority ones [11].

Another important design choice is how to handle the waiting of a job in case of contention. Here we differentiate between two techniques: *spinning* and *suspending*. Spinning, also known as busy-waiting, describes the process that if a job is blocked due to a resource being used, the job continues to occupy its assigned core and waits until it is granted the access to the resource. Suspending on the other hand, removes the blocked job from the core making the processor available to other jobs. The blocked job has to wait until it is granted the access to the resource. While suspension seems more efficient because it allows cores to execute relevant workload instead of busy-waiting on a resource, it can be less efficient than spinning if the cost of suspending and resuming a job outweighs the cost of busy-waiting, which occurs when the critical section is short. Furthermore, suspension requires more OS support and is relatively harder to implement and analyze [11].

Another major design choice is the progress mechanism, which describes the way a lock-holding job executes its critical section. If a lock-holding job is allowed to be preempted by higher-priority jobs (*preemptable locks*), we deal with the (potential) problem of unbounded delays, which must be avoided in real-time systems. Therefore, there are so called progress mechanisms that (partially) force the execution of a lock-holding job. For example, this can be achieved by boosting a job's priority when it enters its critical section or by making the execution non-preemptive altogether [11].

Support for nesting is another design choice when working with shared resources on multiprocessors. It describes whether the system allows a job to acquire multiple locks in a nested manner. This can be a considerable source of increased complexity from both an implementation and analysis point of view. Specifically, there is the risk of running into a deadlock situation, hence it becomes extremely difficult to accurately analyze such systems efficiently [11].

The final design choice that we discuss related to shared resources is where a job executes its critical section once it has acquired the protecting lock. Here we usually differentiate between an *in place* and a *centralized critical section* execution. An in place execution describes the principle that a job executes its critical section as part of its normal execution on the assigned core. A centralized critical section execution describes the method that all critical sections related to a particular resource are executed on an a priori chosen synchronization processor. While an in place execution is the natural choice for spin locks, a centralized critical section execution can provide analytical benefits when dealing with suspension-based locks [11].

Chapter 3

Related Work

In the following, we divide the first half of the related work into two broad categories, namely spinning-based analyses and suspension-based analyses. In the second half, we present the state of the art regarding schedule-abstraction-based analyses, since our work extends specifically this family of analyses to accurately model the access to shared resources.

3.1 Spinning-Based Analyses

Spin locks distinguish themselves from suspension-based locks by allowing jobs to spin on a core while waiting on a resource instead of voluntarily yielding it. This means that there are no additional context switches when using spin-based locks.

The following gives an overview of spinning-based analyses that can be found in the literature for both global and partitioned scheduling [11].

3.1.1 Global Scheduling

In this section, we look at works that have considered spin locks under global scheduling. In [24–26], Holman and Anderson present one of the early works on spin-based real-time locking protocols under global scheduling. They introduce two protocols (*skip* protocol and *rollback* protocol) that focus on FIFO spin locks in systems using optimal proportional fair (Pfair) schedulers, which is a quantum-based scheduler. This means that there exists a system quantum and at each multiple of this quantum, tasks are regularly rescheduled. In their study, Holman and Anderson introduce and address the important distinction between *long* and *short* shared resources. A shared resource is considered to be short, if all critical sections related to the resource are "relatively" short. A resource is considered long if there exists a critical section that is "relatively" long. The exact threshold that determines if a resource is considered short or long is application- and system-specific. Holman and Anderson's works distinguish long and short resources by relating the critical sections to the length of the system quantum. Finally, for both their protocols, they have designed blocking analyses. However, nowadays Pfair scheduling is not widely found in practice, as it induces a lot of overhead (due to frequent synchronization that leads to

many context switches) that result in a relatively poor performance compared to other algorithms.

Later, Devi et al. [18] and Chang et al. [15] have studied spin locks under more common scheduling policies, namely global preemptive EDF and global preemptive FP considering FIFO-ordered spin locks.

Finally in [7], Block et al. have introduced the Flexible Multiprocessor Locking Protocol (FMLP), which is a family of protocols that covers different schedulers and critical section lengths. Similar to Holman and Anderson, FMLP adopts a distinction between short and long shared resources. In addition, it is compatible with both global and partitioned scheduling and for each of the combinations, there exists an analysis that considers a specific variant of the protocol. For long resources, FMLP adopts semaphores which will be discussed in Section 3.2. In terms of global scheduling and short resources, FMLP relies on Devi et al.'s analysis [18]. However, a more accurate analysis considering FMLP with short resources has been provided by Brandenburg's holistic analysis [8], which considers FIFO spin locks. In fact, this analysis is compatible with global and partitioned scheduling and relies on execution-time inflation. As the name implies, it "inflates" a tasks execution time, meaning a task's worst case execution time becomes the sum of its original worst case execution time and the maximum blocking time on the resource due to other task(s) that request(s) the same resource. The analysis has been found to provide an improved performance compared to prior analyses of spin locks [8].

Summing up, while prior works have studied the problem of analyzing the schedulability of systems using spin locks under global scheduling, their focus, so far, has been on preemptive scheduling. The most interesting analysis (in terms of performance) provided in this section is Brandeburg's holistic analysis of FMLP with short resources and will therefore be considered further in our empirical evaluation in Chapter 6.

3.1.2 Partitioned Scheduling

An early blocking analysis has been presented by Gai et al. [22], which uses the, now classic, Multiprocessor Stack Resource Policy (MSRP). MSRP is a policy for partitioned scheduling, which is compatible with both fixed priority and earliest deadline first scheduling. It differentiates between local and global resources, where local resources are resources that are only accessed by tasks on a specific core and global resources can be accessed by any task in the system. These global resources are protected by non-preemptive FIFO spin locks and in fact, Gai et al.'s work was the first to leverage non-preemptive FIFO spin locks by providing a sound protocol and introduce a corresponding schedulability analysis [11]. The analysis is also compatible with both fixed priority and earliest deadline first schedulers and relies on execution-time inflation (as discussed in Section 3.1.1). While this analysis is simple in nature, it also introduces pessimism. A major source of this pessimism is that the analysis considers that every critical section that contributes to the blocking has the length of the longest critical section regarding the particular resource. Clearly, this is pessimistic for shared resources that allow multiple operations with various costs and therefore this analysis would overestimate the final blocking bound.

To reduce this pessimism Brandenburg [8] designed a holistic blocking analysis (as discussed in Section 3.1.1) for MSRP by deriving per-task blocking

bounds, which prevent considering the same long critical sections multiple times. However, the analysis still overestimates bounds when multiple tasks that are assigned to the same core request the same global resource. As the analysis always considers the maximum blocking that can occur on a core, the longest critical section will be considered in multiple inflated worst case execution times, which is not possible and therefore introduces pessimism.

To counter the pessimism found in inflation-based solutions, Wieder and Brandenburg [39] introduced a Mixed Integer Linear Programming (MILP)-based analysis for spin locks under partitioned fixed priority scheduling. This work models direct blocking by solving an MILP optimization problem. Later, the work has been extended by Biondi and Brandenburg [5] to support partitioned earliest deadline first scheduling.

In summary, the scientific literature has studied the schedulability of systems that use spin locks. However, these prior work focus on partitioned scheduling and mostly consider preemptive systems.

3.2 Suspension-Based Analyses

In the following, we discuss existing suspension-based analyses for both global and partitioned scheduling [11]. For each of these sections we differentiate between suspension-oblivious and suspension-aware analyses. Suspension-oblivious analyses are relatively simpler and model self-suspension as inflated execution times (processor demand). Suspension-aware solutions, on the other hand, aim to explicitly model self-suspension (yielding the processor) to bound potential blocking as tight as possible and to reduce pessimism. As before we look at analyses designed for global and partitioned scheduling.

3.2.1 Global Scheduling

For long resources under global scheduling, Block et al. [7] provide one of the earliest analyses that uses a variant of the FMLP and a simple FIFO queue to investigate systems using semaphores in a suspension-oblivious fashion. The analysis aims for simplicity and therefore does not take specific shared resource request patterns into account. Furthermore, there is another family of protocols which is compatible with global and partitioned scheduling, namely the Optimal Locking Protocol (OMLP) [13]. In contrast to FMLP, OMLP ensures a tighter bound on the blocking by using a hybrid queue instead of FMLP's simple FIFO queue. The hybrid queue consists of a bounded FIFO segment of length m (number of cores in the system) and a priority-ordered tail which leads into the FIFO segment. For global OMLP, a suspension-oblivious analysis [12] that considers execution-time inflation has been introduced to accurately model and bound blocking.

In terms of suspension-aware analyses, Yang et al. [41] have presented an analysis under global fixed priority scheduling, which makes use of Block et al's FMLP [7] variant for long resources. The analysis finds a safe response-time bound by making use of a linear programming (LP) approach. Essentially it solves a linear optimization problem to rule out impossible scenarios. To identify these impossible scenarios, a number of constraints are applied that are invariant to the underlying locking protocol and a global fixed priority scheduler.

In the same paper, they have also presented an analysis (again based on linear programming) considering Easwaran and Andersson's Priority Inheritance Protocol (PIP) [38]. The priority inheritance protocol boosts the priority of a lock-holding task by considering the maximum of its own base priority and any priority of a task that is waiting to acquire the lock. Yang et al.'s survey [41] has shown that the long FMLP and the PIP analyses are incomparable in terms of performance and preference should be given based on the workload. When the workload requires fairness, FMLP's FIFO queue would perform slightly better than PIP. On the other hand, if we deal with real-time workloads that need prioritization of urgent jobs, PIP's priority-ordered wait queues would be of more advantage. Finally, yang et al.'s analyses of the long FMLP and the PIP also introduce a source of pessimism. The analyses assume discrete time and while this is a safe relaxation (safe bounds are estimated), the actual response-time bound can be over-approximated [41].

Summing up, the scientific literature has studied the analysis of workloads that share resources under global scheduling. However, these works focus on suspension-based locks and a preemptive execution model. The most interesting work (in terms of performance) is given by the inflation based analysis for global OMLP and yang et al.'s LP-based analysis for long FMLP and PIP. Hence, we will consider these analyses in our empirical evaluation in Chapter 6.

3.2.2 Partitioned Scheduling

To date, various prior work for multiprocessor synchronization is found considering real-time suspension-based protocols for partitioned scheduling. Block et al's FMLP [7] for long resources is one of the early protocols that also supported the analysis of systems using semaphores in a suspension-oblivious fashion. Furthermore, a variant of the OMLP, called partitioned OMLP, has been presented in [12], which was able to restrict the maximum delay more accurately compared to the previous FMLP. Fined-grained analyses of the partitioned OMLP are available in [8,13].

In terms of suspension-aware analyses, early work includes Rajkumar's Multiprocessor Priority Ceiling Protocol (MPCP) [36,37] for partitioned fixed priority scheduling. While the classic MPCP uses a priority queue, Lortz and Shin have found in their work [29] that by using a FIFO queue instead, a significant improvement in schedulability can be achieved. Considering the various flavors of MPCP, several blocking analyses have been introduced in [9,27,35,37]. To date, the most accurate analysis of the MPCP is given in [9] which uses a linear programming approach to accurately model critical sections. Block et al.'s family of FMLP also includes a variant for partitioned scheduling [7]. Later, it has been refined and a partitioned FMLP+ has been introduced in [8]. In comparison to prior semaphore protocols, partitioned FMLP+ allows for a tighter and more accurate bound on the blocking. Several blocking analyses considering FMLP+ have been presented in [8–10] where the LP-based approach [9] provided the most promising results. The LP-based analysis has been extended by Ma et al. and presented in [30] to increase the accuracy even further.

Finally, the literature studies schedulability analyses considering shared resources, but the focus lies on partitioned scheduling and suspension-based locks.

3.3 Schedule-Abstraction-Based Analysis

Generally, in terms of schedulability analyses, there are essentially three frameworks that have been introduced until today. The first family covers closed-form analyses which are classic sufficient analyses that are based on the standard response-time analysis paradigm. Although these tests are usually fast, they introduce a lot of pessimism and are hard to extend.

The second family covers exact solutions that are based on model checkers or constrained programming such as Uppaal. Although these tests are highly accurate and are in fact easy to extend, they severly fail in terms of scalability [40].

The lack of scalability and the amount of pessimism that has been introduced by these previous frameworks, question the utility of such tests in industrial settings. Hence, another family has been introduced which represents the third framework. It covers response-time analyses that explore all possible schedules by utilizing a schedule-abstraction graph [31–33]. To date, three main works have been published on schedule-abstraction-based analyses. The first one [31] is an exact analysis for uniprocessor systems. It considers independent non-preemptive jobs that are scheduled under a work-conserving or non-work-conserving job-level fixed-priority scheduling policy. Nasri et al.'s second work on schedule-abstraction-based analyses is an extension to the previous work and designed for multiprocessor systems. Unlike the previous exact analysis [31], this work [32] provides a sufficient analysis, due to the increased complexity that comes with investigating multiprocessor systems. Furthermore, it considers independent non-preemptive jobs and a global work-conserving job-level fixed-priority scheduler. Finally, the work from [32] has been extended to also account for directed acyclic graph (DAG) tasks with precedence constraints and has been presented in [33]. Interestingly, while [33] resembles [32] in terms of system assumptions, Nasri et al. have proposed a significantly different approach. In [33] a novel system state representation has been introduced, which reduces the number of states, delays a state-space explosion and essentially improves scalability.

In terms of analyzing multiprocessor systems, the work presnted in [33] performs very well and provides a reasonable trade-off solution in terms of run time and accuracy. However, it is based on the assumption that any delays regarding shared resources are accounted by the WCET provided to the system, which simplifies the problem. Therefore, the analysis ignores the existing timing problem related to shared resource accesses, making the solution less representative of real systems we find in the industry.

3.4 Summary

In summary, currently we are not aware of any prior response-time analysis with a comparable level of accuracy in the accounting of synchronization delays for systems that consider global non-preemptive scheduling and shared resources protected by spin locks. In fact, somewhat surprisingly, we found that there is hardly any directly related prior work according to a recent survey of Brandenburg on multiprocessor real-time locking protocols and related analyses [11]. While spin locks have long been recognized to have favorable properties for

worst-case timing analysis [3,5–8,15,16,18,22,24,39] (e.g. protecting the state of core and cache and incuring relatively low runtime overhead), prior work in this space has focused on either *preemptive* global scheduling [3,7,8,15,18,24] or partitioned scheduling [5–8,22,39].

In the context of preemptive global scheduling, the most relevant locking protocols and blocking analyses are Brandenburg's *holistic blocking analysis* of non-preemptive FIFO spin locks [8] as used in Block et al.'s FMLP for *short* critical sections [7] as well as the *suspension-based* global OMLP [12] and Yang et al.'s LP-based analysis of FMLP for *long* critical sections [7,41], and PIP [19, 38,41]. We compare against each of these in our evaluation (Chapter 6).

For a review of real-time locking in general, we refer the reader to Brandenburg's comprehensive survey of the area [11].

Chapter 4

System Model and Problem Definition

We consider a work-conserving global JLFP scheduling policy to schedule a set of *non-preemptive* tasks on a multicore platform with m identical cores. Each task releases jobs according to a specific pattern (e.g., periodic, rate-based or bursty), and each job has a fixed priority. The JLFP scheduler dispatches ready jobs in order of decreasing priority.

4.1 Workload Model

A job $J_i = ([r_i^{min}, r_i^{max}], d_i, p_i, \langle J_{i,1}, \ldots, J_{i,n_i} \rangle)$ is defined by an earliest release time r_i^{min}, a latest release time r_i^{max}, an absolute deadline d_i, and a priority p_i. We assume that timing parameters are discrete, i.e., integer multiples of a clock tick.

A job's execution is modeled by its sequence of *job segments* $\langle J_{i,1}, \ldots, J_{i,n_i} \rangle$, where $J_{i,1}$ and J_{i,n_i} are its first and last segment, respectively. Each job segment $J_{i,j}$ is identified by $J_{i,j} = ([C_{i,j}^{min}, C_{i,j}^{max}], \eta_{i,j}, [L_{i,j}^{min}, L_{i,j}^{max}])$, where $C_{i,j}^{min}$ is the best-case execution time (BCET), $C_{i,j}^{max}$ is the worst-case execution time (WCET), $\eta_{i,j}$ is the set of resources that the segment *requests*, and $L_{i,j}^{min}$ and $L_{i,j}^{max}$ are the minimum and maximum length of the critical section associated with $\eta_{i,j}$, respectively. We obviously must have that $L_{i,j}^{min} \leq C_{i,j}^{min}$ and $L_{i,j}^{max} \leq C_{i,j}^{max}$ since the critical section length is part of the execution time of the segment. In this work, we assume that each segment $J_{i,j}$ can access at most one resource protected by a lock (i.e., $|\eta_{i,j}|$ is equal to 0 or 1). Without any loss of generality, we further assume that the critical section of a segment is located at the beginning of the segment. That is, the segment must first be granted the lock protecting the shared resource in $\eta_{i,j}$ (if any) to start executing.

All segments of a job inherit the job priority. For ease of notation, we use $hp(J_{i,j})$ to refer to the set of segments with a higher priority than $J_{i,j}$. We assume that ties in priority are broken arbitrarily but consistently.

Each job must execute sequentially (on one core) with run-to-completion semantics, but may compete for shared resources with other jobs running in parallel on other cores. We do not consider inter-job precedence constraints in this paper.

4.2 Shared Resources

We let $\mathcal{L}$ denote the set of shared resources protected by locks. A shared resource $\ell_x \in \mathcal{L}$ is *available* if and only if no job is currently accessing it. When a job segment $J_{i,j}$ is given access to a resource, we say that the resource is *granted* to $J_{i,j}$. A segment $J_{i,j}$ cannot start executing until the resource $\ell_x \in \eta_{i,j}$ (if any) is granted to $J_{i,j}$. Resource ℓ_x will be released as soon as the segment's critical section completes (i.e., within $[L_{i,j}^{min}, L_{i,j}^{max}]$ time units after the resource was granted). If a segment $J_{i,j}$ requests access to a resource ℓ_x that is already granted to another job, we say that J_i is *blocked* on ℓ_x. The job J_i *busy-waits* (i.e., *spins*) until it is granted ℓ_x.

In this work, we cover both FIFO- and priority-ordered spin locks, i.e., resources are either granted in the order in which requests arrive or they are granted to the job with the highest priority. Under FIFO-ordered spin locks, if two jobs request the same resource at the exact same time, we assume that the tie is broken arbitrarily but consistently (e.g., in our experiments, ties are broken in favor of lower job IDs). Same holds if two or more jobs have the same priority under priority-ordered spin locks.

4.3 Execution Model

A job is *ready* at time t if it has been released and has not yet started executing. Due to the non-preemptive execution model, a job must run to completion without preemption once it has started execution. Thus, once the first segment $J_{i,1}$ of a job J_i begins its execution on a core, the core is exclusively serving J_i until J_i completes its execution, including during any blocking delays. We say that a job J_i has *claimed* a core if it started executing and has not finished yet. Upon completion we say that J_i *releases* the core, making it again available to other jobs. Hence, a core is either *claimed* (busy executing a job) or *free* for a new job to start execution.

Given the *finish time* f_i and the release time r_i of J_i, we compute the *response time* of the job as $f_i - r_i$.

4.4 Problem Definition

We seek to bound the worst-case response time of each member of a finite set of non-preemptive jobs $\mathcal{J}$ that can access shared resources. The job set $\mathcal{J}$ is the set of all jobs released in an *a priori* computed *observation window*. The input job set in the observation window must be a representative workload for the system, namely, either because the observation window is long enough to capture the whole workload (e.g., in batch scheduling), or because the release patterns of the jobs in the observation window repeats for the rest of the lifetime of the system so that it is sufficient to only analyze one instance. For example, the observation window of synchronous periodic task sets with implicit deadlines that exhibit no deadline misses is one *hyperperiod* (i.e., the least integer multiple of the periods); further safe observation windows for various types of periodic (or multi-frame) task models can be found in [23,31,34].

Our analysis deems a job set $\mathcal{J}$ *schedulable* only if no execution scenario of $\mathcal{J}$ (Definition 1 below) leads to some job exhibiting a response time exceeding

its deadline.

Definition 1. *An execution scenario γ is a mapping of jobs to release times, segment execution times, and critical-section lengths such that $\forall J_i, \in \mathcal{J}$, $r_i \in [r_i^{min}, r_i^{max}]$ and $\forall j, 1 \leq j \leq n_i$, $C_{i,j} \in [C_{i,j}^{min}, C_{i,j}^{max}]$ and $L_{i,j} \in [L_{i,j}^{min}, L_{i,j}^{max}]$.*

Chapter 5

Schedulability Analysis

The proposed schedulability analysis builds a *schedule-abstraction graph (SAG)* [31] to implicitly explore all possible execution scenarios of a job set. In this section, we define our notion of a *resource-aware* SAG and explain its construction and use in bounding each job's worst-case response time.

5.1 Schedule-Abstraction Graph

A SAG is a directed acyclic graph $G = (V, E)$, where V indicates the set of (abstract) states reachable by the system and E represents the set of scheduling decisions leading from one (abstract) system state to another. An edge $e = (v_p, v_q, J_{i,j})$ from v_p to v_q with label $J_{i,j}$ indicates that executing $J_{i,j}$ in state v_p evolves the system to state v_q. We say that a job segment $J_{i,j}$ starts executing *next* in state v_p (or *succeeds* v_p) if there exists an outgoing edge from v_p with label $J_{i,j}$.

By convention [31], state v_1 represents the initial state of the system at time zero, where every core is idle and no segment has started executing yet. A path P from v_1 to a state v_p represents a possible job-segment start order that results in system state v_p. The length of such a path P indicates the number of segments that have started (and potentially already finished) their execution when the system reaches state v_p, i.e., $|P| \triangleq |\mathcal{J}^P|$, where $\mathcal{J}^P$ denotes the set of labels on the edges of path P. If a vertex v_p has multiple incoming edges, then the scheduling decisions that lead to v_p must involve the same set of job segments on any two paths from v_1 to v_p.

Property 1. *(adapted from [33]) For any two paths P and Q, if both P and Q start at v_1 and end in v_p, then $\mathcal{J}^P = \mathcal{J}^Q$.*

5.2 System State Representation

As previously stated, we consider that jobs can be subject to release jitter and that there can be variations in the execution times of its segments. As a consequence of this uncertainty, we must compute a finish time interval $[EFT_{i,j}, LFT_{i,j}]$ for each segment $J_{i,j}$ of a job J_i after a sequence of scheduling decisions taken by the scheduler. This interval is bounded by the job segment's

earliest and latest finish times $EFT_{i,j}$ and $LFT_{i,j}$ in any execution scenario that complies with the assumed sequence of scheduling decisions. Thus, we say that a job segment $J_{i,j}$ of job J_i can possibly finish at or after $EFT_{i,j}$ and is certainly finished at $LFT_{i,j}$. This uncertainty in the finish times of segments introduces a challenge as it also means that we have uncertainty in processor and shared-resource availability times, which then affect the finish time intervals of subsequent job segments.

To address this challenge, we develop a new abstraction to encode information about the system state. In each vertex v_p, the system state abstraction records the availability of the cores and shared resources for any execution scenario that complies with the sequence of scheduling decisions modeled by the edges on the paths leading to v_p.

As discussed in Section 4.3, a core is either free to execute a ready job or claimed by a job that was previously dispatched by the scheduler, i.e., a job has started executing on the core and not all of its segments have finished executing yet. Thus, let $\mathcal{C}(v_p)$ denote the set of jobs that claimed a core (where $|\mathcal{C}(v_p)| \leq m$). A system state is then modeled as follows:

- **Claimed-core availability intervals.** For each job $J_i \in \mathcal{C}(v_p)$, we record the interval $[Cl_i^{min}(v_p), Cl_i^{max}(v_p)]$ where $Cl_i^{min}(v_p)$ and $Cl_i^{max}(v_p)$ indicate when the core claimed by job J_i becomes *possibly* and *certainly* available to execute the next segment of J_i, respectively.

- **Free-core availability intervals.** We record $m - |\mathcal{C}(v_p)|$ intervals $A_x(v_p) = [A_x^{min}(v_p), A_x^{max}(v_p)]$ such that $1 \leq x \leq m - |\mathcal{C}(v_p)|$ indicating when one, two, three, ..., $m - |\mathcal{C}(v_p)|$ cores become possibly and certainly available to execute *ready jobs*.

- **Shared-resource availability intervals.** For each shared resource $\ell_x \in \mathcal{L}$, a state records the interval $[SR_x^{min}(v_p), SR_x^{max}(v_p)]$ where $SR_x^{min}(v_p)$ and $SR_x^{max}(v_p)$ denote the times at which ℓ_x become possibly and certainly available to be accessed by a new job segment, respectively.

For ease of reference, we also introduce the two notations $SR_{i,j}^{min}(v_p)$ and $SR_{i,j}^{max}(v_p)$ to refer to the availability interval of the resource requested by segment $J_{i,j}$ (if any). Hence, if $J_{i,j}$ requests a shared resource (i.e., $\exists \ell_x \in \eta_{i,j}$), then $SR_{i,j}^{min}(v_p)$ and $SR_{i,j}^{max}(v_p)$ are the earliest and latest availability time $SR_x^{min}(v_p)$ and $SR_x^{max}(v_p)$ of the resource ℓ_x requested by $J_{i,j}$. If $J_{i,j}$ does not request a resource, then simply $SR_{i,j}^{min}(v_p) = SR_{i,j}^{max}(v_p) = 0$.

Example. Fig. 5.1 shows how an abstract state v_p in the SAG evolves into a new state v_q when a ready job segment $J_{k,2}$ is scheduled. On the left, we show the status of the cores before and after scheduling $J_{k,2}$. The right side shows the contents of the abstract state v_p and how it evolves after scheduling $J_{k,2}$.

Since J_i has only one segment, its start time defines its finish time interval and thus the time at which Core 1 may become available for ready jobs (between times 4 and 8). Thus, v_p records that one core is free for a ready job to execute within the interval $[4, 8]$ (i.e., $A_1(v_p) = [4, 8]$). State v_p also records that one core is claimed by job J_k whose first segment $J_{k,1}$ will potentially and certainly complete by time 3 (i.e., $Cl_k = [3, 3]$). Moreover, since none of the two segments executing in v_p access a shared resource, the shared resource ℓ_1 is certainly available from time 0 (i.e., $SR_1 = [0, 0]$).

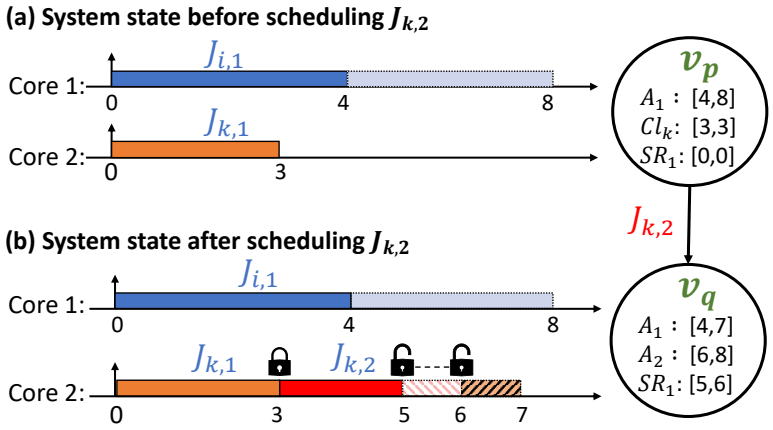


Figure 5.1: **Evolving a state v_p to v_q.**
Two successive system states for $m = 2$ and two jobs J_i (with one segment) and J_k (with two segments) released at time 0. Only the second segment of J_k requests a shared resource ℓ_1 with a critical section length $L_{k,2} \in [2, 3]$. The execution times of the segments are $C_{i,1} \in [4, 8]$, $C_{k,1} \in [3, 3]$ and $C_{k,2} \in [3, 4]$.

Prior to executing $J_{k,2}$, the lock protecting its required shared resource must be acquired. Since the shared resource is available, the resource is granted to $J_{k,2}$ as soon as the core already claimed by job J_k becomes available, i.e., at time 3. Since $J_{k,2}$'s critical section has a variable execution time, it may release the lock any time in the interval $[5, 6]$. This is recorded in the state v_q with the interval $SR_1 = [5, 6]$. Moreover, since $J_{k,2}$ is the last segment of its job, as soon as it finishes, Core 2 is released and becomes available for other jobs. Hence, in state v_q, no core is claimed anymore. Since the execution time of $J_{k,2}$ ranges from 3 to 4 time units, the core claimed by $J_{k,2}$ will become available in the interval $[6, 7]$. This means that one core becomes possibly available at time 4 (Core 1), one core is certainly available at time 7 (Core 2), two cores are possibly available at time 6, and two cores are certainly available at time 8. Thus, v_q records the availability intervals $A_1(v_q) = [4, 7]$ and $A_2(v_q) = [6, 8]$.

5.3 Schedule-Abstraction Graph Generation

The SAG is built iteratively. Each iteration comprises two phases, namely the *expansion phase* and the *merge phase*. In the expansion phase, (one of) the shortest path(s) P in the graph is picked and expanded for every job segment that can possibly start executing next in the state v_p at the end of P. Our model considers a JLFP scheduling policy, but we expand the schedule abstraction graph on a segment level. To recreate the behavior of a JLFP scheduler, in every state, we choose the highest priority ready segment (i.e., segment-level fixed priority) that can start executing at the given time. For every such job segment $J_{i,j}$, we create a new vertex v'_p in the graph, which represents a new system state and is connected to v_p via a directed edge labeled with $J_{i,j}$. The information encoded in the new state v'_p contains an updated version of the free cores, claimed cores and shared-resource availability intervals reflecting that $J_{i,j}$

Algorithm 1: Construction of the SAG for job set $\mathcal{J}$.

Inputs : Job set $\mathcal{J}$
Output: Bounds on the BCRT and WCRT of every job in $\mathcal{J}$

1 $\forall J \in \mathcal{J}, BR_i \leftarrow \infty, WR_i \leftarrow 0$;
2 Initialize G by adding $v_1 = (\{[0,0],...,[0,0]\}, \{[0,0],...,[0,0]\})$;
3 **while** $\exists$ *path* P *from* v_1 *to a leaf vertex such that* $P \neq \mathcal{J}$ **do**
4 P $\leftarrow$ the shortest path from v_1 to a leaf vertex v_p;
5 $\mathcal{R}^P \leftarrow$ set of potentially ready segments obtained with Eq. (5.1);
6 **for** *each segment* $J_{i,j} \in \mathcal{R}^P$ **do**
7 **if** $J_{i,j}$ *can start executing after* v_p *(Theorem 1 or 2)* **then**
8 Build v'_p using Algorithm 2;
9 **if** $J_{i,j} = J_{i,n_i}$ **then**
10 $BR_i \leftarrow \min\{EFT(v'_p) - r_i^{min}, BR_i\}$;
11 $WR_i \leftarrow \max\{LFT(v'_p) - r_i^{max}, WR_i\}$;
12 **end**
13 Connect v_p to v'_p with an edge labeled $J_{i,j}$;
14 **if** $\exists$ *path* Q *that ends in* v_q *such that Rule 1 is satisfied for* v'_p *and* v_q **then**
15 Merge v'_p and v_q in v_z using Eq. (5.21)-(5.23);
16 Redirect all incoming edges of v_q and v'_p to v_z;
17 Remove v_q and v'_p from G;
18 **end**
19 **end**
20 **end**
21 **end**

has now started to execute.

After the graph has been expanded with new states, the merge phase commences. The purpose of the merge phase is to slow down the growth of the graph, to postpone a potential state space explosion for as long as possible. This is achieved by merging any two "similar" states that terminate paths with identical sets of job segments. To preserve soundness, every system state that is reachable from any of the two original states must also be reachable from the merged state, which ensures that no possible execution scenario is discarded.

The exploration completes when no vertex remains to be expanded, i.e., all job segments have been included on every path. At this point, each path represents a set of valid execution scenarios and every possible schedule has been explored.

The complete SAG construction procedure is given in Algorithm 1. Note that it uses two variables (WR_i and BR_i) for each job in $\mathcal{J}$ to keep track of its worst-case (WCRT) and best-case response time (BCRT) on any path in the graph. Those bounds are updated (lines 10–11) whenever the last segment of a job is scheduled on a path, hence indicating the completion of that job in that execution scenario. If, by the end of the exploration, no job has a WCRT exceeding its deadline, then the analysis deems the job set schedulable.

5.4 Expansion Phase

We now consider the expansion and merge phases and show how a path ending in state v_p is expanded to a new state v_q.

5.4.1 Overview

First, we build a set of potentially *ready* job segments in state v_p, i.e., the segments that have not started executing on path P and for which all preceding segments have started and potentially completed. For each such segment $J_{i,j}$, we compute the earliest and latest time $EST_{i,j}(v_p)$ and $LST_{i,j}(v_p)$ at which the segment can start executing in v_p. That is, $EST_{i,j}(v_p)$ denotes the earliest time at which a global work-conserving JLFP scheduler would allow $J_{i,j}$ to start in v_p. Similarly, $LST_{i,j}(v_p)$ indicates the latest time at which $J_{i,j}$ must have certainly started executing if it is the next segment to succeed v_p. If $J_{i,j}$ has not started by $LST_{i,j}(v_p)$, a different segment must have started to execute after v_p. Hence, a segment is said to be eligible to be a successor of v_p only if its earliest start time $EST_{i,j}(v_p)$ is no later than its latest start time $LST_{i,j}(v_p)$. For each eligible job segment, a new vertex v'_p is added to the graph, where v'_p encodes the system state after $J_{i,j}$ started executing. In addition to deciding whether a job segment is eligible to start executing after state v_p, $EST_{i,j}(v_p)$ and $LST_{i,j}(v_p)$ also help compute the earliest and latest finish times $EFT_{i,j}(v_p)$ and $LFT_{i,j}(v_p)$ of $J_{i,j}$.

Next, each step of the expansion phase is explained in detail.

5.4.2 Set of Potentially Ready Job Segments

We consider that a job segment is ready if the job has been released and all its preceding segments have completed. Thus, we define the set of *potentially ready* segments $\mathcal{R}^P$ for path P to be the set of segments that have not yet started to execute (i.e., $J_{i,j} \notin \mathcal{J}^{\mathcal{P}}$) and whose predecessor (if any) has already started and potentially completed its execution (i.e., either $J_{i,j}$ is the first segment of its job J_i, thereby meaning $j = 1$, or $J_{i,j-1} \subseteq \mathcal{J}^{\mathcal{P}}$).

$$\mathcal{R}^P \triangleq \{J_{i,j} \mid J_{i,j} \notin \mathcal{J}^{\mathcal{P}} \wedge (j = 1 \vee J_{i,j-1} \subseteq \mathcal{J}^{\mathcal{P}})\} \tag{5.1}$$

5.4.3 Earliest Start Time

Since we consider that jobs may suffer from release jitter and execution-time variation the exact finishing times of preceding job segments cannot be known. Therefore, the exact time at which a segment $J_{i,j}$ may start to execute is also unknown. For each segment $J_{i,j}$ in $\mathcal{R}^P$, we prove a lower bound $EST_{i,j}(v_p)$ and an upper bound $LST_{i,j}(v_p)$ on the time at which $J_{i,j}$ may start executing in v_p (Equations (5.2) and (5.10), respectively).

$$EST_{i,j}(v_p) = \begin{cases} \infty & \text{if } j = 1 \wedge |\mathcal{C}(v_p)| = m \\ \max\{r_i^{min},\ A_1^{min}(v_p),\ SR_{i,j}^{min}(v_p)\} & \text{if } j = 1 \wedge |\mathcal{C}(v_p)| < m \\ \max\{\mathit{Cl}_i^{min}(v_p),\ SR_{i,j}^{min}(v_p)\} & \text{if } j > 1 \end{cases} \tag{5.2}$$

Lemma 1. *Segment $J_{i,j} \in \mathcal{R}^P$ cannot start executing (as a successor of state v_p) before $EST_{i,j}(v_p)$.*

Proof. The first segment $J_{i,1}$ of a job J_i can start its execution only if (i) it is released, (ii) the shared resource ℓ_x it requests (if any) is available **and** (iii) a

core is available. Thus, if all cores have already been claimed by other jobs (i.e., $|\mathcal{C}(v_p)| = m|$) then $J_{i,1}$ cannot be a successor of v_p and $EST_{i,j}(v_p) = \infty$. This proves the first case of Eq. (5.2).

However, if there is a free core (i.e., $|\mathcal{C}(v_p)| < m|$), then, by definition, A_1^{min} is the earliest time at which a core can potentially become available. Furthermore, r_i^{min} is the earliest release time of J_i and $SR_{i,j}^{min}(v_p)$ is the earliest time at which the shared resource accessed by $J_{i,j}$ may become available. Thus, the earliest time at which $J_{i,1}$ may start to execute is given by $EST_{i,j}(v_p) = \max\{r_i^{min},\ A_1^{min}(v_p),\ SR_{i,j}^{min}(v_p)\}$ if $j = 1$. This proves the second case of Eq. (5.2).

Any segment that is not the first segment of a job (i.e., a segment $J_{i,j}$ with $j > 1$) can start its execution only if (i) the core claimed by the preceding segments belonging to the same job is available, **and** (ii) the shared resource it requests (if any) is also available. Since $J_{i,j}$ is in $\mathcal{R}^P$, all the segments of J_i that precede $J_{i,j}$ must have started (and potentially finished) executing on the core claimed by J_i. Thus, the earliest time at which the core claimed by J_i may become available for $J_{i,j}$ is given by $Cl_i^{min}(v_p)$. Therefore, the earliest time at which $J_{i,j}$ may start to execute is $\max\{Cl_i^{min}(v_p),\ SR_{i,j}^{min}(v_p)\}$ if $j > 1$. This proves the last case of Eq. (5.2). □

5.4.4 Latest Start Time

The latest start time $LST_{i,j}(v_p)$ of segment $J_{i,j}$ is computed considering that the scheduler is (i) work-conserving and (ii) that it follows a non-preemptive JLFP scheduling policy.

First, focus on (i). By definition, a work-conserving scheduler *must* execute a segment as soon as the segment is certainly ready and a core is certainly free. We can thus prove the following two lemmas and their corollary.

Lemma 2. *An upper bound on the time at which a segment $J_{y,z}$ can certainly start executing (as a successor of state v_p) is given by*

$$tw_{y,z} = \begin{cases} \infty & \text{if } z = 1 \wedge |\mathcal{C}(v_p)| = m; \\ \max\{r_y^{max},\ A_1^{max},\ SR_{y,z}^{max}(v_p)\} & \text{if } z = 1 \wedge |\mathcal{C}(v_p)| < m; \\ \max\{Cl_y^{max}(v_p),\ SR_{y,z}^{max}(v_p)\} & \text{if } z > 1. \end{cases} \tag{5.3}$$

Proof. Infinity is an obvious upper bound on the start time of $J_{y,z}$. Therefore, in this proof, we only focus on the cases where $tw_{y,z} \neq \infty$.

A starting segment $J_{y,1}$ *must* start to execute as soon as (i) it is released, (ii) the resource ℓ_x it requests (if any) is available and (iii) a core is available. By definition, r_y^{max} is an upper bound on the release time of $J_{y,1}$, $SR_{y,z}^{max}(v_p)$ is an upper bound on the availability time of the shared resource accessed by $J_{y,z}$ (if any) and $A_1^{max}(v_p)$ denotes the time at which a core is certainly available in state v_p. Thus, $J_{y,1}$ can certainly start executing at $\max\{r_y^{max},\ A_1^{max},\ SR_{y,1}^{max}(v_p)\}$ when $z = 1$ and there is at least one free core in v_p.

Any segment $J_{y,z}$ that is not the first segment of $J_{y,z}$ (i.e., with $z > 1$) can certainly start executing when (i) the resource it requests (if any) is available, and (ii) the segments of J_y that precede $J_{y,z}$ have all completed their execution on the core claimed by J_y. Since $J_{y,z}$ is in $\mathcal{R}^P$, all the segments of J_y preceding

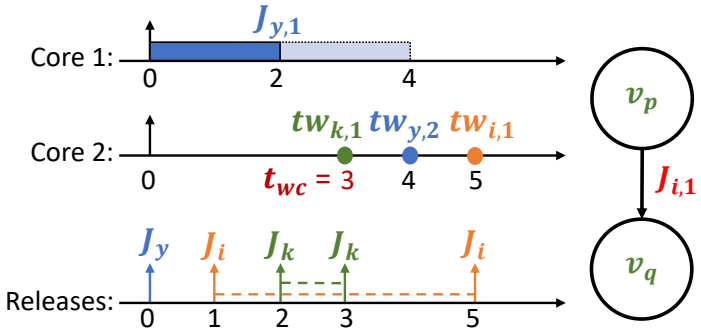


Figure 5.2: **Computing t_{wc}.**
Determining t_{wc} by comparing the different tw's in a system with 2 cores ($m = 2$) and three jobs J_i (with one segment), J_k (with one segment) and J_y (with two segments). The relevant properties of the jobs are given as follows: J_y has a release time of $r_y \in [0, 0]$, while J_i is released within $r_i \in [1, 5]$ and J_k arrives to the system in the interval of $r_k \in [2, 3]$. The first segment of J_y has an execution time of $C_{y,1} \in [2, 4]$ and therefore a core is claimed in v_p for that period. We assume that none of the jobs request shared resources. We further assume equal priorities between the jobs, i.e., $p_y = p_i = p_k$.

$J_{y,z}$ have already started (and potentially finished) executing on the core claimed by J_y, and because $Cl_y^{max}(v_p)$ is an upper bound on the time at which the core claimed by J_y becomes available to execute the next segment of J_y, we have that $J_{y,z}$ certainly starts at $\max\{Cl_y^{max}(v_p),\ SR_{y,z}^{max}(v_p)\}$ when $z > 1$. □

Lemma 3. *A work-conserving scheduler will start executing a job segment no later than* [1]

$$t_{wc} = \min_{\infty}\{tw_{y,z} \mid J_{y,z} \in \mathcal{R}^P\} \tag{5.4}$$

Proof. By Lemma 2, a job segment $J_{y,z} \in \mathcal{R}^P$ is certainly ready to start executing at time $tw_{y,z}$. Thus there is at least one job segment that is ready to execute at $\min_{\infty}\{tw_{y,z} \mid J_{y,z} \in \mathcal{R}^P\}$. Thus, t_{wc} is an upper-bound on the time at which a work-conserving scheduler will start executing a job segment. □

Corollary 1. *A segment $J_{i,j} \in \mathcal{R}^P$ cannot be a direct successor of a state v_p if it starts executing any later than t_{wc}.*

Proof. Since the scheduler will certainly allow a job segment to execute by time t_{wc}, $J_{i,j}$ must start executing at or before t_{wc} if it is the segment to succeed v_p. □

Example 1. Fig. 5.2 shows a system state v_p, where one of the three considered jobs (J_y) has claimed core 1 and started executing its workload. In this example, we consider the scenario in which $J_{i,1}$ succeeds state v_p, i.e., it is the next segment to start executing after v_p. In order to calculate the latest start time

[1] $\min_{\infty}\{X\} = \min\{X \cup \{\infty\}\}$ where X is a set of positive values.

(LST) of $J_{y,1}$, we first need to determine the time at which the scheduler will certainly allow a segment to start executing due to work conservation (t_{wc}). J_i's latest release time is at time 5, however we can see that $J_{y,2}$ and $J_{k,1}$ are certainly ready before that time. This means that in order for $J_{i,1}$ to evolve state v_p to v_q by being the next segment on the path, it cannot start later than the earliest time at which a segment of a different job is certainly ready and a core is available to execute it. We investigate the other job segments and we see that by time 4, $J_{y,1}$ has certainly finished executing, meaning that $J_{y,2}$ will certainly be ready to start its workload. Furthermore, J_k's latest release (r_k^{max}) is at time 3, which means that the scheduler will certainly dispatch J_k at that time. Hence, it is not possible for $J_{i,1}$ to start executing next at time 5 (r_i^{max}) in v_p, because $J_{y,2}$ and $J_{k,1}$ would have certainly started by time 4 and 3 respectively. Therefore, if we consider the path P where $J_{i,1}$ succeeds system state v_p, the latest time at which $J_{i,1}$ can start to execute is $t_{wc} = 3$, because by time 3 we are certain that $J_{k,1}$ will start its execution if $J_{i,1}$ did not.

Now that we covered work conservation, we consider the impact of the JLFP scheduling policy. In the following we derive the time t_{high} at which a considered segment $J_{i,j}$ is certainly not the highest-priority segment anymore in state v_p. This means that $J_{i,j}$ cannot be a successor in state v_p, if it has not started executing before t_{high}. The actual value of t_{high} depends on the specific resource access policy (FIFO- or priority-ordered) used by the spin-locking protocol. Thus, we first discuss how to compute it for FIFO-ordered spin locks, and then for priority-ordered ones.

FIFO-Ordered Spin Locks

First, assume that two segments $J_{i,j}$ and $J_{y,z}$ request the same resource. Since a FIFO spin lock provides access to the shared resource based on the order in which requests were made, the order in which those segments will start executing does not depend on their priority. Therefore, the JLFP scheduling policy does not impact the start time of segments that share the same resource. Then, let $\mathcal{H}$ denote the set of segments with a higher priority than $J_{i,j}$ that do not share a resource with $J_{i,j}$, i.e., $\mathcal{H} = \{J_{y,z} \mid J_{y,z} \in \{\mathcal{R}^P \cap hp(J_{i,j})\} \wedge \eta_{y,z} \cap \eta_{i,j} = \emptyset\}$. Let t_{high}^{FIFO} be an upper-bound on the time at which a considered segment $J_{i,j}$ is certainly not the highest-priority segment in state v_p anymore. For FIFO-ordered spin locks, this means that any segment in $\mathcal{H}$ will certainly be ready to execute at t_{high}^{FIFO}. We prove (Lemmas 4 and 5) that t_{high}^{FIFO} can be computed with Eq. (5.5).

$$t_{high}^{FIFO} = \min_{\infty}\{th_{y,z}^{FIFO}(J_{i,j}) \mid J_{y,z} \in \mathcal{H}\} \tag{5.5}$$

where

$$th_{y,z}^{FIFO}(J_{i,j}) = \begin{cases} \max\{r_y^{max},\ SR_{y,z}^{max}(v_p)\} & \text{if } z = j = 1 \\ \max\{A_1^{max}(v_p), r_y^{max},\ SR_{y,z}^{max}(v_p)\} & \text{if } z = 1 \wedge j > 1 \\ \max\{Cl_y^{max}(v_p),\ SR_{y,z}^{max}(v_p)\} & \text{if } z > 1 \end{cases} \tag{5.6}$$

Lemma 4. *Let $J_{y,z}$ be a segment in $\mathcal{H}$. If $J_{i,j}$ did not start to execute before $th_{y,z}^{FIFO}(J_{i,j})$, then $J_{y,z}$ will start before $J_{i,j}$.*

Proof. We analyze the three cases of Eq. (5.6).
Case 1. Assume that both $J_{i,j}$ and $J_{y,z}$ are the first segment of their respective job (i.e., $j = z = 1$). For a starting segment to be able to execute, it needs to be (1) released, (2) the resource it requests (if it requests one) must be available and (3) a core must be available for it to execute on. By definition, r_y^{max} is an upper bound on (1) and $SR_{y,z}^{max}(v_p)$ is an upper bound on (2). Regarding (3), we note that because $J_{y,1}$ and $J_{i,1}$ are both starting segments, they both compete for the same available cores. Since $J_{y,1}$ has a higher priority than $J_{i,j}$, if both $J_{y,1}$ and $J_{i,j}$ are ready and have their shared resources available at the same time, then $J_{y,1}$ will certainly start before $J_{i,j}$. Therefore, only (1) and (2) decide whether $J_{y,1}$ will start to execute before $J_{i,j}$. Thus, if $J_{i,j}$ did not start before $\max\{r_y^{max}, SR_{y,z}^{max}(v_p)\}$ when $z = j = 1$, then $J_{y,1}$ will be dispatched before $J_{i,j}$.

If $J_{i,j}$ is not a starting segment (i.e., $j > 1$), then it already has a reserved core. Therefore, it does not compete with $J_{y,z}$ for the same core (i.e., we must account for (3)).
Case 2. If $J_{y,z}$ is the first segment of the higher priority job J_y, then, by definition, $A_1^{max}(v_p)$ is a safe upper bound on (3). Thus, because $J_{y,z}$ has higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{A_1^{max}(v_p), r_y^{max}, SR_{y,z}^{max}(v_p)\}$ when $z = 1 \wedge j > 1$.
Case 3. If $J_{y,z}$ is not the first segment of the higher priority job J_y, then job J_y is already released and it already claimed a core. Thus, by definition, $Cl_y^{max}(v_p)$ is an upper bound on (3). Hence, as $J_{y,z}$ has higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{Cl_y^{max}(v_p), SR_{y,z}^{max}(v_p)\}$ when $z > 1$. □

Lemma 5. *A segment $J_{i,j} \in \mathcal{R}^P$ cannot be the direct successor of a state v_p if it starts executing later than $t_{high}^{FIFO} - 1$, in a system using FIFO-ordered spin locks.*

Proof. According to Lemma 4, $\forall J_{y,z} \in \mathcal{H}$, $J_{y,z}$ will start before $J_{i,j}$ if $J_{i,j}$ did not start before $th_{y,z}^{FIFO}(J_{i,j})$. Then, $J_{i,j}$ will not be the next segment to *succeed* v_p, if $J_{i,j}$ did not start before any segment $J_{y,z}$ becomes certainly ready to execute i.e., $\min_\infty\{th_{y,z}^{FIFO}(J_{i,j}) \mid J_{y,z} \in \mathcal{H}\}$. □

Example 2. Fig. 5.3 shows a system state v_p, where two of the three considered jobs (J_y and J_i) have claimed cores and started executing their workload. In this example, we consider the scenario where $J_{i,2}$ succeeds state v_p, i.e., it is the next segment to start executing after v_p. In addition to the work conserving behavior of the scheduler, we need to determine the earliest time at which a higher priority segment is certainly ready to execute (t_{high}^{FIFO}), because we are sure that by that time, $J_{i,2}$ cannot be the next segment to start after state v_p. We investigate the two higher priority jobs in the system (J_y and J_k) and we see that J_k is certainly released at time 6 and J_y's claimed core becomes certainly available for $J_{y,2}$ at time 4. Hence, the higher priority segments $J_{k,1}$ and $J_{y,2}$ will certainly start to execute at time 6 and 4 respectively. In v_p, the core claimed by J_i becomes available for our considered job segment $J_{i,2}$ in the interval of $[3, 5]$. While this interval represents a possible start time for $J_{i,2}$ when we only consider $J_{k,1}$ (since $J_{k,1}$ could start at time 6), we see that it is impossible for $J_{i,2}$ to start at time 5 if we also take into account $J_{y,2}$. In fact,

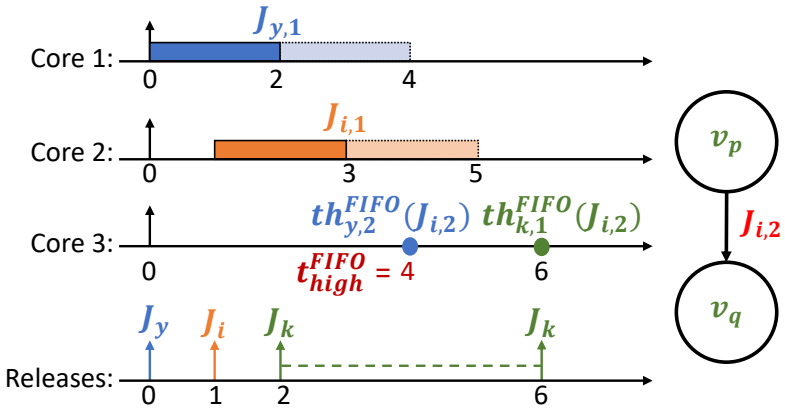


Figure 5.3: **Computing t^{FIFO}_{high} for FIFO-ordered spin locks.** Determining t^{FIFO}_{high} by comparing the different $th^{FIFO}_{y,z}(J_{i,j})$'s in a system with 3 cores ($m = 3$) and three jobs J_i (with two segments), J_k (with one segment) and J_y (with two segments). The relevant properties for this example are given as follows: J_y has a release time of $r_y \in [0, 0]$, while J_i is released at $r_i \in [1, 1]$ and J_k arrives to the system in the interval of $r_k \in [2, 6]$. The first segments of J_y and J_i have an execution time of $C_{y,1} \in [2, 4]$ and $C_{i,1} \in [3, 5]$ respectively. A core for each of these two jobs is claimed in v_p since they consist of multiple segments. We assume that both jobs J_y and J_k have a higher priority than J_i, i.e., $p_y < p_i$ and $p_k < p_i$. We further assume that no shared resource availability affects the starting of the considered job segments.

considering the scenario where $J_{i,2}$ is the next segment to start executing after v_p, $J_{i,2}$ must start prior to time 4, because at that point in time $J_{y,2}$ becomes certainly ready to execute. Recalling that a JLFP scheduler always starts to execute the highest priority ready segment first, $J_{i,2}$ cannot succeed v_p if it has not started prior to time $t^{FIFO}_{high} = 4$, since by that time, $J_{i,2}$ is no longer the highest priority ready segment in the system.

Priority-Ordered Spin Locks

For priority-ordered spin locks, the equation to compute t^{prio}_{high} differs from the solution we have just presented for FIFO-based spin locks. Here, a higher priority segment $J_{y,z}$ can in fact impact the execution of another segment $J_{i,j}$ even when they share the same resource. Lemmas 6 and 7 prove that t^{prio}_{high} can be computed as in Eq. (5.7) if a priority-based locking protocol is used.

$$t^{prio}_{high} = \min_{\infty}\{th^{prio}_{y,z}(J_{i,j}) \mid J_{y,z} \in \{\mathcal{R}^P \cap\ hp(J_{i,j})\}\} \tag{5.7}$$

where

$$th^{prio}_{y,z}(J_{i,j}) = \begin{cases} \max\{r^{max}_y,\ t_r\} & \text{if } z = j = 1 \\ \max\{A^{max}_1(v_p),\ r^{max}_y,\ t_r\} & \text{if } z = 1 \wedge j > 1 \\ \max\{Cl^{max}_y(v_p),\ t_r\} & \text{if } z > 1 \end{cases} \tag{5.8}$$

with

$$t_r = \begin{cases} SR_{y,z}^{max}(v_p) & \text{if } \eta_{y,z} \neq \eta_{i,j} \\ 0 & \text{otherwise.} \end{cases} \tag{5.9}$$

Lemma 6. *Let $J_{y,z} \in \{\mathcal{R}^P \cap\ hp(J_{i,j})\}$ be a segment that has a higher priority than the considered segment $J_{i,j}$. If $J_{i,j}$ did not start to execute before $th_{y,z}^{prio}(J_{i,j})$, then $J_{y,z}$ will start before $J_{i,j}$.*

Proof. We analyze the three cases of Eq. (5.8).
Case 1. Assume that both $J_{i,j}$ and $J_{y,z}$ are the first segment of their respective job (i.e., $j = z = 1$). For a starting segment to be able to execute, it needs to be (1) released, (2) the resource it requests (if it requests one) must be available and (3) a core must be available for it to execute on. By definition, r_y^{max} is an upper bound on (1) and $SR_{y,z}^{max}(v_p)$ is an upper bound on (2). Regarding (3), we note that because $J_{y,1}$ and $J_{i,1}$ are both starting segments, they both compete for the same available cores. Since $J_{y,1}$ has a higher priority than $J_{i,1}$, if both $J_{y,1}$ and $J_{i,1}$ are ready and have their shared resources available at the same time, then $J_{y,1}$ will certainly start before $J_{i,1}$. Therefore, only (1) and (2) decide whether $J_{y,1}$ will start to execute before $J_{i,1}$. Similarly, we only consider $t_r = SR_{y,z}^{max}(v_p)$ in (2) if $J_{y,z}$ does not request the same resource as $J_{i,j}$ (i.e., $\eta_{y,z} \neq \eta_{i,j}$), because otherwise it would mean that they compete for the same resource in which case $J_{y,1}$ will certainly start before $J_{i,1}$, due to its higher priority, if they are both ready and a core is available. If they do share the same resource or any of the segments is resource-independent, we simply ignore (2) (i.e., set $t_r = 0$). Thus, if $J_{i,j}$ did not start before $\max\{r_y^{max},\ t_r\}$ when $z = j = 1$, then $J_{y,z}$ will be dispatched before $J_{i,j}$.

If $J_{i,j}$ is not a starting segment (i.e., $j > 1$), then it already has a reserved core. Therefore, it does not compete with $J_{y,z}$ for the same core (i.e., we must account for (3)).
Case 2. If $J_{y,z}$ is the first segment of the higher priority job J_y, then, by definition, $A_1^{max}(v_p)$ is a safe upper bound on (3). Thus, because $J_{y,z}$ has higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{A_1^{max}(v_p),\ r_y^{max},\ t_r\}$ when $z = 1 \wedge j > 1$.
Case 3. If $J_{y,z}$ is not the first segment of the higher priority job J_y, then job J_y is already released and it already claimed a core. Thus, by definition, $Cl_y^{max}(v_p)$ is an upper bound on (3). Hence, as $J_{y,z}$ has a higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{Cl_y^{max}(v_p),\ t_r\}$ when $z > 1$. □

Lemma 7. *A segment $J_{i,j} \in \mathcal{R}^P$ cannot be the direct successor of a state v_p if it starts executing later than $t_{high}^{prio} - 1$, in a system using priority-ordered spin locks.*

Proof. According to Lemma 6, $\forall J_{y,z} \in \{\mathcal{R}^P \cap hp(J_{i,j})\}$, $J_{y,z}$ will start before $J_{i,j}$ if $J_{i,j}$ did not start before $th_{y,z}^{prio}(J_{i,j})$. Then, $J_{i,j}$ will not be the next segment to succeed v_p if $J_{i,j}$ did not start before $\min_\infty\{th_{y,z}^{prio}(J_{i,j}) \mid J_{y,z} \in \{\mathcal{R}^P \cap hp(J_{i,j})\}\}$. □

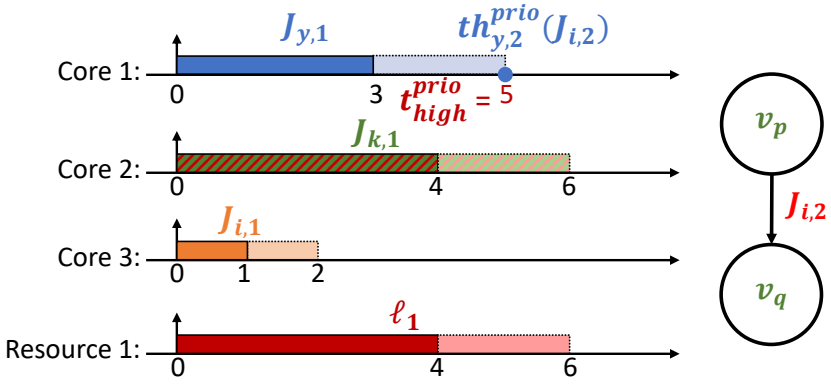


Figure 5.4: **Computing t^{prio}_{high} for priority-ordered spin locks.** Determining t^{prio}_{high} in state v_p by comparing the different $th^{prio}_{y,z}(J_{i,j})$'s in a system with 3 cores ($m = 3$) and three jobs J_i (with two segments), J_k (with one segment) and J_y (with two segments). All the jobs are released at time 0 ($r_y = r_k = r_i \in [0, 0]$). $J_{y,1}$ has an execution time of $C_{y,1} \in [3, 5]$ and occupies core 1 for that period in v_p. Similarly, $J_{k,1}$ executes on core 2 in the interval of $C_{k,1} \in [4, 6]$ with $L_{k,1} \in [4, 6]$. $J_{k,1}$ accesses resource 1 (ℓ_1). $J_{i,1}$, which is the predecessor of our considered segment ($J_{i,2}$), occupies core 3 for $C_{i,1} \in [1, 2]$. Both remaining segments ($J_{y,2}$ and $J_{i,2}$) request resource 1 (ℓ_1) and are therefore dependent on its availability. We assume that both jobs J_y and J_k have a higher priority than J_i, i.e., $p_y < p_i$ and $p_k < p_i$.

Example 3. Fig. 5.4 shows a system state v_p, where all three cores are occupied by the three jobs in the system. In this example, we consider the scenario where $J_{i,2}$ succeeds state v_p, i.e., it is the next segment to start executing after v_p. In contrast to the example shown in Fig. 5.3, due to the priority queue, we must also consider higher priority jobs that access the same resource as $J_{i,2}$ when computing t^{prio}_{high}. First, we note that $J_{k,1}$ keeps the resource ℓ_1 in use and releases it between time 4 and 6. Considering the remaining segments, $J_{i,2}$ becomes certainly ready by time 2, while $J_{y,2}$ can start executing between time 3 and 5. Using a FIFO queue, we would be certain that $J_{i,2}$ will definitely have access granted to the resource next, since its latest request time is still earlier than $J_{y,2}$'s earliest request time. This means that in this example, $J_{y,2}$ would not affect the start time of $J_{i,2}$ considering we have a FIFO queue. However, as we deal with a priority queue, as soon as $J_{y,2}$ becomes ready while $J_{i,2}$ has not started to execute yet, the resource will be granted to $J_{y,2}$. Since $J_{y,2}$ becomes certainly ready at time 5, $J_{i,2}$ must execute prior to $t^{prio}_{high} = 5$, otherwise it will certainly not be the highest ready segment anymore to be granted the resource and to succeed state v_p.

Considering *both* computations of t_{high} (t^{FIFO}_{high} for FIFO-ordered spin locks and t^{prio}_{high} for priority-ordered spin locks), it directly follows that an upper bound on the start time of $J_{i,j}$ can be computed according to Lemma 8.

Lemma 8. *A segment $J_{i,j} \in \mathcal{R}^P$ can be the first segment to start executing in state v_p only if it starts no later than*

$$LST_{i,j} = \min\{t_{wc},\ t_{high} - 1\} \tag{5.10}$$

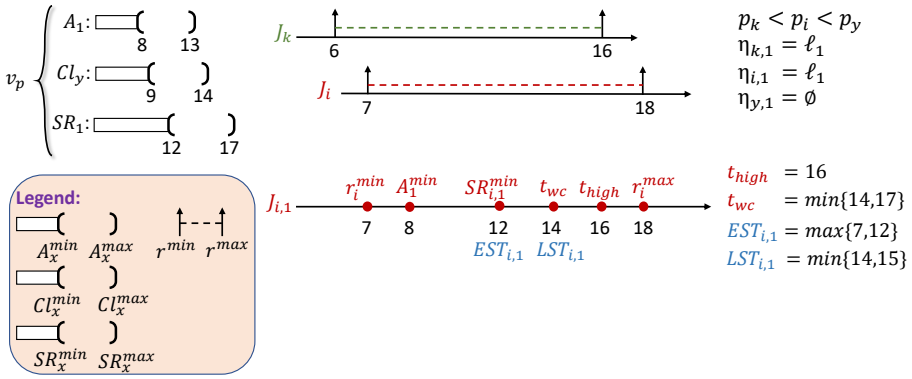


Figure 5.5: **Computing $EST_{i,j}$ and $LST_{i,j}$.**

Proof. Since $LST_{i,j} \leq t_{wc}$ (by Cor. 1) and $LST_{i,j} \leq t_{high} - 1$ (by Lemma 5 and Lemma 7), where $t_{high} = t_{high}^{FIFO}$ or $t_{high} = t_{high}^{prio}$ depending on whether FIFO- or priority-ordered spin locks are used, the claim holds. □

Example 4. Fig. 5.5 considers a state v_p in a system with two cores ($m = 2$) and three jobs (J_k, J_i and J_y). For this example we assume that shared resources are protected using priority-ordered spin locks. Job J_k has the highest priority and consists of one segment ($J_{k,1}$), which requests shared resource ℓ_1. Job J_i has a medium priority and also consist of one segment ($J_{i,1}$), which is our considered successor in this example. We assume that $J_{i,1}$ also requests the shared resource ℓ_1. Finally, Job J_y has the lowest priority and comprises two segments (which do not access ℓ_1) out of which the first one ($J_{y,1}$) has already started executing in state v_p. Hence, J_y has a claimed core which becomes possibly available at time 9 and certainly available at time 14. Furthermore, one core becomes available between time 8 and 13 for any new job. The lock protecting the shared resource ℓ_1 is released between time 12 and 17. J_i is potentially released at time 7 and the earliest time at which a core becomes possibly available is at $A_1^{min} = 8$. However, since $J_{i,1}$ accesses ℓ_1, it needs to wait until the resource is possibly available which happens at $SR_{i,1}^{min} = 12$ at the earliest. Therefore, $EST_{i,1} = 12$ marks the earliest time at which all necessary conditions (job is released and core and shared resource are available) are met. The higher priority job J_k is certainly released by time 16, meaning that J_i must start to execute prior to $t_{high} = 16$. Furthermore, by work conservation, $J_{y,2}$ will certainly start executing by time 14, which means that $J_{i,1}$ must not start later than $t_{wc} = 14$, otherwise it cannot be the successor in state v_p. Since $t_{wc} < t_{high} - 1$, $LST_{i,1} = 14$ is bounded by t_{wc}.

5.4.5 Eligibility Condition

Priority-Ordered Spin Locks

Now that we computed a lower bound $EST_{i,j}(v_p)$ on the earliest start time and an upper bound $LST_{i,j}(v_p)$ on the latest start time of $J_{i,j}$ in state v_p, it is rather

obvious that $J_{i,j}$ can be a successor to v_p only if

$$EST_{i,j}(v_p) \leq LST_{i,j}(v_p). \tag{5.11}$$

Theorem 1. *A segment $J_{i,j}$ is a direct successor of v_p only if $EST_{i,j}(v_p) < \infty$ and Inequality* (5.11) *holds.*

Proof. According to Eq. (5.2), if $EST_{i,j}(v_p) = \infty$ then $J_{i,j}$ is the first segment of job J_i and no free core is available to start to execute $J_{i,j}$ (i.e., every core is claimed by an unfinished job). Since we assume a non-preemptive system, the scheduler cannot dispatch $J_{i,j}$ on a core in v_p. Further, from Lemma 1, we know that $EST_{i,j}(v_p)$ is a lower bound on the earliest time at which $J_{i,j}$ can start executing in v_p. From Lemma 8, we have that $LST_{i,j}(v_p)$ is an upper bound on the time at which $J_{i,j}$ can start executing in v_p. Hence, if $EST_{i,j} > LST_{i,j}$, it creates a contradiction, thereby meaning that $J_{i,j}$ cannot start to execute in v_p and thus cannot be a successor to v_p. □

FIFO-Ordered Spin Locks

While systems using priority-ordered locks only need to consider Eq. (5.11) as an eligibility condition, systems using FIFO spin locks need to fulfill a second condition.

Consider the case where two segments $J_{i,j}$ and $J_{y,z}$ compete for the same shared resource. Since access to the resource is granted in FIFO order, the order in which those job segments will start to execute depends on the order of their requests. Assume that we know a lower bound on the earliest time at which $J_{i,j}$ may request its shared resource, and that we further have an upper bound on the latest time at which $J_{y,z}$ may request the resource. We refer to those bounds as $ERT_{i,j}(v_p)$ and $LRT_{y,z}(v_p)$, respectively. We prove in Theorem 2 that if $ERT_{i,j}(v_p) > LRT_{y,z}(v_p)$, then $J_{y,z}$ must execute before $J_{i,j}$, thereby implying that $J_{i,j}$ cannot be the first segment dispatched in state v_p.

To prove Theorem 2, we first prove a lower and an upper bound on $ERT_{i,j}(v_p)$ and $LRT_{y,z}(v_p)$, respectively.

Lemma 9. *Segment $J_{i,j} \in \mathcal{R}^P$ cannot request a shared resource in v_p earlier than*

$$ERT_{i,j}(v_p) = \begin{cases} \max\{r_i^{min}, A_1^{min}(v_p)\} & \text{if } j = 1; \\ Cl_i^{min}(v_p) & \text{if } j > 1. \end{cases} \tag{5.12}$$

Proof. The first segment $J_{i,1}$ of job J_i cannot request a resource prior to its release, i.e., not before r_i^{min}, nor can it request a resource prior to the earliest time at which a core may become available to start its execution, i.e., not before $A_1^{min}(v_p)$. Thus, the earliest time at which $J_{i,1}$ may request its resource is lower-bounded by $\max\{r_i^{min}, A_1^{min}(v_p)\}$ (hence case 1 of Eq. (5.12)). If $J_{i,j}$ is not the first segment of J_i, then J_i is already released and $J_{i,j}$ requests the resource as soon as preceding segment completes, which is lower-bounded by $Cl_i^{min}(v_p)$ (hence case 2 of Eq. (5.12)). □

Lemma 10. *A job segment $J_{y,z} \in \mathcal{R}^P$ will have certainly requested its shared resource by time*

$$LRT_{y,z}(v_p) = \begin{cases} \max\{r_y^{max}, A_1^{max}(v_p)\} & \text{if } j = 1; \\ Cl_y^{max}(v_p) & \text{if } j > 1. \end{cases} \tag{5.13}$$

Proof. A segment $J_{y,z}$ will certainly request its shared resource when (1) the job it belongs to has been released and (2) the segment has a core to execute on.

By definition, the first segment $J_{y,1}$ of a job J_y is certainly released at time r_y^{max} and a core is certainly available by time $A_1^{max}(v_p)$. Thus, $J_{y,1}$ will have certainly requested its resource by time $\max\{r_y^{max}, A_1^{max}(v_p)\}$ (hence case 1 of Eq. (5.13)).

If $J_{y,z}$ is not the first segment of job J_y, then, because $J_{y,z} \in \mathcal{R}^P$, the predecessor of $J_{y,z}$ must have started to execute. Hence, J_y is already certainly released, and $LRT_{y,1}(v_p)$ is only bounded by the time at which the core claimed by the segment of J_y that precedes $J_{y,z}$ becomes available to execute $J_{y,z}$. That time is upper-bounded by $Cl_y^{max}(v_p)$. Therefore, segment $J_{y,z}$ will have certainly requested its resource by time $Cl_y^{max}(v_p)$ (hence case 2 of Eq. (5.13)). □

Now that the *earliest* time $ERT_{i,j}(v_p)$ at which $J_{i,j}$ may request its resource, and the *latest* time $LRT_{y,z}(v_p)$ at which another segment $J_{y,z}$ has certainly requested its resource have been bounded, we can prove the following condition for $J_{i,j}$ to possibly be the next segment to start executing in state v_p.

Theorem 2. *In a system that uses FIFO-ordered spin locks, a segment $J_{i,j}$ is a direct successor of v_p only if $EST_{i,j}(v_p) < \infty$, Condition* (5.11) *holds and*

$$\forall J_{y,z} \in \mathcal{R}^P \text{ s.t. } \eta_{y,z} = \eta_{i,j} \neq \emptyset, ERT_{i,j}(v_p) \leq LRT_{y,z}(v_p). \tag{5.14}$$

Proof. The $EST_{i,j}(v_p) < \infty$ and Condition (5.11) parts were already proven in Theorem 1. Therefore, we focus on proving Condition (5.14). From Lemma 9, we know that $ERT_{i,j}(v_p)$ is the earliest time at which a job $J_{i,j}$ can request its shared resource. Similarly, from Lemma 10, we know that the latest time at which a different segment $J_{y,z}$ requests its own resource is given by $LRT_{y,z}(v_p)$. Therefore, if $J_{i,j}$ and $J_{y,z}$ request the same resource ℓ_x and $ERT_{i,j}(v_p) > LRT_{y,z}(v_p)$, $J_{y,z}$ must certainly be in front of $J_{i,j}$ in the FIFO queue regulating access to ℓ_x. In this case, $J_{y,z}$ will certainly execute before $J_{i,j}$ and $J_{i,j}$ cannot be a direct successor of v_p. □

5.4.6 Earliest and Latest Finish Times

Since segments execute non-preemptively, if $J_{i,j}$ is the segment dispatched in system state v_p, then its earliest finish time ($EFT_{i,j}(v_p)$) and latest finish time ($LFT_{i,j}(v_p)$) are:

$$EFT_{i,j}(v_p) = EST_{i,j}(v_p) + C_{i,j}^{min}(v_p), \tag{5.15}$$

$$LFT_{i,j}(v_p) = LST_{i,j}(v_p) + C_{i,j}^{max}(v_p). \tag{5.16}$$

5.4.7 Creating a New State

As discussed in Section 5.3, we expand the SAG for every segment $J_{i,j}$ that is eligible according to Theorem 1 or 2 (depending on whether priority-ordered or FIFO-ordered spin locks are used, respectively). For each such segment, we

Algorithm 2: Create a new state v'_p by executing job segment $J_{i,j}$ after v_p.

```
1  if J_{i,j} is the first segment of J_i then
2      if J_{i,j} is not the last segment of J_i then
           // J_i claims a core
3          Create the interval Cl_i = [EFT_{i,j}(v_p), LFT_{i,j}(v_p)];
4          C(v'_p) ← C(v_p) ∪ Cl_i;
5      end
6  else
7      if J_{i,j} is the last segment of J_i then
           // the core claimed by J_i is released
8          C(v'_p) ← C(v_p) \ Cl_i;
9      else
           // update the core claimed by J_i
10         Cl_i = [EFT_{i,j}(v_p), LFT_{i,j}(v_p)];
11         C(v'_p) ← C(v_p);
12     end
13 end
14 Update C using Eqs. (5.17) and (5.18);
   // Update the shared resource availability
15 if η_{i,j} ≠ ∅ then
16     Let ℓ_x = η_{i,j};
17     SR_x^{min}(v'_p) = EST_{i,j}(v_p) + L_{i,j}^{min};
18     SR_x^{max}(v'_p) = LST_{i,j}(v_p) + L_{i,j}^{max};
19 end
   // Update the free cores availability intervals
20 Initialize PA and CA using Lemma 12 ;
21 Sort PA and CA in non-decreasing order;
22 ∀1 ≤ x ≤ m − |C(v'_p)|,  A_x(v'_p) ← [PA_x, CA_x];
```

create a new node v'_p that represents the new state after dispatching $J_{i,j}$, as described in Algorithm 2.

First, depending on whether $J_{i,j}$ is the first, last or an intermediate segment of job J_i, it will claim a core, release the core it previously claimed or keep executing on its claimed core, respectively. Therefore, in lines 1–13, Algorithm 2 updates the set of claimed cores $\mathcal{C}$ by either adding, removing or updating the claimed-core availability interval associated with job J_i. If a core is claimed or was claimed, then its availability interval is set to the finish time interval $[EFT_{i,j}, LFT_{i,j}]$ of the segment $J_{i,j}$ that starts to execute on it (Lines 3 and 10).

For all other claimed cores in $\mathcal{C}$, their availability interval is updated according to Equations (5.17) and (5.18) below.

Lemma 11. *If the first segment executed in system state v_p starts executing no later than $EST_{i,j}(v_p)$, then*

$$Cl_y^{min}(v'_p) = \max\{EST_{i,j}(v_p), Cl_y^{min}(v_p)\} \tag{5.17}$$

$$Cl_y^{max}(v'_p) = \max\{EST_{i,j}(v_p), Cl_y^{max}(v_p)\} \tag{5.18}$$

Proof. Since *the first* segment that starts to execute in system state v_p does so no later than $EST_{i,j}(v_p)$, the earliest time at which the next segment may start to execute is not before $EST_{i,j}(v_p)$. Thus, the cores are not available to execute other segments before $EST_{i,j}(v_p)$. This proves the lemma. □

Thereafter, the availability interval of the resource accessed by $J_{i,j}$ (if any) is updated to align with the end of $J_{i,j}$'s critical section, i.e., $L_{i,j}^{min}$ time units after it started executing at the earliest, and $L_{i,j}^{max}$ time units after it started at the latest.

Finally, Algorithm 2 uses Lemma 12 to create two sets PA and CA that store the times at which each free core becomes possibly and certainly available after $J_{i,j}$ started to execute (Line 20). Then, as discussed in Section 5.2, the free cores availability intervals can be computed by sorting the sets PA and CA in a non-decreasing order and picking the x^{th} element of the sorted set PA (CA, respectively) as the lower bound (upper bound, respectively) on the availability interval A_x (Line 22).

Lemma 12. *The times at which each free core becomes possibly and certainly available in v'_p are contained in the sets PA and CA and are respectively, computed as follows.*

$$PA = \begin{cases} \left\{ \max\{EST_{i,j}, A_x^{min}\} \mid 2 \leq x \leq m - |\mathcal{C}| \right\} & \text{if } j = 1 \neq n_i \\ \left\{ \max\{EST_{i,j}, A_x^{min}\} \mid 2 \leq x \leq m - |\mathcal{C}| \right\} \cup \left\{ EFT_{i,n_i} \right\} & \text{if } j = 1 = n_i \\ \left\{ \max\{EST_{i,j}, A_x^{min}\} \mid 1 \leq x \leq m - |\mathcal{C}| \right\} & \text{if } 1 < j < n_i \\ \left\{ \max\{EST_{i,j}, A_x^{min}\} \mid 1 \leq x \leq m - |\mathcal{C}| \right\} \cup \left\{ EFT_{i,n_i} \right\} & \text{if } j = n_i > 1 \end{cases} \tag{5.19}$$

$$CA = \begin{cases} \left\{ \max\{EST_{i,j}, A_x^{max}\} \mid 2 \leq x \leq m - |\mathcal{C}| \right\} & \text{if } j = 1 \neq n_i \\ \left\{ \max\{EST_{i,j}, A_x^{max}\} \mid 2 \leq x \leq m - |\mathcal{C}| \right\} \cup \left\{ LFT_{i,n_i} \right\} & \text{if } j = 1 = n_i \\ \left\{ \max\{EST_{i,j}, A_x^{max}\} \mid 1 \leq x \leq m - |\mathcal{C}| \right\} & \text{if } 1 < j < n_i \\ \left\{ \max\{EST_{i,j}, A_x^{max}\} \mid 1 \leq x \leq m - |\mathcal{C}| \right\} \cup \left\{ LFT_{i,n_i} \right\} & \text{if } j = n_i > 1 \end{cases} \tag{5.20}$$

Proof. We use the following facts.

Fact 1. As already proven for Lemma 11, because $EST_{i,j}(v_p)$ is the earliest time at which $J_{i,j}$ starts to execute in system state v_p, no segment dispatched after $J_{i,j}$ may start to execute prior to $EST_{i,j}(v_p)$. Therefore, no core may be available to execute new jobs before $EST_{i,j}(v_p)$. Thus, $EST_{i,j}(v_p)$ is a lower bound on the availability time of any free core in v'_p.

Fact 2. If $J_{i,j}$ is not the first segment of job J_i (i.e., $j > 1$), then it already has a claimed core and does not execute on a free core. Thus, the availability of each free core in v_p remains the same in v'_p. In combination with Fact 1, we get $PA \supseteq \left\{ \max\{EST_{i,j}(v_p), A_x^{min}(v_p)\} \mid 1 \leq x \leq m - |\mathcal{C}(v_p)| \right\}$ and $CA \supseteq \left\{ \max\{EST_{i,j}(v_p), A_x^{max}(v_p)\} \mid 1 \leq x \leq m - |\mathcal{C}(v_p)| \right\}$.

Fact 3. Because we assume a work-conserving scheduler, if $J_{i,j}$ is the first segment of J_i (i.e., $j = 1$), it will start executing on the first available free core in v_p. All the other cores will thus keep the same availability interval. Therefore, in combination with Fact 1, we get that $PA \supseteq \left\{ \max\{EST_{i,j}(v_p), A_x^{min}(v_p)\} \mid 2 \leq \right.$

$x \leq m - |\mathcal{C}(v_p)|\}$ and $CA \supseteq \{\max\{EST_{i,j}(v_p), A_x^{max}(v_p)\} \mid 2 \leq x \leq m - |\mathcal{C}(v_p)|\}$.

Fact 4. If $J_{i,j}$ is not the first or last segment of job J_i (i.e., $1 < j < n_i$), then it does *not* release the core claimed by J_i. Thus, the set of free core in v'_p is the same as in v_p.

Fact 5. If $J_{i,j}$ is the last segment of J_i (i.e., $j = n_i$), then it releases the core claimed by J_i at the end of its execution. Thus, the core that was claimed by J_i in v_p becomes free for other jobs to execute on in v'_p. That released core is available at the earliest at the EFT of $J_{i,j}$, and at the latest at the LFT of $J_{i,j}$. Hence, $PA \supseteq EFT_{i,j}(v_p)$ and $CA \supseteq LFT_{i,j}(v_p)$.

Fact 3 proves the first case in the definitions of PA and CA. The combination of Facts 3 and 5 prove the second cases. Fact 4 proves the third cases, and the combination of Facts 2 and 5 proves the fourth cases. □

5.5 Merge Phase

In order to delay a potential state space explosion by considering every execution scenario on a different path of the SAG, we introduce Rule 1 that merges two nodes of the graph into a single node that covers all states covered by the initial two nodes. This slows down the growth of the SAG (in terms of the number of nodes) while maintaining soundness.

Rule 1 (Merge rule)**.** *Two nodes v_p and v_q are merged if $\mathcal{J}^P = \mathcal{J}^Q$ and $\forall x,\ 1 \leq x \leq m - |\mathcal{C}|,\ A_x(v_p) \cap A_x(v_q) \neq \emptyset$.*

When two states v_p and v_q are merged into v_z, each free-core availability interval $A_x(v_z)$, claimed core availability interval $Cl_x(v_z)$ and shared resource availability interval $SR_x(v_z)$ in the merged state v_z is computed so as to fully cover the intervals of the initial states v_p and v_q.

$$A_x(v_z) = [\min\{A_x^{min}(v_p),\ A_x^{min}(v_q)\},\quad \max\{A_x^{max}(v_p), A_x^{max}(v_q)\}] \tag{5.21}$$

$$Cl_x(v_z) = [\min\{Cl_x^{min}(v_p),\ Cl_x^{min}(v_q)\},\quad \max\{Cl_x^{max}(v_p), Cl_x^{max}(v_q)\}] \tag{5.22}$$

$$SR_x(v_z) = [\min\{SR_x^{min}(v_p),\ SR_x^{min}(v_q)\},\quad \max\{SR_x^{max}(v_p), SR_x^{max}(v_q)\}] \tag{5.23}$$

We now prove that a merge maintains soundness.

Lemma 13. *For two states v_p and v_q merged according to Rule 1, the set of claimed cores in v_p and v_q are assigned to the same job segments.*

Proof. Every job that has started and did not yet finish its execution until reaching state v_p has a claimed core in v_p. Since by Rule 1, the set of job segments in the path to v_p and v_q are identical (i.e., $\mathcal{J}^P = \mathcal{J}^Q$), all jobs that claimed a core in v_p must also have a claimed core in v_q, and all jobs that claimed a core in v_q must also have claimed a core in v_p. □

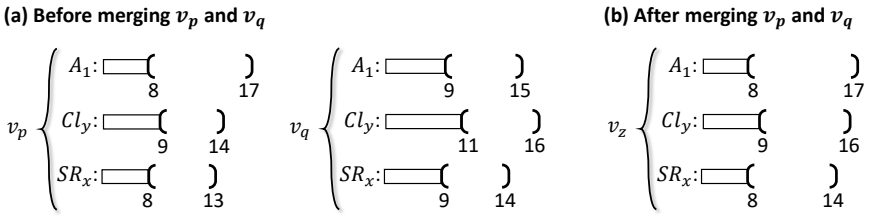


Figure 5.6: **States v_p and v_q before and after merging.**

Lemma 14. *For two states v_p and v_q merged according to Rule 1, the number of free-core availability intervals are the same in v_p and v_q.*

Proof. Lemma 13 proves that the claimed cores in v_p and v_q are assigned to the same job segments. Thus, the number of claimed cores must be the same in v_p and v_q. Hence, the number of free cores must also be the same since the sum of claimed and free cores is always equal to m. It results that we have as many free-core availability intervals in v_p as in v_q. □

Theorem 3. *Merging two states v_p and v_q according to Rule 1 and Equations 5.21, 5.22 and 5.23 is safe, i.e., it does not remove any potentially reachable system state from the graph.*

Proof. Rule 1 forces that the sets of segments that have started executing on the path to v_p and v_q are identical for v_p and v_q. Hence, the set of segments that still need to execute in the merged state v_z is the same as in v_p and v_q. According to Lemma 13 and 14, the set of claimed cores and the number of free cores in state v_p and v_q are the same, and are thus the same in the merged state too. Since the availability interval of shared resources and free and claimed cores in the merged state v_z are the union of those in v_p and v_q, any possible combination of a given number of cores and set of resources becoming available at a given time that is possible in either state v_p or v_q is also possible in the merged state v_z. Hence, all sequences of segment executions that may follow from v_p and v_q are also possible in v_z and the set of all system states reachable from v_z includes every state that is reachable from the original states v_p and v_q. □

Example 5. Figure 5.6 shows two states (v_p and v_q) before and after merging them into state v_z according to Theorem 3. In v_p and v_q, we consider two cores where one is free for any new job segment to execute on and the other is claimed by a job J_y. Furthermore, we consider a shared resource ℓ_x in the system. Figure 5.6 (b) shows that in v_z the availabilities for the free core, the claimed core and the shared resource cover all of their respective intervals found in the original states v_p and v_q. Therefore, it can be visualized that every possible scenario in either of the original states is also possible in the merged state meaning that we do not discard any scenario and that the merge is safe.

5.6 Correctness

We now put all the pieces together and establish soundness.

Theorem 4. *For any execution scenario such that segment $J_{i,j}$ completes at t, there is a path $\langle v_1, ..., v_p, v'_p \rangle$ in the SAG such that $J_{i,j}$ is the label of the edge connecting v_p to v'_p and $t \in [EFT_{i,j}(v_p), LFT_{i,j}(v_p)]$.*

Proof. Assume that there is a path $\langle v_1, ..., v_p \rangle$ such that the claim is respected for all segments that started to execute before $J_{i,j}$ in the execution scenario that led $J_{i,j}$ to finish at time t. Furthermore, assume that the availability intervals of state v_p correctly bound the actual availability times of the shared resources, free and claimed cores.

We prove that $t \in [EFT_{i,j}(v_p), LFT_{i,j}(v_p)]$, that a new system state v'_p is created from scheduling $J_{i,j}$ in v_p and that the availability intervals of state v'_p correctly bound the actual availability times of the shared resources, free and claimed cores after executing $J_{i,j}$.

Under the inductive assumption stated above, Lemma 1 and Lemma 8 prove that $EST_{i,j}(v_p)$ and $LST_{i,j}(v_p)$ are safe lower- and upper-bounds on the start time of $J_{i,j}$, respectively. Because segments execute non-preemptively, Equations (5.15) and (5.16) are thus safe lower- and upper-bounds on t (i.e., we proved that $t \in [EFT_{i,j}(v_p), LFT_{i,j}(v_p)]$). Further, by the inductive assumption, the condition of Theorem 1 or Theorem 2 (depending on whether the systems uses priority-ordered or FIFO-ordered spin locks, respectively) must hold for $J_{i,j}$ and Line 7 of Alg. 1 ensures that the graph is expanded with a new node v'_p. Then, by Lemmas 11 and 12 and the discussion of Alg. 2, the availability intervals of v'_p correctly bound the actual availability of shared resources and cores after executing $J_{i,j}$. Therefore, the inductive assumption is respected for v'_p. Finally, according to Theorem 3, potentially merging v'_p with another node (lines 14 to 18 of Alg. 1) maintains the validity of the inductive assumption.

Crucially, the inductive assumption (i.e., correct availability intervals) obviously holds for v_1 (in which all cores and shared resources are supposed to be available) and thus follows by induction on all the states created by Alg. 1. □

Chapter 6

Empirical Evaluation

We have conducted a large-scale schedulability study to evaluate the accuracy and runtime and to test for scalability limitations of our proposed analyses. In this chapter, we discuss our experimental setup and workloads (6.1), the implementation and the baselines (6.2) as well as the results (6.3) of the experiments.

6.1 Setup and Workloads

To obtain a large corpus of diverse workloads with varied contention characteristics, we generated task sets as follows. For a given number of cores $m \in \{2, 4\}$, we generated $n \in \{m+1, 2m, 3m, 5m\}$ periodic tasks that shared $n^r \in \{2m, 5m\}$ resources. Each task was configured to have a number of critical sections chosen uniformly at random from $\{0, 1, \ldots, n^{cs}\}$, for $n^{cs} \in \{5, 15\}$. The length of each critical section was drawn uniformly at random from either $[1\mu s, 15\mu s]$, $[10\mu s, 50\mu s]$, or $[50\mu s, 150\mu s]$ (*short*, *intermediate*, or *long* critical sections, respectively). Additionally, to obtain Fig. 6.1b, task sets for a specific setup with $m = 6$, $n = 12$, $n^{cs} = 5$ and $n^r = 12$ were generated as well.

For each considered combination of m, n, n^r, n^{cs}, and critical section lengths, we varied the total utilization U from 5% to 95% in steps of 5, and for each U, we generated 250 task sets, for a total of more than 450,000 task sets.

For a given U and n, we generated n per-task utilizations u_1, u_2, u_3, ... that sum to U using Emberson et al.'s *RandFixedSum* method [21]. To reflect that non-preemptive scheduling is used in practice only for workloads with short jobs, for each task, an initial cost value C_i' was drawn from a normal distribution with mean $3ms$ and standard deviation $1.5ms$ (values of less than $10\mu s$ were redrawn). To avoid unrealistic periods, we then selected the task period $T_i \in \{5, 8, 10, 12, 15, 20, 25, 30, 50, 75, 100, 150, 200, 250, 300, 500\}ms$ closest to C_i'/u_i and set the task's WCET to $u_i \cdot T_i$ (but no less than the sum of maximum critical section lengths).

The total WCET was randomly distributed across the task's segments $C_{i,1}^{max}$, $C_{i,2}^{max}, \ldots$ while respecting the corresponding maximum critical-section lengths (if any). Each segment was assigned a BCET $C_{i,j}^{min}$ by drawing a value uniformly at random from $[0.1C_{i,j}^{max}, 0.5C_{i,j}^{max}]$. Each critical section was assigned a minimum length of $0\mu s$ with probability 0.25, and otherwise chosen at random as $L_{i,j}^{min} \in [0.1L_{i,j}^{max}, 0.5L_{i,j}^{max}]$.

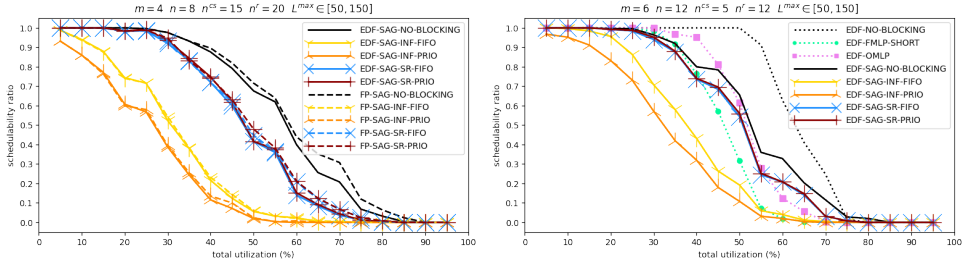
Finally, tasks were assigned unique priorities using the DkC heuristic for fixed-priority scheduling [17]. Afterwards, as a necessary test, one hyperperiod of the task set was simulated assuming that each job executes for its WCET (without any blocking). If the simulation already exhibited a deadline miss, then the task set was discarded to avoid generating task sets that are obviously infeasible under non-preemptive scheduling.

For our experiments, we include an investigation of the runtime of our proposed solution, however our empirical evaluation mainly focuses on the performance metric called the *schedulability ratio*. It defines the ratio of schedulable task sets, i.e., task sets where all jobs meet their timing requirements (no deadline miss), to the total number of generated task sets.

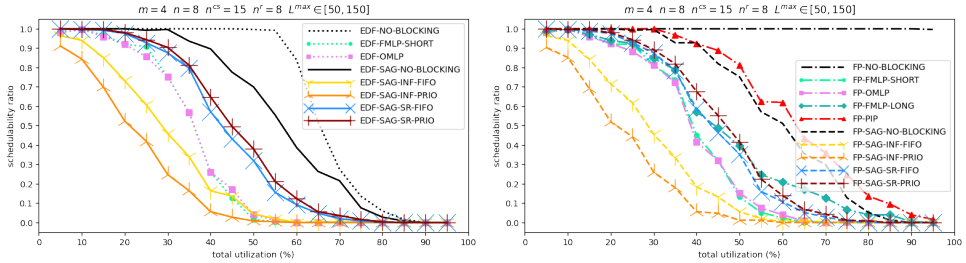
6.2 Implementation and Baselines

We implemented the proposed analysis, which we denote as **{EDF, FP}-SAG-SR-{FIFO, PRIO}** in the following (under global non-preemptive EDF and FP scheduling, respectively), considering either the usage of FIFO-ordered or priority-ordered spin locks.

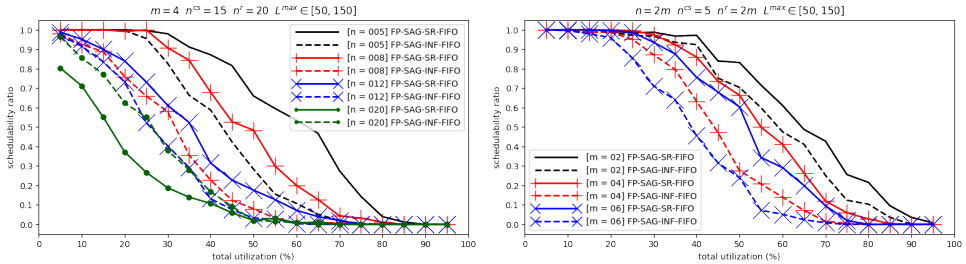
As there exist no directly comparable analysis to compare against (the proposed solution is the first of its kind), we further considered the following loosely related baselines to provide context: 1. **{EDF, FP}-SAG-NO-BLOCKING**: Nasri et al.'s analysis [33] *without any blocking* as provided in the open-source SchedCAT library [1]. This analysis is not sound in the presence of shared resources; it merely serves to indicate an upper bound on attainable schedulability. 2. **{EDF, FP}-SAG-INF-{FIFO, PRIO}**: Nasri et al.'s analysis [33] with an *inflation-based blocking analysis* where each job's WCET is increased prior to response-time analysis to account for possible blocking based on the holistic blocking approach [8, Ch. 5.4]. This analysis is sound, but structurally pessimistic (not scenario-aware), and thus provides a simple lower bound on schedulability that the proposed analysis exceeds. 3. **EDF-NO-BLOCKING**: the schedulability tests for *preemptive* global EDF provided in SchedCAT *without any blocking*, as an upper bound on schedulability under preemptive EDF scheduling. 4. **FP-NO-BLOCKING**: similarly, the schedulability tests for *preemptive* global FP scheduling without any blocking. 5. **{EDF, FP}-FMLP-SHORT**: inflation-based holistic blocking analysis [8] of non–preemptive FIFO-ordered spin locks (i.e., "FMLP for short resources" [7]), as provided in SchedCAT. 6. **{EDF, FP}-OMLP**: inflation-based holistic blocking analysis of the *suspension-based* global OMLP locking protocol [12], as provided in SchedCAT. 7. **FP-FMLP-LONG**: Yang et al.'s analysis [41] of the suspension-based "FMLP for long resources" [7]. 8. **FP-PIP**: similarly, Yang et al.'s analysis [41] of the suspension-based PIP [19,38].



(a) **comparison of SAG analyses for** $m = 4$ (b) **comparison of EDF analyses for** $m = 6$



(c) **comparison of EDF analyses for** $m = 4$ (d) **comparison of FP analyses for** $m = 4$



(e) **schedulability for varying** n (f) **schedulability for varying** m

Figure 6.1 **Schedulability results.**

6.3 Results

In total, our experimental setup considered 18 individual analyses for over 95 different scenarios (i.e., parameter combinations). A representative selection of our results is shown in Figs. 6.1 and 6.2 with the specific parameter choices given in each plot. To avoid clutter, only a subset of the 18 curves is shown in each graph.

Figure 6.1a shows a comparison of the various SAG analyses for $m = 4$. First, there is little difference with regard to whether jobs are prioritized according to fixed task-level priorities [17] or by job-level EDF priorities. More importantly, however, the results show a large gain in schedulability relative to the inflation-based baseline. We see this trend for both, FIFO- and priority-ordered spin locks. For example, while FP-SAG-INF-FIFO only deems about 15% of the task sets to be schedulable at 45% total utilization, although FP-SAG-SR-FIFO shows that more than 60% of the tested task sets are actually schedulable. Furthermore, we see that slightly higher gains are achieved when using priority-ordered spin locks, as FP-SAG-INF-PRIO detects even less schedulable task sets

($< 10\%$) than FP-SAG-INF-FIFO at 45% utilization, while FP-SAG-SR-PRIO detects slightly more than FP-SAG-SR-FIFO. In this experiment, an average gain ranging from $18\% - 25\%$ in schedulability ratio has been achieved where FP-SAG-SR-PRIO provided the largest benefits. Overall, the proposed analysis closes *more than half of the gap* between the inflation-based baseline and the upper bound on attainable schedulability represented by {EDF, FP}-SAG-NO-BLOCKING, which highlights a significant reduction in pessimism.

Figure 6.1b focuses on EDF scheduling and shows similar trends for $m = 6$, with EDF-SAG-SR-{FIFO, PRIO} attaining much higher schedulability than EDF-SAG-INF-{FIFO, PRIO}. Furthermore, a comparison with EDF-OMLP and EDF-FMLP-SHORT shows the proposed analysis to be competitive with the state of the art for *preemptive* systems (for the considered workloads), especially considering that the EDF-NO-BLOCKING baseline reveals that preemptive scheduling has a slight advantage in this scenario. Note that our evaluation does not reflect any differences in scheduling and runtime overheads, which can be expected to be (much) lower under non-preemptive scheduling. Figure 6.1c shows largely identical trends for $m = 4$.

Figure 6.1d shows an analogous comparison for FP scheduling; here preemptive scheduling has a significant advantage and Yang et al.'s suspension-aware analysis [41] of the PIP stands out as particularly effective. However, note again that this is a somewhat lopsided comparison due to workload differences (preemptive vs. non-preemptive) and since we are discounting lower spin-lock overheads.

Figure 6.1e shows the effect of increasing the number of tasks. Generally, schedulability drops as n increases, which is unsurprising as an increase in tasks is correlated with an increase in contention. Nonetheless, significant accuracy gains are apparent compared to the inflation-based baseline. For example, for $n = 5$ and $U = 60\%$, FP-SAG-SR-FIFO shows more than 50% of the workloads to be schedulable in this experiment, whereas FP-SAG-INF-FIFO deems less than 20% of the workload to be schedulable.

In Fig. 6.1f, we look at the effect of varying the number of cores. Here we clearly see that, for all investigated values of m, the proposed analysis performs better than the inflation-based baseline. Schedulability drops with increasing m since an increase in parallelism corresponds to an increase in contention (since $n = 2m$ in the shown scenarios).

Finally, Figs. 6.2a to 6.2d show the average runtime (in seconds) of our analysis. For these results, it has to be noted that due to the limited access to servers, experiments were conducted on different machines. Consequently, individual points are not comparable, which is why we focus on the general trend of the graphs. We use Fig. 6.2a as our default and vary different parameters to investigate the general effect on the runtime. First, we inspect the effect of considering long $[50\mu s, 150\mu s]$ instead of short $[1\mu s, 15\mu s]$ critical section by comparing Fig. 6.2b to our default in Fig. 6.2a. We can see that there is hardly an effect on the runtime. Both plots stay within the range of about $0 - 0.2$ seconds. Next, we compare Figs. 6.2a and 6.2c to see how the runtime behaves when increasing the number of tasks from 5 to 8. Here we observe a more prominent change in the runtime, where the average runtime grew above 1 second. This is an expected behavior, as an increase in the number of tasks allows for more contention and essentially more execution scenarios that need to be analysed. More execution scenarios mean that our analysis will create

and investigate more states, which results in a longer duration of the analysis and hence increased runtime. Lastly and most interestingly, we have analyzed the effect of increasing the maximum number of critical sections per task from 5 to 15 by comparing Figs. 6.2a and 6.2d. We notice a significant increase in the average runtime. While Fig. 6.2a shows that on average, experiments have finished in less than a second, we can see in Fig. 6.2d that experiments have taken far longer and almost reaching up to 40 seconds in average. Increasing the number of per task critical sections while keeping the number of shared resources constant means that many tasks compete for the same resources. This increased contention leads to a significant rise in the number of execution scenarios as we now consider all possible combinations of access orderings for each critical section of a task and build full paths for each of them. Essentially this leads to a remarkable increase in the number of investigated states and hence leads to a rise in the runtime. Furthermore, to investigate the collected runtime data from a different angle, we have plotted the 95th and 98th percentiles for EDF-SAG-SR-PRIO and EDF-SAG-SR-FIFO in Figs. 6.2e to 6.2h. First, Figs. 6.2e and 6.2f show that in fact 95% of the experiments finished in less than 100 seconds for EDF-SAG-SR-PRIO and in less than 200 seconds for EDF-SAG-SR-FIFO. They also show that the FIFO variant is more costly in terms of runtime and experiences more timeouts (here shown by the timeout limit 7200 seconds) compared to the priority variant. The 98th percentiles given in Figs. 6.2e and 6.2f show that especially around 30% to 50% utilization both variants experience an increase in the runtime. This is also a trend that we can observe in the plots shown in Figs. 6.2a to 6.2d and therefore, in the following we will discuss the general trends observed regarding all runtime experiments.

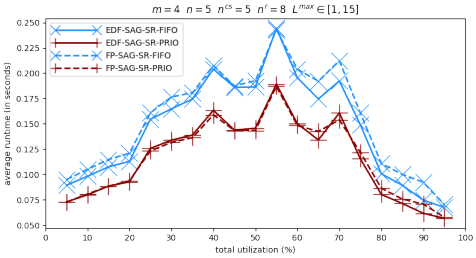
First, the average runtime and the scatter plots portraying the 95th and 98th percentiles revealed that our analysis considering priority-ordered spin locks performed better in our runtime experiments than the analysis using FIFO-ordered spin locks. This is an unsurprising result, since the analysis for FIFO-ordered spin locks is more extensive and has to compute additional parameters such as the earliest and latest request time. Finally, for all the runtime-related experiments, we see a another general trend. As the utilization increases and roughly approaches the middle range (30% - 50%), we see an increase in the average runtime and from 50% onwards, we observe a decline. This behavior follows from the fact that with increased utilization, there is an increase in contention. Increased contention means that it is more difficult to schedule a job set and that we have a higher chance of interleaving intervals. An increase in interleaving intervals results in increased branching and the creation of multiple states since we have to consider all possible execution scenarios for the analysis to be sound. This is directly coupled with an increase in average runtime. However, if we increase the total utilization even further, the considered job set becomes even more difficult to schedule and the likelihood of a deadline miss increases. Since in each state of the exploration, our analysis checks if timing requirements are respected, a deadline miss would instantly stop the exploration and the job set would be deemed unschedulable. This also means that the runtime of the analysis declines with a further increase in utilization since more job sets are found to be unschedulable at an early stage of the analysis.

Finally, our evaluation also revealed some limitations. First, note how in Fig. 6.1e the results for FP-SAG-SR-FIFO for $n = 20$ tasks are worse than for the baseline FP-SAG-INF-FIFO, but *not* for lower task counts. This is a

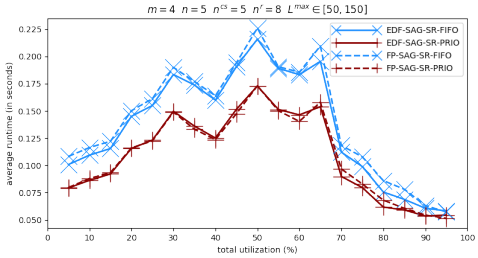
result of the increased runtime of the proposed analysis. In our experiments, we configured a timeout per job set. As the runtime of the analysis increases with n, especially FP-SAG-SR-FIFO started to exhibit a significant number of timeouts for $n = 20$. Scalability limitations also meant that while it is certainly possible to analyze a given system and even perform experiments for a chosen range of parameters when $m > 4$ (e.g., Fig. 6.1b shows results for $m = 6$), the general increase in runtime made an evaluation across the entire parameter space (i.e., *hundreds of thousands of workloads*, recall Section 6.1) impractical for $m \geq 6$.

Surprisingly, we also found some rare cases where EDF, FP-SAG-SR-{FIFO, PRIO} perform (slightly) worse than the corresponding EDF, FP-SAG-INF-{FIFO, PRIO} analyses even when there is no timeout. Further investigations led us to discover a source of pessimism related to the way shared-resource availabilities are encoded. Our current state abstraction assumes that the resource remains locked (and thus unavailable) during the whole time between SR^{min} and SR^{max} in the worst case. Although accurate in most cases, this interval may become very long in comparison to the actual critical section length in some cases. Therefore, there are rare situations where even an inflation based test may perform better. Such occasional instances can be easily handled by running both analyses and retaining the better result.

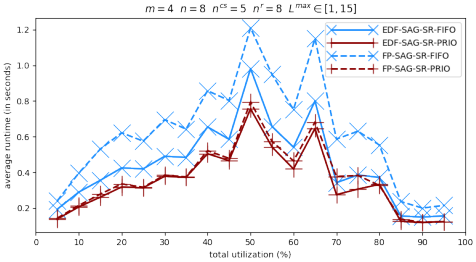
Nonetheless, we overall conclude that *almost* always the proposed analysis shows substantial accuracy gains in comparison to the state of the art and, *for the considered workloads*, is competitive with (or even superior to) solutions designed for preemptive systems.



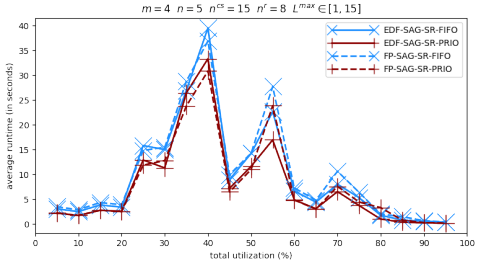
(a) **runtime for $n = 5$, $n^{cs} = 5$, $L^{max} \in [1\mu s, 15\mu s]$**



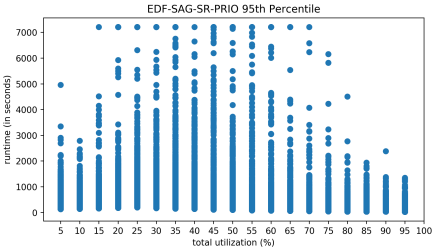
(b) **runtime for $n = 5$, $n^{cs} = 15$, $L^{max} \in [50\mu s, 150\mu s]$**



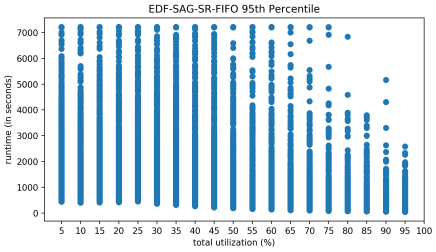
(c) **runtime for $n = 8$, $n^{cs} = 5$, $L^{max} \in [1\mu s, 15\mu s]$**



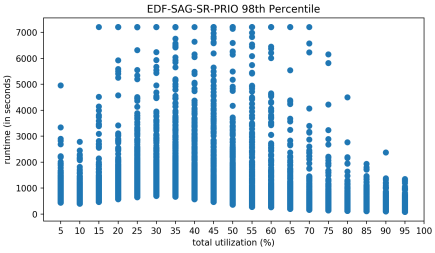
(d) **runtime for $n = 8$, $n^{cs} = 15$, $L^{max} \in [1\mu s, 15\mu s]$**



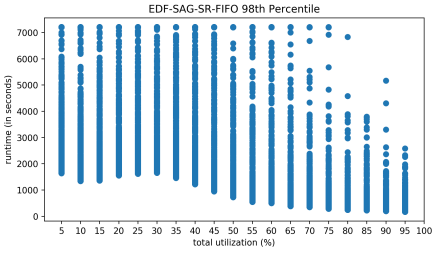
(e) **runtime 95th percentile PRIO**



(f) **runtime 95th percentile FIFO**



(g) **runtime 98th percentile PRIO**



(h) **runtime 98th percentile FIFO**

Figure 6.2 **Runtime results.**

Chapter 7

Extensions of the Work

7.1 Partial Order Reduction Techniques

In our proposed analysis, we cover every execution scenario of a job set by building the schedule abstraction graph. For large job sets with many interleaving intervals, this results in the creation of a lot of states. One of the sub-goals of this project is to delay a potential state space explosion and improve scalability, which has already been partially achieved by choosing a smart representation of the workload (Section 4.1) and the system states (Section 5.2), but also by merging similar states (Section 5.5). As an extension, we propose a further reduction in the number of states created, by applying so called partial order reduction techniques, which allow the smart pruning of the schedule abstraction graph while maintaining soundness. The key idea of partial order reduction techniques is that we eliminate redundant branches when building the graph. At this point, a natural question is, when is branching redundant? Branching is redundant if at any given state there exists an eligible job segment whose execution is not impacted by any present or future segment (i.e., any segment that has not started to execute yet), nor does it impact any segment that is not yet in the path. In such a case, the state can be extended by only considering a segment that fulfills these conditions, as it does not discard any possible execution scenario and at the same time prevents the redundant creation of additional states. Therefore, we have designed partial order reduction rules that check for segments that fulfill the stated conditions.

In the following, we give an idea about the designed rules and provide visual examples to support the explanation. The formal proofs of the rules can be found in Appendix A.

7.1.1 Non-Starting Segments

First, we look at a non-starting segments, i.e., an intermediate or finishing segment. In order for a segment $J_{i,j}$ in v_p to be eligible for partial order reduction, no execution scenario should be discarded when extending the state v_p by $J_{i,j}$ without branching. This is the case, if the execution of $J_{i,j}$ is not impacted by nor does it impact any segment that still has to execute. Therefore, the key idea for a non-starting segment to be considered for partial order reduction is, that (i) there is no segment that can start prior to the earliest start time of $J_{i,j}$ in v_p,

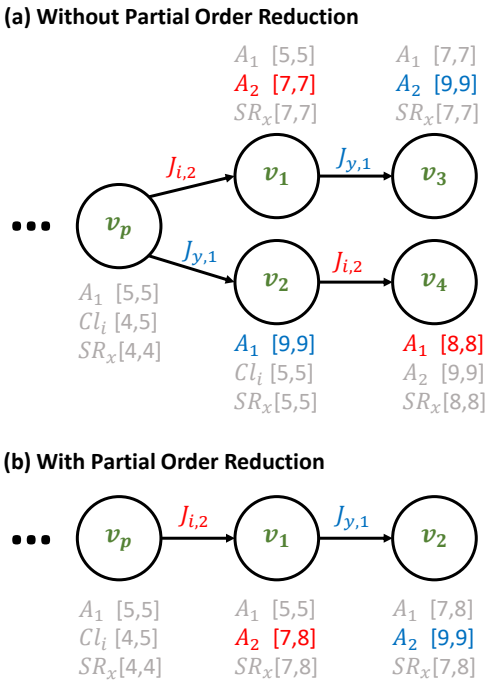


Figure 7.1: **Partial Order Reduction: Non-Starting Segment.** Two schedule-abstraction graphs that display the effect of partial order reduction using a non-starting segment for a system with $m = 2$ and two remaining segments $J_{i,2}$ (finishing segment of job J_i) and $J_{y,1}$ (only segment of job J_y released at $r_y \in [5, 5]$). Only $J_{i,2}$ requests the resource ℓ_x with a critical section length $L_{i,2} \in [3, 3]$. The execution times of the segments are $C_{i,2} \in [3, 3]$ and $C_{y,1} \in [4, 4]$. We assume that J_y has a higher priority than J_i, i.e., $p_y < p_i$.

and (ii) there is no segment that can potentially request the same resource as $J_{i,j}$ prior to the latest finish time of $J_{i,j}$. Hence, we write the following claims, for which proofs are given in Appendix A.1.

Lemma 15. *The start time of a non-starting segment $J_{i,j} \in \mathcal{R}^P$ in v_p is not impacted by any segment $J_{y,z} \notin \mathcal{J}^{\mathcal{P}}$ if*

$$LST_{i,j}(v_p) < ERT_{y,z} \mid \eta_{i,j} \cap \eta_{y,z} \neq \emptyset \tag{7.1}$$

Lemma 16. *The execution of a non-starting segment $J_{i,j} \in \mathcal{R}^P$ in v_p does not impact the start time of any segment that requests the same resource as $J_{i,j}$, i.e., $J_{y,z} \notin \mathcal{J}^{\mathcal{P}} \wedge \eta_{i,j} \cap \eta_{y,z} \neq \emptyset$ if*

$$LFT_{i,j}(v_p) < ERT_{y,z} \tag{7.2}$$

Lemma 17. *A state v_p for which there exists a non-starting segment $J_{i,j}$ that can potentially start to execute at $EST(v_p) = EST_{i,j}$ and for which Eqs. (7.1) and (7.2) hold, can be expanded by one edge labeled $J_{i,j}$ without discarding any execution scenario.*

Example 6. Fig. 7.1 shows two schedule-abstraction graphs, one with and one without applying partial order reduction in a system with two cores ($m = 2$) and two remaining segments ($J_{i,2}$ and $J_{y,1}$). In Fig. 7.1 (a), we see that state v_p can be expanded considering both segments, which leads to the creation of multiple states. However in terms of the actual schedule, both segments do not impact each other, because they do not compete for the same core or resources. This means that as long as no scenario is discarded in the graph due to updating core availabilities according to EST, we can decide not to branch in v_p. Hence, we summarize both edges of $J_{i,2}$ in Fig. 7.1 (a) and replace it with one edge in Fig. 7.1 (b). While $J_{i,2}$ leads to availabilities of $[7, 7]$ and $[8, 8]$ in Fig. 7.1 (a), we consider both cases in Fig. 7.1 (b) as $J_{i,2}$ results in an availability interval of $[7, 8]$. Note that we could not expand v_p with $J_{y,1}$ in Fig. 7.1 (b) as it would discard the scenario (due to core updating according to $EST_{y,1}$) where $J_{i,2}$ starts to execute at time 4.

7.1.2 Starting Segments

For a starting segment $J_{i,j}$ that does not access any resources, we have designed a rule that ensures that by extending a state v_p only considering $J_{i,j}$, no execution scenario is discarded. This presumes that the execution of $J_{i,j}$ is not impacted by nor does it impact any segment that still has to execute. Since $J_{i,j}$ is a starting segment that does not require a resource, it only competes and affects (or can be affected) by other starting segments. The key idea is that a starting segment can be considered for partial order reduction if (i) there is no segment that can start prior to the earliest start time of $J_{i,j}$ in v_p and (ii) in every instance of $J_{i,j}$'s release interval ($[r_i^{min}, r_i^{max}]$), the number of k certainly available cores $|A_k^{max}|$ is larger or equal to the number of possibly released starting segments. Hence, we write the following claims, for which formal proofs are given in Appendix A.2.

Lemma 18. *The start time of a starting segment $J_{i,j}$ in v_p that does not access a resource, is not impacted by any present or future segment, if in every instance of $J_{i,j}$'s release interval ($[r_i^{min}, r_i^{max}]$), the number of k certainly available cores $|A_k^{max}|$ is larger or equal to the number of possibly released starting segments.*

Lemma 19. *The start time of any starting segment $J_{y,z}$ that has not yet started to execute, is not impacted by a considered starting segment $J_{i,j}$ in v_p that does not access any resource, if $EST(v_p) = EST_{i,j}$ and in every instance of $J_{i,j}$'s release interval ($[r_i^{min}, r_i^{max}]$), the number of k certainly available cores $|A_k^{max}|$ is larger or equal to the number of possibly released starting segments.*

Lemma 20. *Expanding state v_p in the schedule abstraction graph without branching, considering a starting segment $J_{i,j}$, that does not access a resource, does not discard any execution scenario if, $EST(v_p) = EST_{i,j}$ and in every instance of $J_{i,j}$'s release interval ($[r_i^{min}, r_i^{max}]$), the number of k certainly available cores $|A_k^{max}|$ is larger or equal to the number of possibly released starting segments.*

Example 7. In Fig. 7.2 (a), we see that state v_p can be expanded considering both segments, which leads to the creation of multiple states. Since $J_{i,1}$ and $J_{y,1}$ are both starting segments, they both compete for a free core. However, by the time both segments are potentially released, they will both certainly have a core to execute on meaning that they do not impact each other in any way.

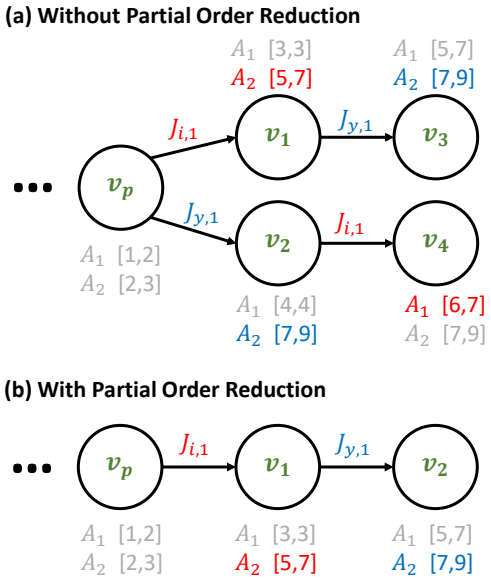


Figure 7.2: **Partial Order Reduction: Starting Segment.**
Two schedule-abstraction graphs that display the effect of partial order reduction using a starting segment for a system with $m = 2$ and two remaining segments $J_{i,1}$ (only segment of job J_i released at $r_i \in [3, 5]$) and $J_{y,1}$ (only segment of job J_y released at $r_y \in [4, 6]$) that do not request a resource. The execution times of the segments are $C_{i,1} \in [2, 2]$ and $C_{y,1} \in [3, 3]$.We assume that J_y has a higher priority than J_i, i.e., $p_y < p_i$.

This means that we can decide not to branch in v_p and instead summarize both edges of $J_{i,1}$ in Fig. 7.2 (a) by replacing it with one edge in 7.2 (b). While $J_{i,1}$ leads to availabilities of $[5, 7]$ and $[6, 7]$ in Fig. 7.1 (a), we consider both cases in Fig. 7.1 (b) as $J_{i,2}$ results in an availability interval of $[5, 7]$. Note that we could not expand v_p with $J_{y,1}$ in Fig. 7.1 (b) as it would discard the scenario (due to $EST_{y,1}$) where $J_{i,1}$ starts to execute at time 3.

7.2 Multi-Unit Resources

As a further addition to this thesis, we have extended the newly proposed analysis, by providing a solution to make it compatible with multi-unit resources and k-exclusion locking protocols.

As the name implies, a multi-unit resource is a resource that can be accessed by more than one task simultaneously. Such a resource consists of k (usually) similar or identical units that can be accessed at most by k tasks at the same time. The resource is protected by a lock with k tokens, meaning that once k tasks access the resource, it becomes unavailable (locked) for any additional job that requests it. This means, that any new job can only be granted access to a multi-unit resource if we have one or more token(s) available. An examples of multi-unit resources could be execution threads in thread-pools, communication channels, I/O buffers or just plain memory space.

The extension is motivated by the fact that it is very common in real-time systems to find, so called, resource pools, which are multiple similar units of a resource. Another motivation is the inclusion of k-exclusion protocols in new technologies such as general-purpose and real-time computation on graphics processing units [20].

In the following, we dive into how the existing analysis has been modified in order to support systems with multi-unit resources handled by k-exclusion locking protocols.

7.2.1 Shared Resource Representation

The access to multi-unit resources could be modeled by accounting for each resource unit specifically and keeping track of when each of the units becomes possibly and certainly available. Building the graph using this representation, however, would lead to an exploration where we consider every possible combination of job accesses to specific resource units. Assuming that a job that requests a resource ℓ_x only requires any one of the k tokens to access the resource, it is not of significant importance to model which particular resource unit has been used. Keeping in mind scalability and the aim of delaying a potential state space explosion as much as possible, we have designed a smarter solution. For each multi-unit resource $\ell_{x,k}$ where k denotes the total number of tokens associated with the resource, we keep track of k intervals $SR_{x,q}(v_p) = [SR^{min}_{x,q}(v_p), SR^{max}_{x,q}(v_p)]$ such that $1 \leq q \leq k$ indicating when one, two, three, ..., k resource units of ℓ_x become possibly and certainly available to be granted to a ready job segment.

Furthermore, in Section 5.2 we have introduced the notions of $SR^{min}_{i,j}(v_p)$ and $SR^{max}_{i,j}(v_p)$ to refer to the availability of the resource accessed by segment $J_{i,j}$. Since we now deal with multi-unit resources, we have to redefine these notions. $SR^{min}_{i,j}(v_p)$ and $SR^{max}_{i,j}(v_p)$ for a job segment $J_{i,j}$ that requests a multi-unit resource $\ell_{x,k}$, now represent the time at which *one* resource unit becomes possibly and certainly available, respectively, i.e., $SR^{min}_{i,j}(v_p) = SR^{min}_{x,1}(v_p)$ and $SR^{max}_{i,j}(v_p) = SR^{max}_{x,1}(v_p)$ for $\ell_{x,k} \in \eta_{i,j}$.

7.2.2 Earliest and Latest Start Time

We have introduced the use of multi-unit resources and we were able to consider the new model by keeping the same equations and redefining $SR^{min}_{i,j}(v_p)$ and $SR^{max}_{i,j}(v_p)$. The lemmas and proofs that ensure a correct computation of the earliest and latest start times (EST and LST) have to be revisited to prove the soundness of the analysis when considering the extension. These proofs represent a modified version of the already introduced proofs in Section 5.4.3 and 5.4.3 and can be found in Appendix B.1. While in our original analysis, a job segment is dependent on the one availability of the resource it requests, in this extension a job segment can start to execute once all resource-independent conditions are met and at least one token of the resource (requested by a considered job segment $J_{i,j}$) is available meaning that at least one resource unit can be used by $J_{i,j}$.

7.2.3 Store and Update Multi-Unit Resource Availability

Since we keep track of when k resource units of a resource $\ell_{x,k}$ become possibly and certainly available in a state, the intervals need to be updated appropriately to be used in the subsequent state(s). Similar to the procedure of the free core availabilities, we create two sets RPA and RCA for each multi-unit resource $\ell_{x,k}$ that store the times at which each resource unit becomes possibly and certainly available after $J_{i,j}$ has started to execute. The times are sorted in non-decreasing order and the q^{th} element of RPA (RCA, respectively) provides the lower bound (upper bound, respectively) on the availability interval $SR_{x,q}$. This is done for each multi-unit resource $\ell_{x,k} \in \boldsymbol{\mathcal{L}}$.

Lemma 21. *The times at which each resource unit of $\ell_{x,k}$ becomes possibly and certainly available in v'_p are contained in the sets RPA_x and RCA_x, respectively and are computed as follows.*

$$RPA_x = \begin{cases} \left\{ \max\{EST_{i,j}, SR^{min}_{x,q}\} \mid 1 \leq q \leq k \right\} & \text{if } \ell_{x,k} \notin \eta_{i,j} \\ \left\{ \max\{EST_{i,j}, SR^{min}_{x,q}\} \mid 2 \leq q \leq k \right\} \cup \left\{ EST_{i,j} + L^{min}_{i,j} \right\} & \text{if } \ell_{x,k} \in \eta_{i,j} \end{cases} \tag{7.3}$$

$$RCA_x = \begin{cases} \left\{ \max\{EST_{i,j}, SR^{max}_{x,q}\} \mid 1 \leq q \leq k \right\} & \text{if } \ell_{x,k} \notin \eta_{i,j} \\ \left\{ \max\{EST_{i,j}, SR^{max}_{x,q}\} \mid 2 \leq q \leq k \right\} \cup \left\{ LST_{i,j} + L^{max}_{i,j} \right\} & \text{if } \ell_{x,k} \in \eta_{i,j} \end{cases} \tag{7.4}$$

Proof. We use the following facts.

Fact 6. As already proven for Lemma 11, because $EST_{i,j}(v_p)$ is the earliest time at which $J_{i,j}$ starts to execute in system state v_p, no segment dispatched after $J_{i,j}$ may start to execute prior to $EST_{i,j}(v_p)$. Therefore, no resource (including $\ell_{x,k}$) may be available for any job before $EST_{i,j}(v_p)$. Thus, $EST_{i,j}(v_p)$ is a lower bound on the availability time of any resource in v'_p.

Fact 7. If $J_{i,j}$ does not access multi-unit resource $\ell_{x,k}$, the resource units are not used by $J_{i,j}$. Therefore, the resource availability $SR_{x,q}$ for each q in v'_p is bounded by the resource availability $SR_{x,q}$ in the previous state v_p. Since, from Fact 6, we also know that $EST_{i,j}(v_p)$ is a lower bound, the resource availability $SR_{x,q}$ for each q in v'_p is given by $\{\max\{EST_{i,j}, SR_{x,q}\} \mid 1 \leq q \leq k$. This proves the first case of Eq. (7.3) and Eq. (7.4).

Fact 8. If $J_{i,j}$ does access multi-unit resource $\ell_{x,k}$, it will access the first available unit (between time $SR^{min}_{x,1}$ and $SR^{max}_{x,1}$). Since the other resource units are not used by $J_{i,j}$, they are updated according to $\{\max\{EST_{i,j}, SR_{x,q}\}$ as explained in Fact 7. On the other hand, the resource unit that has been used for $J_{i,j}$'s execution is updated considering the start time (EST and LST) and the critical section length ($L_{i,j}$) of $J_{i,j}$. The resource unit granted to $J_{i,j}$ is therefore possibly available from time $EST + L^{min}_{i,j}$ and certainly available at time $LST + L^{max}_{i,j}$ in state v'_p. Therefore, considering that job segment $J_{i,j}$ accesses a multi-unit resource $\ell_{x,k}$ and starts executing in system

state v_p, we update the earliest and latest availabilities for each k of $\ell_{x,k}$ according to $\left\{ \max\{EST_{i,j}, SR^{min}_{x,q}\} \mid 2 \leq q \leq k \right\} \cup \left\{ EST_{i,j} + L^{min}_{i,j} \right\}$ and $\left\{ \max\{EST_{i,j}, SR^{max}_{x,q}\} \mid 2 \leq q \leq k \right\} \cup \left\{ LST_{i,j} + L^{max}_{i,j} \right\}$, respectively. This proves the second case of Eq. (7.3) and Eq. (7.4).

□

7.2.4 Merging

Since the representation of a shared resource has changed, the rule for merging shared resource availabilities in our extension also needs to change. In Eq. (5.23) we merge two different availability intervals of a shared resource ℓ_x of state v_p and v_q into one interval in state v_z, that covers both intervals of the initial states. For this multi-unit resource extension, we merge shared resource availabilities considering the interval of each unit belonging to the multi-unit resource $\ell_{x,k}$. This means that in the merged state v_z, every q where $1 \leq q \leq k$ in $\ell_{x,k}$ has an availability interval $SR_{x,q}(v_z)$ that comprises both availabilities $SR_{x,q}(v_p)$ and $SR_{x,q}(v_q)$. Hence we write the modified merge rule as follows.

$$SR_{x,q}(v_z) = [\min\{SR^{min}_{x,q}(v_p),\ SR^{min}_{x,q}(v_q)\},\quad \max\{SR^{max}_{x,q}(v_p), SR^{max}_{x,q}(v_q)\}] \tag{7.5}$$

Chapter 8

Conclusions

8.1 Summary

In this work, we presented a novel worst-case response time (WCRT) analysis for global job-level fixed-priority (JLFP) scheduling and non-preemptive tasks with repeating job-release patterns that share resources protected by FIFO- or priority-ordered spin locks. To the best of our knowledge, it is the first solution to this problem. Furthermore, it extends the family of schedule-abstraction-based analysis to support and model the access to shared resources in a highly accurate manner. Our analysis implicitly explores all possible execution and resource access orderings using a novel system-state abstraction that explicitly models resource contention.

A comparison with inflation-based analyses has shown that our work is substantially less pessimistic. The empirical evaluation has also shown, that for the type of workloads considered in the experiments, our solution comes much closer to the hypothetical blocking-free upper bounds and is competitive even with solutions for preemptive systems. This suggests that our analysis successfully discards many more scenarios that reflect impossible combinations of shared resource access orders, low-priority blocking and/or high-priority interference.

We proposed steps for improving the scalability of the analysis by introducing partial order reduction techniques that aim to smartly discard redundant branches of the schedule abstraction graph. While these techniques improve scalability by reducing the total number of states, this reduction is expected to also improve the performance of the analysis in terms of runtime. Finally, we have extended the work even further by making the analysis compatible with multi-unit resources and k-exclusion locking protocols. This has been achieved by modifying our notions of shared resource availabilities to account for the multiple tokens belonging to a particular resource class.

Finally, we conclude that the positive results of our work virtually across the board provide ample motivation to further improve accuracy and scalability of this promising schedule-abstraction-based approach in future work.

8.2 Future Work

Since the work has been extended to consider partial order reduction, we believe that there is a potential for additional partial order reduction rules that can improve scalability even further.

Furthermore, our investigation, as discussed in Section 6.3, has shown that there exists a source of pessimism related to the encoding of shared resource availabilities in a state. Hence, it would be interesting to investigate even further and find a solution that can reduce or even erase this source of pessimism.

Our line of research, namely schedule-abstraction-based analysis, allows for many interesting future works. Since the work presented in this thesis opens the doors to the explicit analysis of shared resource accesses using schedule-abstraction graphs, an interesting extension for our model would be the consideration of parallel DAG tasks, which are known to be favorable for multi-processor platforms. Such workloads could be considered by our analysis by introducing precedence constraints to our model. Furthermore, another interesting extension to our work could be considering nested locks in the analysis.

Finally, since our analysis focuses on spin-based locks, an interesting future work would be designing a schedule-abstraction-based analysis that considers suspension-based locks.

Bibliography

[1] SchedCAT: The schedulability test collection and toolkit. `https://github.com/brandenburg/schedcat`.

[2] Sara Afshar. *Lock-Based Resource Sharing for Real-Time Multiprocessors*. PhD thesis, Mälardalen University, 2017.

[3] James H Anderson, Rohit Jain, and Kevin Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 346–355, 1998.

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

[5] Alessandro Biondi and Björn B Brandenburg. Lightweight real-time synchronization under p-edf on symmetric and asymmetric multiprocessors. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 39–49. IEEE, 2016.

[6] Alessandro Biondi, Björn B Brandenburg, and Alexander Wieder. A blocking bound for nested FIFO spin locks. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS)*, pages 291–302, 2016.

[7] Aaron Block, Hennadiy Leontyev, Bjorn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE international conference on embedded and real-time computing systems and applications (RTCSA 2007)*, pages 47–56. IEEE, 2007.

[8] Björn B Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[9] Björn B Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for p-fp scheduling. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 141–152. IEEE, 2013.

[10] Björn B Brandenburg. The fmlp+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 61–71. IEEE, 2014.

[11] Björn B Brandenburg. Multiprocessor real-time locking protocols: A systematic review. *arXiv preprint arXiv:1909.09600*, 2019.

[12] Bjorn B Brandenburg and James H Anderson. Optimality results for multiprocessor real-time locking. In *2010 31st IEEE Real-Time Systems Symposium*, pages 49–60. IEEE, 2010.

[13] Björn B Brandenburg and James H Anderson. The omlp family of optimal multiprocessor real-time locking protocols. *Design automation for embedded systems*, 17(2):277–342, 2013.

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

[15] Yang Chang, Robert I Davis, and Andy J Wellings. Reducing queue lock pessimism in multiprocessor schedulability analysis. In *Proceedings of the 18th International Conference on Real-Time Networks and Systems (RTNS)*, pages 99–108, 2010.

[16] Travis S Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS)*, pages 148–157, 1993.

[17] Robert I Davis and Alan Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 398–409. IEEE, 2009.

[18] UmaMaheswari C Devi, Hennadiy Leontyev, and James H Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 75–84, 2006.

[19] Arvind Easwaran and Björn Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 377–386, 2009.

[20] Glenn A Elliott and James H Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems. *Real-Time Systems*, 49(2):140–170, 2013.

[21] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the Synthesis of Multiprocessor Tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'10)*, 2010.

[22] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings 22nd ieee real-time systems symposium (rtss 2001)(Cat. No. 01PR1420)*, pages 73–83. IEEE, 2001.

[23] 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.

[24] Philip Holman and James H Anderson. Locking in pfair-scheduled multiprocessor systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 149–158. IEEE, 2002.

[25] Philip Holman and James H Anderson. Object sharing in pfair-scheduled multiprocessor systems. In *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, pages 111–120. IEEE, 2002.

[26] Philip Holman and James H Anderson. Locking under pfair scheduling. *ACM Transactions on Computer Systems (TOCS)*, 24(2):140–174, 2006.

[27] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *2009 30th IEEE Real-Time Systems Symposium*, pages 469–478. IEEE, 2009.

[28] Sylvain Lauzac, Rami Melhem, and Daniel Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No. 98EX168)*, pages 188–195. IEEE, 1998.

[29] Victor B Lortz and Kang G Shin. Semaphore queue priority assignment for real-time multiprocessor synchronization. *IEEE Transactions on Software Engineering*, 21(10):834–844, 1995.

[30] Zhongqi Ma, Ryo Kurachi, Gang Zeng, and Hiroaki Takada. Further analysis on blocking time bounds for partitioned fixed priority multiprocessor scheduling. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–6. IEEE, 2016.

[31] Mitra Nasri and Bjorn B Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23. IEEE, 2017.

[32] 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*, pages 9–1, 2018.

[33] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. Response-time analysis of limited-preemptive parallel dag tasks under global scheduling. In *31st Conference on Real-Time Systems*, pages 21–1, 2019.

[34] Saranya Natarajan, Mitra Nasri, David Broman, Björn B. Brandenburg, and Geoffrey Nelissen. From code to weakly hard constraints: A pragmatic end-to-end toolchain for timed c. In *IEEE Real-Time Systems Symposium*, pages 167–180, 2019.

[35] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings., 10th International Conference on Distributed Computing Systems*, pages 116–117. IEEE Computer Society, 1990.

[36] Ragunathan Rajkumar. Dealing with suspending periodic tasks. *IBM Thomas J. Watson Research Center*, 1991.

[37] Ragunathan Rajkumar. *Synchronization in real-time systems: a priority inheritance approach*, volume 151. Springer Science & Business Media, 2012.

[38] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[39] Alexander Wieder and Björn B Brandenburg. On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 45–56. IEEE, 2013.

[40] Beyazit Yalcinkaya, Mitra Nasri, and Björn B Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1228–1233. IEEE, 2019.

[41] Maolin Yang, Alexander Wieder, and Björn B Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *2015 IEEE Real-Time Systems Symposium*, pages 1–12. IEEE, 2015.

Appendix A

Partial Order Reduction Proofs

A.1 Non-Starting Segments

Lemma 15 *The start time of a non-starting segment $J_{i,j} \in \mathcal{R}^P$ in v_p is not impacted by any segment $J_{y,z} \notin \mathcal{J}^{\mathcal{P}}$ if $LST_{i,j}(v_p) < ERT_{y,z} \mid \eta_{i,j} \cap \eta_{y,z} \neq \emptyset$*

Proof. Following the workload model from Section 4.1, we know that every non-starting segment accesses a resource, meaning that its start time is dependent on the availability of the shared resource. Since $J_{i,j}$ is not the first segment of a job, it has a claimed core and its start time is therefore affected by when its claimed core becomes available. Hence, the start time of a non-starting segment $J_{i,j}$ in state v_p is dependent on its claimed core availability and the resource availability as described by Equations (5.2), (5.3), (5.6) and (5.8). Since $J_{i,j}$ has a reserved core for its execution, it does not compete for one, meaning that no present or future segment (of another job) can affect the claimed core availability. Furthermore, if the latest start time of $J_{i,j}$ is earlier than the earliest time at which any remaining segment $J_{y,z}$ that needs the same resource ℓ_x can request ℓ_x, then $J_{i,j}$ will certainly get the next access to the resource. This means that its start time is not impacted by any other segment that requires the same resource and has not executed yet. □

Lemma 16 *The execution of a non-starting segment $J_{i,j} \in \mathcal{R}^P$ in v_p does not impact the start time of any segment that requests the same resource as $J_{i,j}$, i.e., $J_{y,z} \notin \mathcal{J}^{\mathcal{P}} \wedge \eta_{i,j} \cap \eta_{y,z} \neq \emptyset$ if $LFT_{i,j}(v_p) < ERT_{y,z}$*

Proof. $J_{i,j}$ has a reserved core and therefore only competes with job segments that request the same resource. Hence, its execution can impact the start time of such segments. However, if by the latest time at which $J_{i,j}$ finishes its execution, no other segment has potentially requested the used resource, we are certain that $J_{i,j}$'s execution has not delayed or impacted the start time of any other job segment that shares the same resource. This follows from the fact that if the condition holds, then any remaining job segment that requests the same resource will do so after $J_{i,j}$ has certainly finished, meaning it is not impacted by $J_{i,j}$. □

Lemma 17 *A state v_p for which there exists a non-starting segment $J_{i,j}$ that can potentially start to execute at $EST(v_p) = EST_{i,j}$ and for which Eqs.* (7.1) *and* (7.2) *hold, can be expanded by one edge labeled $J_{i,j}$ without discarding any execution scenario.*

Proof. $EST(v_p)$ represents the earliest time at which any segment could potentially start executing in state v_p. Since $EST(v_p) = EST_{i,j}$, no job segment can start prior to $EST_{i,j}$ meaning that no potential start time scenario is discarded in the graph by updating availabilities according to Eqs. (5.17) to (5.20). Since Eq. (7.1) holds, $J_{i,j}$'s start time is not affected by any present or future segment, which means that its start time interval in v_p covers any other start time interval of $J_{i,j}$ that would have resulted from branching in state v_p. Furthermore, since Eq. (7.2) holds, the execution of $J_{i,j}$ does not impact the start time of any other segment that has not yet finished. This means that in a subsequent state v'_p after $J_{i,j}$ has started executing, every scenario that was possible in v_p is still possible in v'_p (obviously except for executing $J_{i,j}$), meaning that we have not discarded any execution scenario. □

A.2 Starting Segments

Lemma 18 *The start time of a starting segment $J_{i,j}$ in v_p that does not access a resource, is not impacted by any present or future segment, if in every instance of $J_{i,j}$'s release interval ($[r_i^{min}, r_i^{max}]$), the number of k certainly available cores $|A_k^{max}|$ is larger or equal to the number of possibly released starting segments.*

Proof. The execution of a considered starting segment $J_{i,j}$ in v_p, which does not request a resource, depends on the availability of a core. Jobs that have already started in v_p have a claimed core and since $J_{i,j}$ does not access a resource, segments belonging to such jobs do not affect the start time of $J_{i,j}$. Since $J_{i,j}$ competes with other starting segments for a free core in order to execute, its start time is not impacted, if in every possible instance of $J_{i,j}$'s release interval there is a free core for it to execute on. Given that in every instance of $J_{i,j}$'s release interval the number of jobs that possibly try to claim a free core is smaller or equal to the number of certainly available cores, $J_{i,j}$ will definitely have a core to execute on, which means that in this execution scenario it is not impacted or delayed by any other segment. This proves the given lemma. □

Lemma 19 *The start time of any starting segment $J_{y,z}$ that has not yet started to execute, is not impacted by a considered starting segment $J_{i,j}$ in v_p that does not access any resource, if $EST(v_p) = EST_{i,j}$ and in every instance of $J_{i,j}$'s release interval ($[r_i^{min}, r_i^{max}]$), the number of k certainly available cores $|A_k^{max}|$ is larger or equal to the number of possibly released starting segments.*

Proof. $EST(v_p)$ represents the earliest time at which any segment could potentially start executing in state v_p. Since $EST(v_p) = EST_{i,j}$, no job segment can start prior to $EST_{i,j}$ meaning that no potential start time scenario is discarded in the graph by updating availabilities according to Eqs. (5.17) to (5.20). Furthermore, since at every time instance of $J_{i,j}$'s release interval the amount of certainly available cores is larger or equal to the number of jobs that potentially want to claim a core, $J_{i,j}$'s execution will not impact or delay the potential start

time of any other starting segment. Finally, all non-starting segment are not impacted since they already have a claimed core and $J_{i,j}$ does not access any resource. □

Lemma 20 *Expanding state v_p in the schedule abstraction graph without branching, considering a starting segment $J_{i,j}$, that does not access a resource, does not discard any execution scenario if, $EST(v_p) = EST_{i,j}$ and in every instance of $J_{i,j}$'s release interval ($[r_i^{min}, r_i^{max}]$), the number of k certainly available cores $|A_k^{max}|$ is larger or equal to the number of possibly released starting segments.*

Proof. No execution scenario is discarded from the schedule abstraction graph, if the considered segment $J_{i,j}$ is not impacted by any other present or future segment and the execution of $J_{i,j}$ does not impact any other segment that is not in the path yet. Since, we have proven in Lemmas 18 and 19 that under these conditions $J_{i,j}$ is not impacted by and does not impact any segment that still needs to execute, this lemma holds. □

Appendix B

Modified Rules for Multi-Unit Resources

B.1 Earliest and Latest Start Time

Lemma 1 (b). *Segment $J_{i,j} \in \mathcal{R}^P$ cannot start executing (as a successor of state v_p) before $EST_{i,j}(v_p)$.*

Proof. The first segment $J_{i,1}$ of a job J_i can start its execution only if (i) it is released, (ii) the shared resource $\ell_{x,k}$ it requests (if any) is available **and** (iii) a core is available. Thus, if all cores have already been claimed by other jobs (i.e., $|\mathcal{C}(v_p)| = m|$) then $J_{i,1}$ cannot be a successor of v_p and $EST_{i,j}(v_p) = \infty$. This proves the first case of Eq. (5.2).

However, if there is a free core (i.e., $|\mathcal{C}(v_p)| < m|$), then, by definition, A_1^{min} is the earliest time at which a core can potentially become available. Furthermore, r_i^{min} is the earliest release time of J_i and $SR_{i,j}^{min}(v_p)$ is the earliest time at which one token of shared multi-unit resource $\ell_{x,k}$ accessed by $J_{i,j}$ may become available. Thus, the earliest time at which $J_{i,1}$ may start to execute is given by $EST_{i,j}(v_p) = \max\{r_i^{min},\ A_1^{min}(v_p),\ SR_{i,j}^{min}(v_p)\}$ if $j = 1$. This proves the second case of Eq. (5.2).

Any segment that is not the first segment of a job (i.e., a segment $J_{i,j}$ with $j > 1$) can start its execution only if (i) the core claimed by the preceding segments belonging to the same job is available, **and** (ii) the multi-unit resource it requests (if any) is also available. Since $J_{i,j}$ is in $\mathcal{R}^P$, all the segments of J_i that precede $J_{i,j}$ must have started (and potentially finished) executing on the core claimed by J_i. Thus, the earliest time at which the core claimed by J_i may become available for $J_{i,j}$ is given by $Cl_i^{min}(v_p)$. Therefore, the earliest time at which $J_{i,j}$ may start to execute is $\max\{Cl_i^{min}(v_p),\ SR_{i,j}^{min}(v_p)\}$ if $j > 1$. This proves the last case of Eq. (5.2). □

Lemma 2 (b). *An upper bound on the time at which a segment $J_{y,z}$ can certainly start executing (as a successor of state v_p) is given by $tw_{y,z}$.*

Proof. Infinity is an obvious upper bound on the start time of $J_{y,z}$. Therefore, in this proof, we only focus on the cases where $tw_{y,z} \neq \infty$.

A starting segment $J_{y,1}$ *must* start to execute as soon as (i) it is released, (ii) the multi-unit resource $\ell_{x,k}$ it requests (if any) is available and (iii) a core is available. By definition, r_y^{max} is an upper bound on the release time of $J_{y,1}$, $SR_{y,z}^{max}(v_p)$ is an upper bound on the availability time of one shared resource unit (of $\ell_{x,k}$) accessed by $J_{y,z}$ (if any) and $A_1^{max}(v_p)$ denotes the time at which a core is certainly available in state v_p. Thus, $J_{y,1}$ can certainly start executing at $\max\{r_y^{max}, A_1^{max}, SR_{y,1}^{max}(v_p)\}$ when $z = 1$ and there is at least one free core in v_p.

Any segment $J_{y,z}$ that is not the first segment of $J_{y,z}$ (i.e., with $z > 1$) can certainly start executing when (i) one roken of the resource it requests (if any) is available, and (ii) the segments of J_y that precede $J_{y,z}$ have all completed their execution on the core claimed by J_y. Since $J_{y,z}$ is in $\mathcal{R}^P$, all the segments of $J_{y,z}$ preceding $J_{y,z}$ have already started (and potentially finished) executing on the core claimed by J_y, and because $Cl_y^{max}(v_p)$ is an upper bound on the time at which the core claimed by J_y becomes available to execute the next segment of J_y, we have that $J_{y,z}$ certainly starts at $\max\{Cl_y^{max}(v_p), SR_{y,z}^{max}(v_p)\}$ when $z > 1$. □

Lemma 4 (b). *Let $J_{y,z}$ be a segment in $\mathcal{H}$. If $J_{i,j}$ did not start to execute before $th_{y,z}(J_{i,j})$, then $J_{y,z}$ will start before $J_{i,j}$ in a system considering FIFO-ordered spin locks.*

Proof. We analyze the three cases of Eq. (5.6).
Case 1. Assume that both $J_{i,j}$ and $J_{y,z}$ are the first segment of their respective job (i.e., $j = z = 1$). For a starting segment to be able to execute, it needs to be (1) released, (2) at least one token of the resource it requests (if it requests one) must be available and (3) a core must be available for it to execute on. By definition, r_y^{max} is an upper bound on (1) and $SR_{y,z}^{max}(v_p)$ is an upper bound on (2). Regarding (3), we note that because $J_{y,1}$ and $J_{i,1}$ are both starting segments, they both compete for the same available cores. Since $J_{y,1}$ has a higher priority than $J_{i,j}$, if both $J_{y,1}$ and $J_{i,j}$ are ready and have their shared resources available at the same time, then $J_{y,1}$ will certainly start before $J_{i,j}$. Therefore, only (1) and (2) decide whether $J_{y,1}$ will start to execute before $J_{i,j}$. Thus, if $J_{i,j}$ did not start before $\max\{r_y^{max}, SR_{y,z}^{max}(v_p)\}$ when $z = j = 1$, then $J_{y,1}$ will be dispatched before $J_{i,j}$.

If $J_{i,j}$ is not a starting segment (i.e., $j > 1$), then it already has a reserved core. Therefore, it does not compete with $J_{y,z}$ for the same core (i.e., we must account for (3)).
Case 2. If $J_{y,z}$ is the first segment of the higher priority job J_y, then, by definition, $A_1^{max}(v_p)$ is a safe upper bound on (3). Thus, because $J_{y,z}$ has higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{A_1^{max}(v_p), r_y^{max}, SR_{y,z}^{max}(v_p)\}$ when $z = 1 \wedge j > 1$.
Case 3. If $J_{y,z}$ is not the first segment of the higher priority job J_y, then job J_y is already released and it already claimed a core. Thus, by definition, $Cl_y^{max}(v_p)$ is an upper bound on (3). Hence, as $J_{y,z}$ has higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{Cl_y^{max}(v_p), SR_{y,z}^{max}(v_p)\}$ when $z > 1 \wedge j > 1$. □

Lemma 6 (b). *Let $J_{y,z} \in \{\mathcal{R}^P \cap hp(J_{i,j})\}$ be a segment that has a higher priority than the considered segment $J_{i,j}$. If $J_{i,j}$ did not start to execute before*

$th_{y,z}(J_{i,j})$, then $J_{y,z}$ will start before $J_{i,j}$ in a system with priority-ordered spin locks.

Proof. We analyze the three cases of Eq. (5.8).
Case 1. Assume that both $J_{i,j}$ and $J_{y,z}$ are the first segment of their respective job (i.e., $j = z = 1$). For a starting segment to be able to execute, it needs to be (1) released, (2) at least one token of the resource it requests (if it requests one) must be available and (3) a core must be available for it to execute on. By definition, r_y^{max} is an upper bound on (1) and $SR_{y,z}^{max}(v_p)$ is an upper bound on (2). Regarding (3), we note that because $J_{y,1}$ and $J_{i,1}$ are both starting segments, they both compete for the same available cores. Since $J_{y,1}$ has a higher priority than $J_{i,1}$, if both $J_{y,1}$ and $J_{i,1}$ are ready and have their shared resources available at the same time, then $J_{y,1}$ will certainly start before $J_{i,1}$. Therefore, only (1) and (2) decide whether $J_{y,1}$ will start to execute before $J_{i,1}$. Similarly, we only consider $t_r = SR_{y,z}^{max}(v_p)$ in (2) if $J_{y,z}$ does not request the same multi-unit resource as $J_{i,j}$ (i.e., $\eta_{y,z} \neq \eta_{i,j}$), because otherwise it would mean that they compete for the same resource in which case $J_{y,1}$ will certainly start before $J_{i,1}$, due to its higher priority, if they are both ready and a core is available. If they do share the same resource or any of the segments is resource-independent, we simply ignore (2) (i.e., set $t_r = 0$). Thus, if $J_{i,j}$ did not start before $\max\{r_y^{max}, t_r\}$ when $z = j = 1$, then $J_{y,z}$ will be dispatched before $J_{i,j}$.

If $J_{i,j}$ is not a starting segment (i.e., $j > 1$), then it already has a reserved core. Therefore, it does not compete with $J_{y,z}$ for the same core (i.e., we must account for (3)).
Case 2. If $J_{y,z}$ is the first segment of the higher priority job J_y, then, by definition, $A_1^{max}(v_p)$ is a safe upper bound on (3). Thus, because $J_{y,z}$ has higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{A_1^{max}(v_p), r_y^{max}, t_r\}$ when $z = 1 \wedge j > 1$.
Case 3. If $J_{y,z}$ is not the first segment of the higher priority job J_y, then job J_y is already released and it already claimed a core. Thus, by definition, $Cl_y^{max}(v_p)$ is an upper bound on (3). Hence, as $J_{y,z}$ has higher priority than $J_{i,j}$, $J_{y,z}$ will be dispatched before $J_{i,j}$ if $J_{i,j}$ did not start to execute before $\max\{Cl_y^{max}(v_p), t_r\}$ when $z > 1 \wedge j > 1$. □