Certification-Cognizant Mixed-Criticality Scheduling in Time-Triggered Systems

vom
Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von
Jens Theis
geboren in Neunkirchen (Saar)

D 386

Eingereicht am:	05.02.2015
Tag der mündlichen Prüfung:	13.03.2015
Dekan des Fachbereichs:	Prof. Dr.-Ing. Hans D. Schotten
Promotionskomission	
Vorsitzender:	Prof. Dr.-Ing. Wolfgang Kunz
Berichterstattende:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
	Prof. Alan Burns

Erklärung gem. § 6 Abs. 3 Promotionsordnung

Ich versichere, dass ich diese Dissertation selbst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzen Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Dissertation wurde weder als Ganzes noch in Teilen als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht. Es wurde weder diese noch eine andere Abhandlung bei einem anderen Fachbereich oder einer anderen Universität als Dissertation eingereicht.

(Ort, Datum) (Jens Theis)

Abstract

In embedded systems, there is a trend of integrating several different functionalities on a common platform. This has been enabled by increasing processing power and the arise of integrated system-on-chips. The composition of safety-critical and non-safety-critical applications results in mixed-criticality systems. Certification Authorities (CAs) demand the certification of safety-critical applications with strong confidence in the execution time bounds. As a consequence, CAs use conservative assumptions in the worst-case execution time (WCET) analysis which result in more pessimistic WCETs than the ones used by designers. The existence of certified safety-critical and non-safety-critical applications can be represented by dual-criticality systems, i.e., systems with two criticality levels.

In this thesis, we focus on the scheduling of mixed-criticality systems which are subject to certification. Scheduling policies cognizant of the mixed-criticality nature of the systems and the certification requirements are needed for efficient and effective scheduling. Furthermore, we aim at reducing the certification costs to allow faster modification and upgrading, and less error-prone certification. Besides certification aspects, requirements of different operational modes result in challenging problems for the scheduling process. Despite the mentioned problems, schedulers require a low runtime overhead for an efficient execution at runtime.

The presented solutions are centered around time-triggered systems which feature a low runtime overhead. We present a transformation to include event-triggered activities, represented by sporadic tasks, already into the offline scheduling process. Further, this transformation can also be applied on periodic tasks to shorten the length of schedule tables which reduces certification costs. These results can be used in our method to construct schedule tables which creates two schedule tables to fulfill the requirements of dual-criticality systems using mode changes at runtime. Finally, we present a scheduler based on the slot-shifting algorithm for mixed-criticality systems. In a first version, the method schedules dual-criticality jobs without the need for mode changes. An already certified schedule table can be used and at runtime, the scheduler reacts to the actual behavior of the jobs and thus, makes effective use of the available resources. Next, we extend this method to schedule mixed-criticality job sets with different operational modes. As a result, we can schedule jobs with varying parameters in different modes.

To my parents, Elke and Manfred Theis.

Nur wer selber brennt,

kann andere entzünden.[1]

[1] English translation: “Only he who burns himself can ignite others.”

Acknowledgments

Writing the first words of a thesis or paper are always the most difficult ones. You stare at the screen and you do not know how to start. Now, I am doing this again and there is the temptation to look into the acknowledgments of someone else to get inspired, but this part is too important for myself to simply copy ideas. Hence, I resist this temptation and make these acknowledgments to **my acknowledgments**.

After years of learning, trying to understand problems, and thinking about their solutions, I am finishing writing my PhD thesis. I wish I would not be indebted to anybody, but in fact, I will never be able to give back all the support from my family, my friends, my supervisor, and my colleagues.

First, I want to express my gratitude to Prof. Gerhard Fohler, who gave me the chance to work on this thesis under his supervision, for his guidance and the valuable discussions in the last years. I am grateful for the atmosphere he created at the Department of Real-Time System where I enjoyed working with and talking to colleagues who are unique in their own special way. In line with this, I want to thank the other members of my thesis committee, Prof. Wolfgang Kunz and Prof. Alan Burns.

Next, I want thank my (ex-)colleagues Ramon Serna Oliver, Raphael Guerra, Rodrigo Coelho, Anand Kotra, Stefan Schorr, Cuong Viet Ngo, Anoop Bhagyanath, Naga Rajesh, Markus Metz, Simara Pérez, Mitra Nasri, Ankit Agrawal, Ali Abbas Syed, and Gautam Gala. I enjoyed working with you and I also liked the hours of "procrastination" and fun.

As a PhD student in Germany, you are thankful for a person who practices some kind of sorcery: turning empty sheets into filled out travel authorization requests, travel expenses reports, and much more. In my case, I am deeply grateful for Stephanie Jung who supported me as "Sorceress Steffi" and was also a source of joy and support and relief during my time at the Real-Time Systems department. Thanks for your support and the great time, Steffi.

What do you need to become a PhD? Of course, you need good ideas, people to discuss them, write papers, and publish them. However, you cannot do this without technical devices to program your algorithms, write paper submission, print your texts, prepare presentations, and so on. For this reason, I want to thank Markus Müller for his technical support with the small and big problems modern computers (and printers) cause. Besides this, his jokes enriched everybody's daily routine and it was a pleasure to work you, Markus. Thank you Markus, "Earl of Backup" and "Lord of SVN".

Last but not least, I want to thank the students Pramod Murthy and Lars Guse who supported me with their work at the Real-Time Systems department.

All of you, I wish the best for your future life. May the odds be ever in your favor.

Writing the text of your thesis is a long process. Starting from the first word and ending with a printed version of you thesis. At some point, you do not read what you have written in your texts but you read what you wanted to write. For this reason, I want thank my proof-readers Ali, Ankit, Gautam, Rodrigo, and Stefan from our department. I want to thank even more my friends Bianca, Christine, Julia, Kerstin, and Nina, who spent their spare time to proof-read my thesis. I have no idea how difficult it was for you to read these texts without any background knowledge of electrical engineering and information technology. Thank you so much for your help.

On a personal side, I want thank my family for their support, especially during the recent years of studying and working on my thesis. My parents made it possible for me to study, supported and encouraged me so that I can now write the lines of my thesis which are most important from my personal view. No words can describe how grateful I am, hence, I keep it simple: Thank you!

As important as your family are your friends. You meet people and some become friends... few of them become true friends forever. They share their life with you and you share your life with them. At some point, some of them go other ways, but independent of where they are, they are still true friends. I want to thank my friends for the time we shared and I am joyfully looking to the future to share more time with you.

Jens Theis
Kaiserslautern, 05.02.2015

The work presented in this thesis has been partially supported by the European project *Distributed Real-time Architecture for Mixed Criticality Systems* (DREAMS) under project No. 610640.

Danksagung

Die ersten Worte einer (Doktor-)Arbeit oder eines Fachartikels zu schreiben, sind immer die schwierigsten. Man starrt den Bildschirm an und weiß nicht, wie man anfangen soll. Ich tue dies jetzt wieder und bin versucht, mir die Danksagung eines anderen anzuschauen, um mich inspirieren zu lassen. Allerdings ist dieser Teil meiner Arbeit für mich persönlich zu wichtig, als dass ich einfach Ideen kopieren möchte. Deshalb widerstehe ich dieser Versuchung und werde diese Danksagung zu **Meiner Danksagung** machen.

Nach Jahren des Lernens, des Versuchens Probleme zu verstehen und über deren Lösungen nachzudenken bin ich im Begriff meine Doktorarbeit fertig zu schreiben. Ich wünschte ich wäre niemanden etwas schuldig, aber die Wahrheit ist, dass ich niemals in der Lage sein werde all die Unterstützung meiner Familie, meiner Freunde, meines Dissertationsbetreuers und meiner Kollegen zurückzahlen zu können.

Als erstes möchte ich meine Dankbarkeit gegenüber Prof. Gerhard Fohler, der mir die Möglichkeit gab diese Arbeit mit Hilfe seiner Betreuung zu verfassen, für seine Anleitung und wertvollen Diskussionen in den letzten Jahren ausdrücken. Ich bin dankbar für die Arbeitsatmosphäre, die er am Lehrstuhl für Echtzeitsysteme schuf, die mir ermöglichte mit Freude zusammen mit einzigartigen Kollegen zu arbeiten. Außerdem möchte ich den anderen Mitgliedern der Prüfungskommission, Prof. Wolfgang Kunz und Prof. Alan Burns, danken.

Als nächstes möchte ich mich bei meinen (ehemaligen) Kollegen Ramon Serna Oliver, Raphael Guerra, Rodrigo Coelho, Anand Kotra, Stefan Schorr, Cuong Viet Ngo, Anoop Bhagyanath, Naga Rajesh, Markus Metz, Simara Pérez, Mitra Nasri, Ankit Agrawal, Ali Abbas Syed und Gautam Gala bedanken. Ich habe die Zusammenarbeit mit euch genossen und erfreute mich an der einen oder anderen Stunde Ablenkung und Spaß mit euch.

Als Doktorand in Deutschland ist man dankbar für eine Person, die eine bestimmte Art Zauberei beherrscht: leere Blätter in ausgefüllte Dienstreiseanträge, Reisekostenabrechnungen und vieles mehr zu verwandeln. In meinem Fall bin ich Stephanie Jung zutiefst dankbar für ihre Unterstützung als "Zauberin Steffi". Sie war außerdem eine Quelle der Freude und Unterstützung und Rückhalt während meiner Zeit am Lehrstuhl für Echtzeitsysteme. Danke für deine Unterstützung und die schöne Zeit, Steffi.

Was benötigt man um einen Doktortitel zu erlangen? Man benötigt natürlich gute Ideen, Leute um diese zu diskutieren, Fachartikel zu schreiben und diese zu veröffentlichen.

Allerdings ist dies nicht möglich ohne die technischen Geräte um deine Algorithmen zu programmieren, Fachartikel zu schreiben, deine Texte auszudrucken, Präsentation vorzubereiten und so vieles mehr. Aus diesem Grund möchte ich Markus Müller für seine technische Unterstützung bei den kleinen und großen Problemen moderner Computer (und Drucker) danken. Außerdem bereicherten seine Witze unseren Alltag und es war eine Freude mit dir zu arbeiten, Markus. Danke Markus, "Fürst des Backups" und "Herr des SVNs".

Letztlich möchte ich noch den Studenten Pramod Murthy und Lars Guse, die mich mit ihrer Arbeit am Lehrstuhl unterstützt haben, bedanken.

Ich wünsche euch allen das Beste für eure Zukunft. Möge das Glück stets mit euch sein.

Das Schreiben der Doktorarbeit ist ein langer Prozess. Angefangen mit dem ersten Wort bis hin zu der gedruckten Fassung der Arbeit. Irgendwann gelangt man an den Punkt, an dem man nicht mehr liest was man geschrieben hat, sondern was man schreiben wollte. Aus diesem Grund danke ich meinen Kollegen Ali, Ankit, Gautam, Rodrigo, und Stefan für das Korrektur lesen. Außerdem möchte ich noch stärker bei meinen Freunden Bianca, Christine, Julia, Kerstin und Nina bedanken, die ihre Freizeit opferten um meine Arbeit Korrektur zu lesen. Ich habe keine Vorstellung, wie schwierig es für euch war, diese Texte zu lesen ohne das elektrotechnische bzw. informationstechnische Hintergrundwissen zu haben. Ich danke euch so sehr für eure Hilfe.

Auf der persönlichen Seite möchte ich meiner Familie für ihre Unterstützung, im Speziellen während den letzten Jahren des Studierens und Arbeitens an meiner Doktorarbeit, bedanken. Meine Eltern ermöglichten es mir zu studieren, unterstützten und ermutigten mich, so dass ich jetzt die Zeilen meiner Arbeit schreiben kann, die für mich persönlich am wichtigsten sind. Worte können nicht beschreiben, wie dankbar ich bin und deshalb sage ich einfach: Danke!

Genauso wichtig wie meine Familie sind mir meine Freunde. Man trifft Leute und manche werden Freunde... wenige werden wahre Freunde für immer. Sie teilen ihr Leben mit dir und du teilst dein Leben mit ihnen. Irgendwann gehen einige von ihnen andere Wege, aber unabhängig wo sie gerade sind, bleiben sie wahre Freunde. Ich möchte mich bei meinen Freunden für die Zeit, die wir zusammen verbracht haben, bedanken und ich schaue mit Freude in die Zukunft, um noch mehr Zeit mit euch zu verbringen.

Jens Theis
Kaiserslautern, 05.02.2015

Die gezeigte Arbeit dieser Dissertation wurde in Teilen durch das europäische Projekt *Distributed Real-time Architecture for Mixed Criticality Systems* (DREAMS) unter der Projektnummer 610640 unterstützt.

Publications

I have authored or co-authored the following publications:

Conference and refereed workshop papers

- Jens Theis and Gerhard Fohler. *Transformation of Sporadic Tasks for Off-line Scheduling with Utilization and Response Time Trade-Offs.* Proceedings of 19th International Conference on Real-Time and Network Systems (RTNS 2011), Nantes, France, September 2011
- Jens Theis, Gerhard Fohler and Sanjoy Baruah. *Schedule Table Generation for Time-Triggered Mixed Criticality Systems.* Proceedings of 1st International Workshop on Mixed Criticality Systems (WMC 2013), Vancouver, Canada, December 2013
- Jens Theis and Gerhard Fohler. *Mixed Criticality Scheduling in Time-Triggered Legacy Systems.* Proceedings of 1st International Workshop on Mixed Criticality Systems (WMC 2013), Vancouver, Canada, December 2013

Technical Reports

- Jens Theis and Gerhard Fohler. *Transformation of Tasks for Minimizing the Hyper-period with an Advanced Hyper-period Calculation.* Technical Report, Chair of Real-Time Systems, Technische Universität Kaiserslautern, December 2012
- Jens Theis and Gerhard Fohler. *Mode Changes for Time-Triggered Systems with Certification-Cognizant Mixed-Criticality Applications.* Technical Report, Chair of Real-Time Systems, Technische Universität Kaiserslautern, February 2013
- Jens Theis and Gerhard Fohler. *Definitions for Slot-based Schedulers.* Technical Report, Chair of Real-Time Systems, Technische Universität Kaiserslautern, June 2013

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

Symbols

$\triangleleft$	marks the end of a definition
◀	marks the end of a theorem
■	marks the end of a proof
$\Diamond$	marks the end of an example
$\mathbb{N}$	set of natural numbers: $\mathbb{N} = \{0, 1, 2, 3, ...\}$

Abbreviations

AMC	Adaptive Mixed-Criticality
CAs	Certification Authorities
CBEDF	Criticality Based Earliest Deadline First
CBS	Constant Bandwidth Server
DPS	Dynamic Priority Scheduling
EDF	Earliest Deadline First
ESA	Exhaustive Search Algorithm
ET	Event-Ttriggered
FA	Federated Approach
FHS	Fast Hyper-period Search
FPS	Fixed Priority Scheduling
GCD	Greatest Common Divisor
G-EDF	Global EDF
IMA	Integrated Modular Avionics
LCM	Least Common Multiple
LLF	Least Laxity First
MCR	Mode Change Request
OCBP	Own Criticality Based Priority

PC	Partitioned Criticality
P-EDF	Partitioned EDF
PES	Priority Exchange Server
PS	Polling Server
RET	Reserved Execution Time
RM	Rate Monotonic
RTS	Real-Time System
SMC	Static Mixed-Criticality
SS	Sporadic Server
TBS	Total Bandwidth Server
TT	Time-Triggered
TTA	Time-Triggered Architectures
UAV	Unmanned Aerial Verhicle
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

Chapter 1

Introduction

Increasing processing power and integrated system-on-chips resulted in the trend of integrating several functionalities on the same shared platform. These mixed-criticality systems are composed of safety-critical and non-safety-critical applications. While strict real-time requirements have to be fulfilled for all functionalities, further requirements of the certification process of safety-critical applications result in additional challenges. The conservative assumptions of Certifications Authorities (CAs) increase the complexity and difficulty of the scheduling process.

An approach commonly used in safety-critical systems is the implementation as a time-triggered (TT) system. The benefit of this approach is the determinism which eases certification, upgrading, and modernization of these systems at a manageable effort. Furthermore, implementing safety-critical and non-safety-critical requirements into different operational modes reflects the different requirements and behaviors under specific circumstances in mixed-criticality systems. Thus, using TT schedulers with mode changes can reduce the certification complexity and can provide efficient and effective scheduling of mixed-criticality systems.

In this chapter, we provide a general introduction to real-time system and the real-time scheduling problem in Section 1.1. In Section 1.2, we introduce the concept of mixed-criticality systems and the challenges of these systems. Next, we describe the problem we deal with in this thesis in Section 1.3. In Section 1.4, we present the main contribution of our work. Finally, we present the organization of the rest of this thesis in Section 1.5.

1.1 Real-Time Systems

Real-time systems (RTSs) are software and hardware systems in which the correctness of computations is not only defined by their values, but also the time when the results are provided. As a consequence, applications are defined including timing constraints. In general, these timing constraints have to be enforced strictly [CL90]. Typical systems are safety-critical systems and essential sub-systems in which violation of timing constraints can result in catastrophic consequences. Besides timing constraints, the environment also introduces constraints as the system cannot control the environment. RTSs have to react within predetermined time intervals to the environmental events.

Enforcing the timing constraints is done by the process of *scheduling*. Applications are usually composed of one or several processes which are represented in the scheduler as tasks and/or jobs. In the scheduling process, we determine the execution order of tasks and jobs. Scheduling algorithms are defined by a set of rules that enforce the timing constraints of the scheduled tasks and jobs. We give a brief classification of scheduling algorithms in Chapter 3. A correct scheduling policy results in a schedule that guarantees the execution of all tasks and jobs within their timing constraints. Major concepts are worst-case execution times (WCETs) and execution windows of tasks and jobs. The WCET of a task or job is an upper bound on the execution time the task or job needs to execute in the worst-case scenario. The execution window describes the time interval between earliest start time and execution deadline of the tasks and jobs. We present more details about the scheduling parameters of tasks and jobs in Chapter 3. Scheduling can be done before the system starts (offline or TT scheduling) or at runtime (online or event-triggered (ET) scheduling). TT scheduling needs full knowledge of the systems parameters, but the advantage is the deterministic behavior.

Definition 1.1. Determinism [Kop11]
A physical system behaves *deterministically* if given an initial state at instant t and a set of future timed inputs, then the future states and the values and times of future outputs are entailed. $\triangleleft$

As a result, the system "is easy to certificate, upgrade and modernize at manageable effort" [TG]. On the contrary, ET scheduling is more flexible at runtime to react to events, and worst-case assumptions are still guaranteed such that the system is predictable. Obtaining predictable results is the major difference of RTS to "conventional" computing systems in which average performance is the major optimization goal.

Definition 1.2. Predictability [But05]
To guarantee a minimum level of performance, the system must be able to *predict* the consequences of any scheduling decision. If some task cannot be guaranteed within its time constraints, the system must notify this fact in advance, so that alternative actions can be planned in time to cope with the event. $\triangleleft$

Well studied example scheduling algorithms are Rate Monotonic (RM) and Earliest Deadline First (EDF) which were presented in [LL73]. Scheduling algorithms produce correct real-time schedules under specified system models and assumptions.

Increasing complexity of systems results in changing requirements during the lifetime of a system. Timing constraints and resource requirements, e.g., execution time demands on processors, are adapted to changes in the environment. Guaranteeing worst-case scenarios in all possible situations can lead to extremely pessimistic assumptions and under-utilized systems. A possible implementation to overcome pessimistic assumptions and under-utilization of systems are mode changes. The system is defined in different modes to separate requirements in different situations. Thus, a more accurate reaction to environmental events is possible without the need of over-provisioning the system by guaranteeing all worst-case scenarios at the same time. As a consequence, new challenging system models and requirements lead to the development of algorithms optimized for the new requirements and optimization goals. In recent years, increasing computational capabilities resulted in the new field of mixed-criticality systems [Ves07]. In the next section, we describe the requirements and characteristics of these mixed-criticality systems in more detail.

1.2 Mixed-Criticality Systems

Cost, energy efficiency, and related considerations resulted in the trend of embedded systems integrating several functionalities on the same shared platform. These functionalities can be both safety-critical and non-safety-critical, which represents mixed-criticality systems. A typical example of software standards addressing mixed-criticality issues are the **AUT**omotive **O**pen **S**ystem **AR**chitecture (AUTOSAR) [AUT] in the automotive industry and ARINC [ARI] in the avionics domain.

Typically, the mixed-criticality nature of tasks in scheduling theory is expressed by different assumptions about their WCET. The WCET is a conservative upper bound on the execution time. A tight determination of WCETs is very difficult in practice, e.g., due to hardware unpredictabilities as caches, pipelines, speculative execution, and out-of-order execution.

Independent of the importance of tasks, the mixed-criticality aspect often results in a categorization into tasks that are subject to certification and tasks that are not. “Future systems are likely to be constructed from compostable components with known levels of certifiability. The challenge is to identify, develop and implement both a certification process and a compostability framework that will support compostable and incremental certification” [Sys]. Certification is usually done by Certification Authorities (CAs) which are only concerned with the correctness of safety-critical functionalities whereas designers are concerned with the correctness of the entire system. CAs make assumptions about the worst-case behavior of tasks which are more pessimistic and conservative than the worst-case assumptions by designers. The resulting certification requirement is a strong confidence that the WCETs are actual upper bounds on the execution times of the code [BLS10].

The certification process is often centered on safety-critical standards such as DO-178B [Joh92]. Certification of integrated platforms is often done by complete spatial and temporal isolation, e.g., in “ARINC 653-1: Avionics application software standard interface” [ARI03], which can result in sub-optimally usage of resources. As a result, industry, academia, and standards bodies search for efficient and cost-effective certification processes [BBD$^+$12b]. One possible approach is the development of certification-cognizant mixed-criticality scheduling methods.

The term criticality is used in different ways which has been pointed out by Graydon and Bate [GB13]. In this thesis, we focus on the scheduling view of mixed-criticality systems. Dual-criticality systems are system with two criticality levels. In this thesis, we use the criticality levels LO and HI for dual-criticality systems. Varying parameters for different task behaviors result in the need for flexible scheduling. In a dual-criticality system, a task exhibits LO-criticality behavior if its actual execution time is not greater than the LO-criticality WCET. LO-criticality tasks are not allowed to execute for more than their specified LO-criticality WCET which then corresponds to an erroneous behavior. If the actual execution time is greater than the LO-criticality one and does not exceed the HI-criticality WCET, then the tasks exhibits HI-critical behavior. Else the task shows erroneous behavior. Scheduling policies have to deal with the problem of *criticality inversion*, i.e., when a lower criticality task is favored over a higher criticality task [dNLR09].

An often used approach is TT mixed-criticality schedulers with mode changes. Different operational modes reflect different requirements and behaviors of the system. For instance, in a dual-criticality system, LO- and HI-criticality behavior can be implemented in two modes. A transition to the HI-criticality mode is only a theoretical possibility that the scheduling analysis can exploit [BV08]. In a robust priority assignment, it is less likely that a mode change to a HI-criticality level is necessary [BBD11, BB13]. A common assumption is that LO-criticality tasks can be abandoned in case of a HI-criticality system state. Burns and Baruah showed an approach in which LO-criticality tasks are not skipped on a mode change [BB13]. As alternative to skipping them, LO-criticality tasks should be allowed to make some progress as long as they are not interfering with HI-criticality tasks. Furthermore, the authors state that a mode change back to LO-criticality mode should also be possible. Finally, mode changes can only be triggered by HI-criticality tasks because LO-criticality tasks are not allowed to execute for longer than their LO-criticality WCET.

1.3 Problem Statement

The trend to mixed-criticality systems causes new challenges in scheduling theory. In the following, we show the major problems which we focus on in this thesis.

Traditional scheduling criteria as deadlines (e.g., Earliest Deadline First) and period- or deadline-based priority assignments (e.g., Rate Monotonic or Deadline Monotonic) are not enough for effective scheduling algorithms. The different nature of safety-critical and non-safety-critical functionalities require scheduling policies which accommodate for the mixed-criticality parameters of tasks and jobs. In the simplest case, a system is composed of LO-criticality and HI-criticality tasks and jobs which are either subject to certification or not. Furthermore, a hierarchical structure of criticality levels indicating the consequences of not fulfilled timing requirements is also used in practice. For instance, in DO-178B, five criticality levels ranging from catastrophic consequences, i.e., loss of essential functions, to no consequences, i.e., the failure has no effects on the safe operation of the system, are used in case of a failing task or job. As a result, a scheduling policy which is cognizant of the mixed-criticality and certification requirements is needed to guarantee safe operation of the safety-critical mixed-criticality systems. The major goal is an effective use of the resources while guaranteeing the system requirements.

Additionally to the effective scheduling, the certification aspect is important in safety-critical mixed-criticality systems. Simplified certification procedures are less error-prone. Furthermore, a simplified certification is less time consuming such that changes and upgrades of a system can be implemented and certified faster, resulting in a shorter time-to-market. As a consequence, scheduling policies should aim at low or decreased certification costs.

Multiple functionalities in mixed-criticality systems can also result in varying parameters of the tasks and jobs based on changing requirements depending on the system state. Besides two or multiple criticality levels, systems can be designed by several operational modes. Thus, task and job parameters may change not only based on the criticality levels and certification requirements but also based on the current operational mode of the system. As a consequence, the scheduler has to be able to react to the different requirements of the mixed-criticality applications.

Although many different requirements of mixed-criticality systems have to be fulfilled, the goal of scheduler design is also to develop efficient scheduling policies. The scheduler should feature a low runtime overhead. High runtime overheads consume essential resources such that the scheduling and execution of the actual workload is becoming more challenging. As a consequence, this can result in more expensive systems to compensate for the high overheads.

1.4 Contributions

In the following, we briefly summarize the contributions of this thesis with respect to the aforementioned problem statement.

First, we present a method to transform sporadic tasks into periodic reservation tasks. The transformation works on the sporadic task parameters independent of the actual scheduling policy. Due to the unknown arrival times of the sporadic task instances at runtime, we have to guarantee that we execute all instances even with worst case arrival. Using this transformation, we can choose from a range of possible parameters of the periodic reservation tasks and thus, gain some flexibility in the task parameters. Further, we include the switching overhead at runtime into the computation of the reservation task parameters such that we can control the runtime overhead caused by the transformation. As a result, we can include ET activities already in the offline scheduling process. Additionally, we show how we can use the method to transform periodic tasks to obtain additional WCETs and periods. The additional periods can be used to minimize the hyper-period of schedule tables. The minimization of the hyper-period improves the certification costs of TT schedule tables. TT schedule tables represent a constructive proof and hence, only the correctness of the scheduling tables in the table have to be checked. Shorter schedule tables reduce the time to check the correctness of the schedule table.

Second, we show an algorithm to create TT schedule tables with dual-criticality job sets. The scheduling process results in two schedule tables whereas one table is constructed such that certification requirements are guaranteed. The algorithm consists of several components: to reduce the complexity of scheduling, we present a method to split mixed-criticality jobs into jobs with only one WCET. Furthermore, the complexity of the scheduling process is reduced by considering the demand of HI-criticality jobs during scheduling and backtracking. The constructive proof by the scheduling tables simplifies the certification and thus, reduces the certification costs. At runtime, the jobs have only to be dispatched according to the schedule table and thus, we obtain a low runtime overhead.

Next, we present a method based on slot-shifting [Foh95] to schedule dual-criticality jobs in a TT system. The selection function includes the criticality levels of jobs and the resulting computational requirements into the scheduling process. Thus, we can guarantee the HI-criticality jobs which are certified. The method is based on an already certified offline schedule table such that there is no need for re-certification. At runtime, slot-shifting is able to react on the actual behavior of all jobs and assign the available computation time to other jobs. Schorr and Fohler showed that slot-shifting has an applicable runtime overhead [SF13]. Furthermore, we can include ET activities based on slot-shifting's acceptance test at runtime.

Finally, we extend the dual-criticality slot-shifting approach shown before to generic mode changes. Using this method, we can schedule non-mixed-criticality or mixed-criticality jobs with varying parameters in different modes. The flexibility of the jobs ranges from different execution parameters in different modes to added or removed jobs in other modes. We include two versions such that either we can check the feasibility of mode changes at every instant of the schedule or we trigger mode changes to guarantee the timing constraints of jobs with criticality levels higher than the current system state. Furthermore, we provide an acceptance test for ET jobs such that we can test the feasibility of including them into modes in which the jobs are active.

1.5 Organization

The rest of this thesis is structured into the following chapters:

Chapter 2 – Related Work

In this chapter, we describe some approaches related to the presented work in this thesis which form the background of the presented methods.

Chapter 3 – Fundamentals

In this chapter, we show terms and notations used in this thesis. Furthermore, we also present the basic concepts of tasks, jobs, and schedulers. Based on these concepts, we define fundamental properties of task sets, job sets, and schedules.

Chapter 4 – Task Parameter Transformation

In this chapter, we present a transformation of sporadic tasks into periodic reservation tasks which can be used to guarantee the runtime execution of sporadic tasks already in the TT schedule table. We analyze our method with respect to the reserved utilization and the worst-case response times of the sporadic tasks. An evaluation which uses the transformation to minimize the hyper-periodic of periodic task sets shows the applicability of the method and concludes the chapter.

Chapter 5 – Time-Triggered Schedule Tables with Mode Changes

In this chapter, we propose an offline scheduling algorithm. In this algorithm, we create two schedule tables for dual-criticality TT systems. The method consists of three major steps: First, the LO-criticality demand is separated from the additionally needed demand in the HI-criticality case by splitting the mixed-criticality jobs into non-mixed-criticality jobs. Second, we implement a variable called *leeway* to early detect paths in a search tree that will result in an infeasible schedule and thus, reduce the complexity of the scheduling process. Finally, we show a backtracking method based on the leeway to reduce the complexity of backtracking. Then, the approach is evaluated by an empirical comparison with a fixed priority scheduling (FPS) mixed-criticality scheduler. A discussion on the complexity of the certification concludes the chapter.

Chapter 6 – Mixed-Criticality Slot-Shifting without Mode Changes

In this chapter, we describe a method based on slot-shifting to schedule dual-criticality jobs in a TT system. The method uses a (certified) schedule table of a legacy system as input to guarantee HI-criticality requirements without the need for mode changes. LO-criticality jobs are only abandoned if this is necessary to fulfill the timing requirements of HI-criticality jobs. A discussion about dynamic resource usage concludes the chapter.

Chapter 7 – Slot-Shifting with Generic Mode Changes

In this chapter, we present a generalization of the method shown in Chapter 6 by an extension to mixed-criticality jobs with more than two criticality levels. Additionally, the extension is capable of scheduling jobs with generic mode changes. To conclude the chapter, we discuss how much flexibility in the mixed-criticality model is reasonable.

Chapter 8 – Conclusions

In this chapter, we summarize the contributions of this thesis and bring the concluding remarks.

Chapter 2

Related Work

In this chapter, we present mixed-criticality scheduling methods related to this thesis' work. Section 2.1 shows event-triggered (ET) runtime schedulers. The section is divided into fixed priority scheduling (FPS) and dynamic priority scheduling (DPS). In Section 2.2, we present time-triggered (TT) schedulers which create one or several schedule tables to determine the order of executions at runtime.

2.1 Event-Triggered Approaches

The approaches presented in the following aim at uni-processor platforms unless mentioned differently. Besides the shown approaches, there are many other event-triggered (ET) algorithms, e.g., [BBD$^+$12b, BV08, SPBB13, BBD$^+$12a].

2.1.1 Fixed Priority Scheduling

Vestal proposed the mixed-criticality task model [Ves07] which we show in more detail in Section 3.1.2. Furthermore, he presented an algorithm to schedule mixed-criticality tasks. In his approach, priorities are assigned according to deadline monotonic, i.e., tasks with shorter relative deadlines have a higher priority. Periods and worst-case execution times (WCETs) of tasks with higher criticality level are divided into several portions, i.e., shorter periods with proportionally decreased WCETs, such that a higher criticality level is reflected by a higher priority. At runtime, the tasks are dispatched according to their priorities. Additionally, Vestal considered an alternative priority assignment by the so-called "Audsley approach". In the Audsley approach, we check whether a task can be assigned the lowest priority without violation of its timing constraints. If this is not possible, then another task is selected to test whether it can meet its deadline if it is assigned the lowest priority. If this is possible, the task is assigned the lowest priority, removed from the set of remaining tasks, and the test for the lowest priority continues with the set of remaining tasks until the set of remaining tasks is empty or the method cannot find a task which can have the lowest priority.

Baruah et al. presented a fixed priority scheduling (FPS) approach for dual-criticality systems, i.e., mixed-criticality systems with two criticality levels LO and HI [BBD13]. If the system is in LO-criticality system state, priorities are used which are determined using the LO-criticality WCETs. When the system changes to the HI-criticality system state, LO-criticality tasks are dropped and another priority assignment is used for the HI-criticality tasks. The improvement to the Vestal approach shown before is the possibility to use a different priority assignment at runtime when the system is in HI-criticality system state.

Burns and Baruah showed a dual-criticality approach which does not abandon LO-criticality tasks when the system is switched to HI-criticality system state [BB13]. To not drop LO-criticality tasks, three possible changes are shown when the system changes its state to HI. First, it is possible to reduce the priority of LO-criticality tasks. Second, LO-criticality tasks are allowed to execute for a shorter time, i.e., the execution budget of these tasks is reduced. Finally, it is possible to increase the periods of LO-criticality tasks.

Baruah et al. showed a method for a fixed priority assignment of periodic tasks with more than two criticality levels [BBD11]. In the Partitioned Criticality (PC) scheme priorities are assigned according to the criticality level of tasks. Within each criticality level, the priorities are assigned by an optimal priority assignment as, e.g., deadline monotonic. The benefit of this approach is that runtime monitoring is not necessary. In addition, the authors showed two schemes for dual-criticality systems. The priority assignment of the Static Mixed-Criticality (SMC) scheme is based on the Audsley approach. At runtime, each task is allowed to execute up to its representative WCET. The second scheme, the Adaptive Mixed-Criticality (AMC) scheme, improves the performance of SMC. As in SMC, priorities are assigned according to the Audsley approach. A criticality level indicator, initialized to LO, is used to represent the current state of the system. As long as the level indicator is set to LO, tasks are scheduled according to their priority. If a HI-criticality task does not signal completion by its LO-criticality WCET, the criticality level indicator is set to HI. As a consequence, LO-criticality tasks are not executed anymore.

Baruah et al. presented the Own Criticality Based Priority (OCBP) approach for time-triggered (TT) jobs with an arbitrary number of criticality levels [BLS10]. Priorities are assigned based only on the criticality of a job. A job is assigned the lowest priority if there is enough time to schedule its WCET at its criticality level within its execution window even if all other jobs have a higher priority. If a job can be assigned the lowest priority, this procedure is repeated with the remaining jobs; otherwise another job is tested for the lowest priority. Li and Baruah showed how the OCBP approach can be used to schedule dual-criticality sporadic task systems [LB10]. They solve the problem that OCBP needs full knowledge of the job parameters to assign the priorities which are not known for sporadic tasks before runtime.

De Niz et al. showed the Zero-Slack Scheduling approach for dual-criticality systems [dNLR09]. In this approach, HI-criticality tasks can run for their HI-criticality WCET. If HI-criticality tasks only execute for their LO-criticality WCET, the difference can be added to the slack which can be used by LO-criticality tasks. The authors presented and evaluated their approach based on Rate Monotonic, but the usage with other priority-based algorithms (e.g., EDF) is also possible.

2.1.2 Dynamic Priority Scheduling

Park and Kim presented Criticality Based Earliest Deadline First (CBEDF) for dual-criticality jobs [PK11]. Their approach divides the schedule into intervals based on the job deadlines. For each interval, the algorithm determines two slack values: the empty slack and the remaining slack. The empty slack is computed offline and determines the available time to execute LO-criticality jobs after reserving the HI-criticality WCETs of all HI-criticality jobs. If a HI-criticality job executes only for its LO-criticality WCET, then the difference between LO-criticality and HI-criticality WCET is added to the remaining slack which is initially equal to zero. Chen et al. showed problems with the schedulability conditions on CBEDF and presented sufficient schedulability conditions [CLTX13]. Furthermore, they improved CBEDF by considering empty slack which can be consumed by HI-criticality jobs under specific conditions.

Ekberg et al. presented an EDF-based approach to bound and shape the demand to schedule dual-criticality sporadic tasks [EY12, GESY11]. The authors set artificial (shorter) deadlines to change the demand of sporadic tasks aiming at a better schedulability. By setting shorter deadlines HI-criticality tasks are more likely to be scheduled before LO-criticality tasks. When the system changes its state from LO to HI, LO-criticality tasks are dropped and the HI-criticality tasks are scheduled according to their original deadlines.

Mollison et al. showed a two-level hierarchical scheduler for multi-processor systems [MEA+10]. Their method considers periodic tasks with five criticality levels, ranging from A to E. Tasks with criticality level A are scheduled by table-driven scheduling and thus, are assigned statically to a processor. The next lower criticality level B also partitions tasks to a specific processor which are then scheduled by EDF, i.e., tasks with criticality level B are scheduled by Partitioned-EDF (P-EDF). Tasks with criticality levels C and D are scheduled by a Global-EDF (G-EDF) scheduler. The G-EDF scheduler at level C (respectively D) is invoked when there are no schedulers active with higher criticality level. Finally, tasks with criticality level E are scheduled by best effort. As a result, the presented architecture provides temporal isolation between the different criticality levels.

2.2 Time-Triggered Approaches

In the following, we show TT scheduling approaches which feature the benefit of a simpler certification. These schedulers create one or several schedule tables which represent a constructive proof. As a consequence, only the correctness of the schedule tables have to be proven instead of proving the correctness of all possible scheduling decisions in all possible situations in case of an ET scheduling approach.

Baruah and Fohler presented an FPS scheduling approach to create schedule tables for dual-criticality systems as a proof-of-concept [BF11]. They use the Audsley approach to assign priorities to TT jobs. The priority ordering is then used to create two schedule tables; one table under LO-criticality and one under HI-criticality assumptions. In Section 5.6.1, we show the approach in more detail and use it as a reference for our schedule table generation method.

Socci et al. showed a dual-criticality algorithm to transform priority assignments into TT schedule tables [SPB13]. Based on two priority assignments, one for LO-criticality behavior and one for HI-criticality behavior, one schedule table is generated for each behavior. Furthermore, the authors prove that this single schedule table per mode dominates the given priority assignment, i.e., fixed priority per mode.

Jan et al. considered the problem of scheduling dual-criticality periodic tasks on multiprocessor systems [JZLP14]. Their method is based on linear programming and creates two schedule tables; one for LO-criticality and one HI-criticality behavior. The authors show two proposals to construct the schedule tables: In the first version, the schedule table for HI-criticality behavior is determined and then, based on this, the LO-criticality schedule table is computed. In the second version, HI-criticality jobs are split based on the method, which we will present in Section 5.2 and then both schedule tables are created at the same time.

Chapter 3

Fundamentals

In this chapter, we present terms, notations and the basic concepts used in this thesis. First, we present generic task models including the characteristic parameters of these tasks. Additionally, we show the Vestal model of mixed-criticality tasks. Second, we present a brief classification of scheduling algorithms and the resulting characteristics of schedules which will be used in this thesis.

The task models and the Vestal mixed-criticality model are shown in Section 3.1. Section 3.2 presents the scheduling classification and the schedule characteristics.

3.1 Task Model

In this section, we show task and job models and their characteristics. We start with generic task models and then, we present the commonly used Vestal model for mixed-criticality tasks.

3.1.1 Generic Task and Job Model

Based on the release pattern, tasks can be classified into periodic, aperiodic, and sporadic tasks. *Periodic* tasks [LL73] are released with a constant time interval between two consecutive releases. On the contrary, *aperiodic* tasks [But05] can be released at arbitrary time instants. Typical events for aperiodic task releases are alert conditions, or failures of periodic tasks that must be retried [TL94, Thu93]. Finally, *sporadic* tasks [Mok83] have an arbitrary release time but two consecutive releases are separated by a minimum-interarrival time. Typical events for sporadic task releases are external events, e.g., device interrupts, but with a restriction of the maximum number of events within a time interval and thus, preventing a monopolization of the system [Mok83].

When a task is released, then this is called a *job* or an *instance* of the task.

Periodic tasks consist of an infinite sequence of jobs. Due to the constant time interval between two releases, release times of all instances are known and hence, this represents a *time-triggered (TT)* task.

Additionally, aperiodic tasks can consist of one single job or a sequence of jobs, whereas in both cases, the release time is only known when the job is released at run-time. From scheduling point of view, there is no difference whether an aperiodic job is the only release of an aperiodic task or whether it is one job of a sequence of releases of an aperiodic task. As a result, we can use the terms aperiodic task and aperiodic job interchangeably for a job with an unknown release time.

Finally, sporadic tasks also consist of an infinite sequence of jobs. The actual release time of jobs is unknown but with a minimum time between two consecutive jobs. Due to the unknown release time of aperiodic and sporadic tasks, these tasks (and their jobs) are *event-triggered (ET)* tasks.

In our schedulers presented in this thesis, we refer to jobs with a known release time as TT or offline jobs. On the contrary, jobs with unknown release time, i.e., jobs of aperiodic or sporadic tasks, are referred to as ET or online jobs. If a scheduler works on job basis, there is no difference whether a job is part of a sequence of jobs or just a single release.

In the following, we show the parameters that characterize tasks and jobs.

Periodic Tasks. A periodic task τ_i is characterized by the tuple $\tau_i = \langle C_i, T_i, D_i \rangle$ with

- C_i: worst-case execution time (WCET), i.e., upper bound on the execution time;
- T_i: period, i.e., time interval between two consecutive releases;
- D_i: relative deadline, i.e., maximum time to execute C_i time units;

Further (optional) parameter are

- Φ_i: offset (also phase), i.e., delay until the first job is released;
- p_i: priority, i.e., assigned priority for fixed priority scheduling (FPS);

If the offsets of all tasks in a task set are equal to zero, then this task set is called *synchronous*. The critical instant for a scheduler, i.e., the situation which represents the worst-case release to schedule occurs when the task set is synchronous [LL73]. If no task offsets are given, we assume them to be equal to zero. A job $\tau_{i,j}$ of a periodic task τ_i is released at $r_{i,j} = \Phi_i + (j-1)T_i$ with $j \in \{1, 2, ..., \infty\}$. As a consequence, the absolute deadline of a job $\tau_{i,j}$ results in $d_{i,j} = r_{i,j} + D_i = \Phi_i + (j-1)T_i + D_i$ with $j \in \{1, 2, ..., \infty\}$. Periodic tasks can have *precedence constraints*, i.e., a periodic task can only be executed if a specified other periodic tasks has completed its execution before [Foh97, CSB90]. A precedence constraint between tasks τ_1 and τ_2, i.e., τ_2 can only be executed after τ_1, is represented by $\tau_1 \prec \tau_2$.

TT jobs. If we work on job basis, we refer to TT jobs by the tuple $J_i = \langle C_i, r_i, d_i \rangle$ with

- C_i: WCET of the job;
- r_i: release time of the job;
- d_i: absolute deadline of the job;

An optional parameter is the priority p_i of a job. Furthermore, precedence constraints can also be set between TT jobs.

Sporadic Tasks. A sporadic task $\tau_{\mathrm{S}i}$ is characterized by $\tau_{\mathrm{S}i} = \langle C_{\mathrm{S}i}, T_{\mathrm{S}i}, D_{\mathrm{S}i} \rangle$ with

- $C_{\mathrm{S}i}$: WCET, i.e., upper bound on the execution time;
- $T_{\mathrm{S}i}$: period (also minimum inter-arrival time), i.e., minimum time interval between two consecutive releases;
- $D_{\mathrm{S}i}$: relative deadline, i.e., maximum time to execute $C_{\mathrm{S}i}$ time units;

The actual release time is unknown until a job of the sporadic task is released in the system at runtime. The absolute deadline of a job is set $D_{\mathrm{S}i}$ time units after its actual release time. In the worst case, a sporadic task behaves like a periodic task [ABRW92].

Aperiodic tasks and ET jobs. If we work on a job basis, there is no difference in the behavior of the scheduler whether a job with unknown release time is an ET job, a single aperiodic job, or a job from a sequence of jobs of an aperiodic task. We characterize these jobs by the tuple $J_{\mathrm{A}i} = \langle C_{\mathrm{A}i}, r_{\mathrm{A}i}, d_{\mathrm{A}i} \rangle$ with

- $C_{\mathrm{A}i}$: WCET of the job;
- $r_{\mathrm{A}i}$: release time of the job;
- $d_{\mathrm{A}i}$: absolute deadline of the job;

Relative deadlines can be classified into implicit, constraint, and arbitrary deadlines. First, *implicit deadlines* are equal to the period of periodic and sporadic tasks. Implicit deadlines allow to simplify the task parameters such that we only need to set a period instead of period and relative deadline. Second, *constraint deadlines* allow relative deadline values to be smaller than or equal to the period of periodic and sporadic tasks. Finally, *arbitrary deadlines* can also be larger than the period of a task. The consequence of arbitrary deadlines is that there can be several jobs of one task active at the same time.

The time interval between release and and absolute deadline is called the *execution window* of a job.

The *hyper-period* H is calculated by the least common multiple of all task periods in the task set. The execution sequence of a synchronous periodic task repeats itself after this hyper-period. Hence, if we can schedule a task set until the hyper-period, we can also schedule it for an infinitely long time.

The *utilization* of a periodic task determines the fraction of processor time spent to execute the task. Equation (3.1) shows the calculation of the utilization of a periodic task τ_i. Due to the fact that sporadic task behave in the worst case like periodic tasks, thus, we can analogously calculate the worst-case utilization of sporadic tasks in Equation (3.2). The utilization of a task set is calculated by the sum of task utilizations, which is shown in Equation 3.3 for a periodic task set with n tasks.

$$\text{utilization of a periodic task:} \qquad U_i = \frac{C_i}{T_i} \tag{3.1}$$

$$\text{worst-case utilization of a sporadic task:} \qquad U_{\mathrm{S}i} = \frac{C_{\mathrm{S}i}}{T_{\mathrm{S}i}} \tag{3.2}$$

$$\text{utilization of a periodic task set:} \qquad U = \sum_{i=1}^{n} U_i \tag{3.3}$$

The *demand* determines the amount of execution time which has to be processed within a time interval. The demand $g(t_1, t_2)$ summarizes the WCETs of all jobs with release time later than t_1 and deadline before t_2. Baruah et al. showed that it is sufficient to check intervals starting from 0 until all absolute deadlines to check whether a synchronous periodic task set with constraint deadlines can be scheduled [BRH90]. In other words, the demand of an interval starting at 0 with length L, $g(0, L)$, must be smaller than or equal to the length of the interval. To calculate the demand in this interval, we have to summarize the WCETs of jobs with execution window within this interval. Inequality (3.4) shows this condition for a periodic task set with n tasks.

$$g(0,L) \leq L \qquad \forall L \in \{d_{i,j} | d_{i,j} \leq \min(L^*, H)\} \tag{3.4}$$

$$\text{with} \qquad g(0,L) = \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i$$

$$\text{and} \qquad d_{i,j}\text{: all absolute deadlines of jobs of all periodic tasks}$$

$$\text{and} \qquad L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}$$

In the shorter version $g(t)$, we refer to the demand between 0 and at time t of jobs with release time and deadline less than or equal to t which is also applicable for TT jobs.

Eventually, we can classify tasks and jobs by the consequences of missing timing constraints. First, *soft* tasks and jobs violating timing constraints do not jeopardize correct system behavior and thus, are usually not characterized by a deadline. Second, *firm* tasks and jobs also do not jeopardize correct system behavior in case of a missed deadline but are completely useless for the system. Finally, *hard* tasks and jobs can jeopardize the correct behavior of the entire system in case of a missed deadline. We can guarantee periodic tasks and TT jobs and hence, implementation of hard periodic tasks and TT jobs is no problem. Further, sporadic tasks can be guaranteed at design time with a worst-case guarantee test. On the contrary, aperiodic tasks and jobs cannot be guaranteed due to their unrestricted releases and thus, only soft and firm aperiodic tasks (and jobs) are reasonable to implement.

3.1.2 Vestal Mixed-Criticality Model

Vestal presented a task model for mixed-criticality tasks [Ves07]. The target area of the task model is a system with different tasks performing safety-critical and non-safety-critical functions on the same nodes. Vestal presented the model based on periodic tasks and extended their parameters by the *criticality level* of each task. As a result of this, tasks are also defined by several WCETs instead of only one. WCETs are assigned for each criticality level up to the criticality level of tasks and jobs. For higher criticality levels, there is no WCET assigned or the WCETs are set to the WCET of the highest assigned criticality level.

As a result, in a mixed-criticality system with k criticality levels, a periodic mixed-criticality task is characterized by the tuple $\tau_i = \langle \chi_i, C_i(1), ..., C_i(k), T_i, D_i \rangle$ with

- χ_i: criticality level of the job;
- $C_i(j)$: WCET with criticality level j and $j \in \{1, 2, ..., k\}$;
- T_i: period of the task;
- D_i: relative deadline of the task;

For TT jobs, we can replace the period and the relative deadline by the release time and absolute deadline of the job. As a result, TT mixed-criticality jobs are characterized by the tuple $J_i = \langle \chi_i, C_i(1), ..., C_i(k), r_i, d_i \rangle$. The WCETs of a task or job are monotonically increasing for increasing criticality level. Typically, in dual-criticality systems, i.e., systems with two criticality levels, criticality levels LO and HI for low and high criticality levels are used.

3.2 Scheduling

In this section, we present a general classification of scheduling algorithms. Furthermore, we show characteristic properties of schedules.

3.2.1 General Scheduling Classification

Scheduling algorithms can be classified according to the priority determination scheme, i.e., the method which determines the order in which the scheduler selects the next executing job. On the one hand, there is fixed task priority scheduling, mostly referred to as *fixed priority scheduling (FPS).* In FPS each task is assigned exactly one priority value which does not change over the lifetime of the system. This priority is assigned to each job of the task. An example algorithm of this category is Rate Monotonic (RM) which assigns the highest priority to the task with the shortest period. On the other hand, there is dynamic task priority scheduling, which is mostly referred to as *dynamic priority scheduling (DPS).* We can divide DPS into two sub-categories: fixed job priority scheduling and dynamic job priority scheduling. A fixed job priority scheduler assigns one constant priority to each job of a task, i.e., the priority of the task changes but an assigned priority of a job does not change. An example algorithm is Earliest Deadline First (EDF) which uses the absolute deadline of a job as priority of the job. On the contrary, in dynamic job priority scheduling, the priority of a job can change over time. As an example, the Least Laxity First (LLF) scheduler assigns the highest priority to the job with the smallest laxity value, i.e., the smallest difference between absolute deadline and remaining execution time.

Another classification is based on the criterion that triggers the invocation of the scheduler. The invocation of the scheduler can be *time-triggered (TT)* or *event-triggered (ET).* A TT scheduler is invoked at predetermined time instances. For instance, a TT *schedule table* stores the time instances when one job stops its execution and the scheduler dispatches the next job for execution. To use a TT scheduler, all release times of jobs have to be known at design time. On the contrary, an ET scheduler is invoked at predefined events, e.g., at release of a job or when a job completes its execution. Based on the scheduler activation, the decision mode of a scheduler is determined. On the one hand, once a job started its execution, it can run until its completion which is called *non-preemptive* scheduling, e.g., [JSM91, BC06]. On the other hand, if the scheduler can preempt the execution of a job to schedule another job, the scheduler is called *(fully) preemptive.* Further, it is possible that the preemption of a job is only allowed at specific locations (preemptions points). In this case, the scheduler is *partially preemptive.*

In this thesis, we present different scheduling algorithms which are based on *slots*. Scheduling decisions with the granularity of slots corresponds to partially preemptive scheduling, i.e., we can take scheduling decisions only at the beginning of a slot. Within each slot, the scheduler is executed and then the selected job is executed. During system design, slot sizes have to be determined. We can calculate the minimum time a job can execute in a slot based on the worst-case time the scheduler needs to execute and the slot length. Figure 3.1 shows the definition of slots. The maximum execution time of the scheduler is represented by t_{sched}. As a result, we can guarantee a minimum execution time for the selected job of t_{exec}. At runtime, the scheduler execution time varies based on the actual system state. As a consequence, there can be more time to execute the selected job which is also shown in the figure.

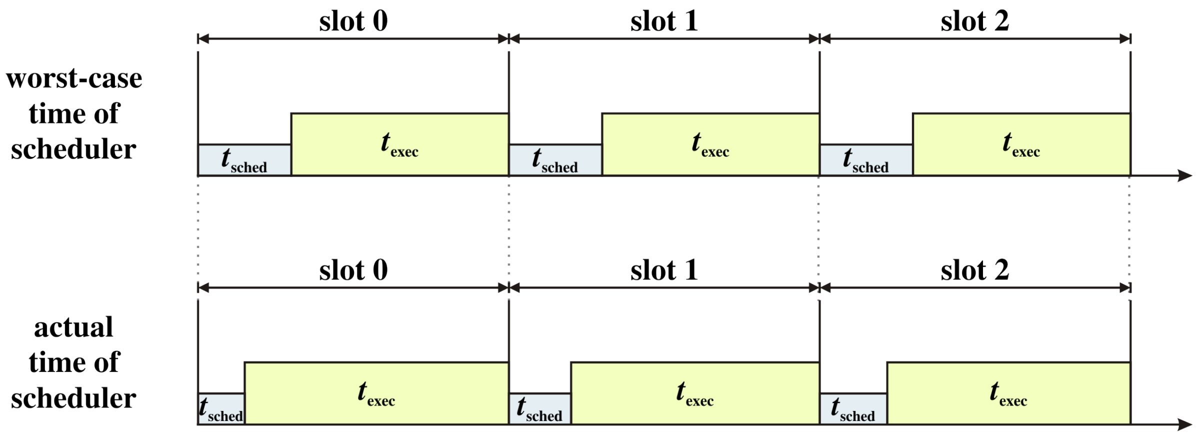


Figure 3.1: *Definition of a slot with scheduler execution within slots.*

For completeness, Figure 3.2 shows the problems if we want to define a slot with constant execution times of the jobs within a slot. As a reference, we show slots when the schedulers always executes for its worst-case time. On the one hand, we can synchronize [Lam78, KO87] the start of slots such that we compensate shorter actual execution times of the scheduler by inserting idle time. As a result, we obtain deterministic start times of the slots. On the other hand, if we directly start the execution of the selected job after the scheduler execution, the start times of slots drift apart from the reference and hence, are not deterministic anymore. The scheduler execution outside of slots results either in inclusion of idle time or non-deterministic slot start and end times. For this reason, the slot model should be defined with scheduler execution within the slots. As a result, we can guarantee a minimum time to execute jobs and execute the jobs longer when the scheduler executes for less than its worst-case execution time.

When implementing a slot-based scheduler, using release times, deadlines, and WCETs based on slots simplifies scheduling. The transformation into slot-based time values can be done as follows: release times are set to the start time of the next slot after the actual release time. Additionally, deadlines are set to the start time of the slot just before the deadline. The WCETs are divided by the guaranteed execution time t_{exec} within one slot to determine the number of slots needed to execute the job. Non-integer slot numbers

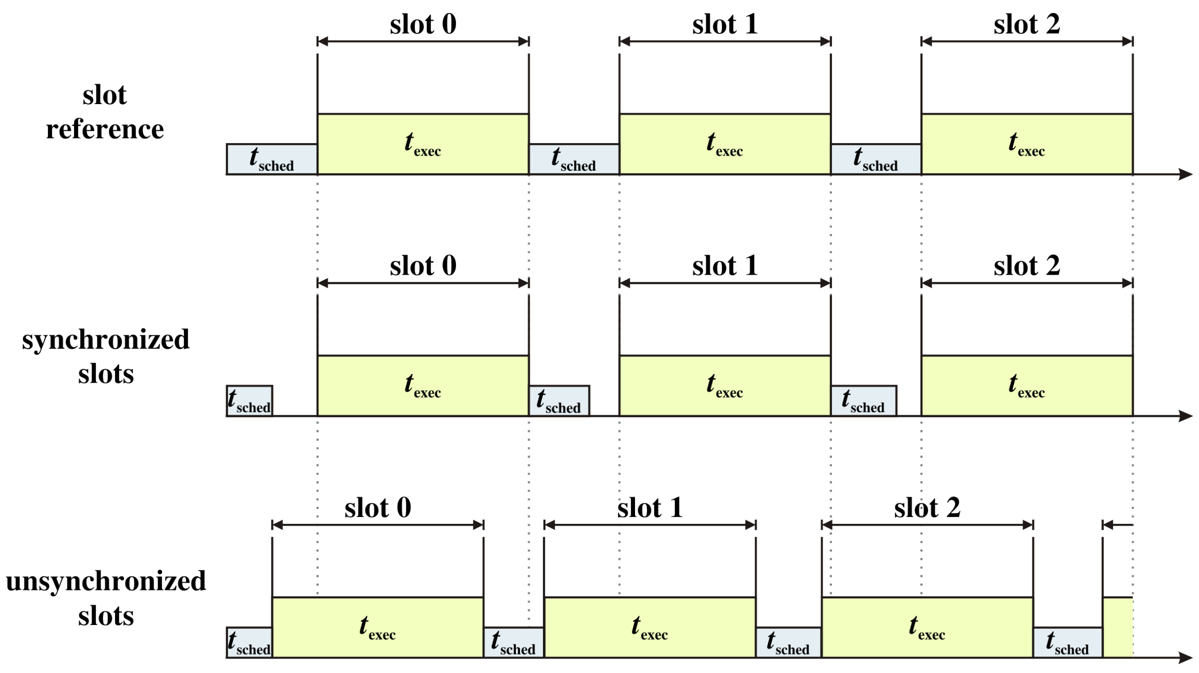


Figure 3.2: *Definition of a slot with scheduler execution outside the slots.*

are rounded to the next full integer number of slots. If a job finishes execution within a slot, we cannot start the slot earlier because this results in non-deterministic slot start times.

In this thesis, we refer to time instances in the schedules with the variable t. Furthermore, when using time intervals of several slots, an interval is defined as follows: $[t_{start}, t_{end})$ whereas t_{start} and t_{end} are multiples of the length of a slot.

So far, we considered schedulers to create a schedule for a single processing unit (processor). Besides this uni-processor scheduling problem, systems with more than one processor need to allocate tasks and jobs to the processors. Such systems are called *multi-processor* systems. In the earlier years, they were implemented as *distributed systems*, i.e., systems in which each processor is implemented as a single hardware chip. Nowadays, multi-processor systems are mostly implemented using multi-core chips, i.e., a single chip contains more than one processor which is usually called "core". Schedulers for multi-processor systems can be classified into partitioned, global, and hybrid approaches. *Partitioned* scheduling algorithms allocate tasks to processors and on each processor a single-processor scheduling algorithm is used. As a result, tasks and jobs cannot migrate from one processor to another. On the contrary, *global* scheduling algorithms can schedule all tasks and jobs on an arbitrary processor in the system. Hence, jobs can migrate between all processors at every time instant which is called fully migrative. In *hybrid* approaches, it is possible to restrict migrations. For instance, once a job of a task started execution on a processor, it cannot migrate anymore, but other jobs of the same task can be assigned to a different processor. Another example is the class of cluster-based algorithms. These algorithms assign tasks to a sub-set of processors in which these tasks and jobs are fully migrative.

To conclude the brief classification of schedulers, we mention two types of schedulers which we will use in this thesis.

Scheduling can be represented by a search tree [Kor85]. Each scheduling decision corresponds to an edge in the tree. As a consequence, each node represents the partial schedule based on the scheduling decisions so far. Leaf nodes represent complete schedules. Tree search schedulers are TT schedulers, e.g., [FK90, BNDPS02]. A benefit of this approach is that we can change already taken scheduling decisions to improve the schedule. A common way to do this is backtracking. We present this approach in more detail in Chapter 5.

We can use mode change schedulers, e.g., [Foh93, PB98], to accommodate for different requirements. At design time, we can create schedule tables for the different modes. At runtime, the scheduler schedules the jobs according to the schedule table of the current mode. If the system needs to change its behavior, we can change the mode and use a different schedule table. Our scheduling algorithms presented in Chapters 5 and 7 are based on these mode changes.

3.2.2 Schedule Characteristics

Scheduling algorithms must result in schedules that meet the timing constraints of all tasks and jobs. We call these schedules feasible.

Definition 3.1. Feasible Schedule
A schedule in which all tasks and jobs execute for their defined worst-case execution time within their execution window is called a *feasible schedule.* $\triangleleft$

Not all task sets and job sets can result in feasible schedules. A trivial example is a task set which only consists of one periodic task with a WCET which is longer than its relative deadline. As a result, if it is possible to find a feasible schedule, then the task set (job set) is called feasible.

Definition 3.2. Feasible Task (Job) Set
A task (job) set, for which a feasible schedule exists, is called *feasible task (job) set.* $\triangleleft$

If we use a specific scheduler with a feasible task (job) set and the scheduler can create a feasible schedule, the task (job) set is called schedulable.

Definition 3.3. Schedulable Task (Job) Set
A task (job) set which can be scheduled by a specific scheduling algorithm is called a *schedulable task (job) set.* $\triangleleft$

Based on the implemented scheduling algorithm, specific characteristics of the schedule and the tasks can be derived. An important characteristic is the *worst-case response time (WCRT)* of tasks [JP86]. The response time of a job of a task is the time interval between its release time and the point in time when the job completes its execution. The WCRT determines the longest possible response time of a task. We will analyze the response times of tasks in Chapter 4.

Schedulability tests are used to test the schedulability of task sets. We can group the schedulability tests in to three groups: sufficient, necessary, and exact tests. Task sets that pass a *sufficient* schedulability test are definitely schedulable. If the task set does not pass the test, the sufficient test cannot determine whether the task set is schedulable or not. Furthermore, failing a *necessary* schedulability test proves that the task set is definitely not schedulable whereas passing the test means that the tested task set can be schedulable or not. Additionally, schedulability tests which are necessary and sufficient are *exact* tests. Passing an exact schedulability test proves the schedulability of the task set. On the contrary, failing the exact test proves that the task set is definitely not schedulable.

Typical exceptions that can be detected during task and job executions are deadline misses and overruns. A deadline miss occurs when a job does not finish execution before its deadline. A job overruns if the job does not complete its execution within its WCET and requests more computation time after that. In a correctly working system, both exceptions should never occur.

Chapter 4

Task Parameter Transformation

In this chapter, we describe a proactive method for handling sporadic tasks with standard offline scheduling in time-triggered (TT) systems without changing the schedule table at runtime. The method is based on reservation tasks which are included into the offline scheduling process to guarantee the execution of sporadic tasks at runtime. In the schedule table, we reserve execution time for the sporadic task instances such that at runtime, independent of the unknown arrival time of sporadic task instances their deadlines can be met.

The transformation of sporadic tasks into periodic tasks with implicit deadlines is independent of the applied offline scheduling algorithm. This transformation at design time allows for a simplified guarantee and certification before start of the system because the sporadic tasks can be guaranteed already in the offline scheduling process. Additionally, the transformation allows for an minimization of reserved utilization and a reduction of the worst-case response times (WCRTs) of sporadic tasks.

Further, we apply the presented transformation method on periodic tasks to evaluate the efficiency and effectiveness of the method. We calculate new periods and worst-case execution times (WCETs) for the periodic tasks aiming at a minimization of the resulting hyper-period of the periodic task set.

We start with a brief description of prior work in the field of handling non-periodic tasks and optimization of task parameters in Section 4.1. Next, we present our transformation method to create periodic reservation tasks to guarantee sporadic tasks already in the offline scheduling process in Section 4.2. In Section 4.3, we analyze our method with respect to the needed utilization of reservation tasks to guarantee sporadic tasks. Further, we show an extension of the transformation to optimize the WCRTs of sporadic tasks in Section 4.4. As a result of the extension, we show the trade-off between the optimization of the utilization and the WCRT in Section 4.5. An example in Section 4.6 illustrates how the method works. Finally, Section 4.7 presents evaluation results based on applying the presented method on periodic tasks to minimize the length of the hyper-period.

4.1 Prior Work

Safety-critical applications are often represented by periodic tasks [LL73]. Using periodic tasks, all parameters, e.g., release times, deadlines, and worst-case execution times (WCETs), are known before runtime, i.e., at design time. At runtime, the occurrence of aperiodic and/or sporadic tasks is possible which is caused by for instance interrupts [Mok83]. We can classify methods to handle aperiodic and sporadic tasks into two groups: offline and online methods. On the one hand, offline methods include knowledge about worst-case behavior of sporadic tasks. These methods are *proactive* approaches. On the other hand, online methods react on arrivals of aperiodic and sporadic tasks at runtime which represents *reactive* approaches. In the following, we give a brief overview of offline and online methods with their advantages and disadvantages.

The simplest reactive approach is using background scheduling for aperiodic and sporadic tasks. In this approach, aperiodic and sporadic tasks are only scheduled if there are no jobs of periodic tasks ready to run. The major advantage of this approach is a very simple implementation with the guarantee that non-periodic tasks do not interfere with periodic tasks. The obvious drawback is the possibly extremely long response times and also there is no guarantee of actually being scheduled. A common approach to overcome these drawbacks are server-based algorithms. Server algorithms are available for fixed priority scheduling (FPS) and dynamic priority scheduling (DPS).

Lehoczky et al. presented a simple server called Polling Server (PS) based on FPS [LSS87]. The server task is characterized by its budget C_S and its period T_S. At runtime, when the server becomes active, i.e., it would be selected for execution according to its priority, it checks whether there is an aperiodic or sporadic task to execute. If there is a non-periodic task ready to execute, the task is served with the current budget of the server and the budget is replenished at next server period. Else the budget is completely dropped and replenished at next server period. The advantage of this method is that the simplicity of the mechanism and it improves the response times of aperiodic and sporadic tasks. On the contrary, capacity is lost if there is no aperiodic or sporadic task ready to execute when the server becomes active. Hence, the response times of aperiodic and sporadic tasks can increase when the budget is dropped.

To overcome the disadvantage of the lost capacities, Lehoczky et al. also presented the FPS-based Priority Exchange Server (PES) [LSS87]. This server algorithm exchanges capacity at server priority level for capacity at priority level of periodic tasks. By doing this, the server does not immediately drop its capacity if there is no aperiodic or sporadic task ready to execute. If there is no task ready to execute, we still lose capacity.

Example. We show the behavior of this algorithm using the following example task set shown in Table 4.1.

Figure 4.1 shows the resulting schedule applying FPS with PES. On top, the execution of aperiodic tasks is shown; thereunder the server task τ_S and the periodic tasks with server capacity at the corresponding priority level are shown.

At the beginning of the schedule, there is no aperiodic task ready to execute. The capacity of the server at highest priority level is decreased while capacity at the priority level of the executing task τ_1 is increased by the same amount. After τ_1 finished execu-

periodic tasks	WCET	period	priority
τ_1	2	8	2
τ_2	1	12	3
τ_S	3	6	1

aperiodic tasks	WCET	release time
τ_{A1}	3	4
τ_{A2}	5	9

Table 4.1: *Task set for PES example.*

tion, τ_2 starts execution using the highest priority capacity which is the server priority and generates capacity at its own priority level. Due to the fact that there is no task ready to execute capacity is lost at the highest priority level with available capacity.

At $t = 4$, an aperiodic task τ_{A1} arrives in the system and is served with the highest priority capacity available, i.e., first at priority of τ_1, then at priority of τ_2, and finally at server priority. At $t = 9$, a second aperiodic task τ_{A2} arrives and is executed for one time unit using the available capacity. After that, there is no available capacity for serving aperiodic tasks. Hence, periodic tasks are executed until the server capacity is replenished at $t = 12$ when the next server instance is released. This procedure is repeated until the aperiodic task finishes its execution. ◊

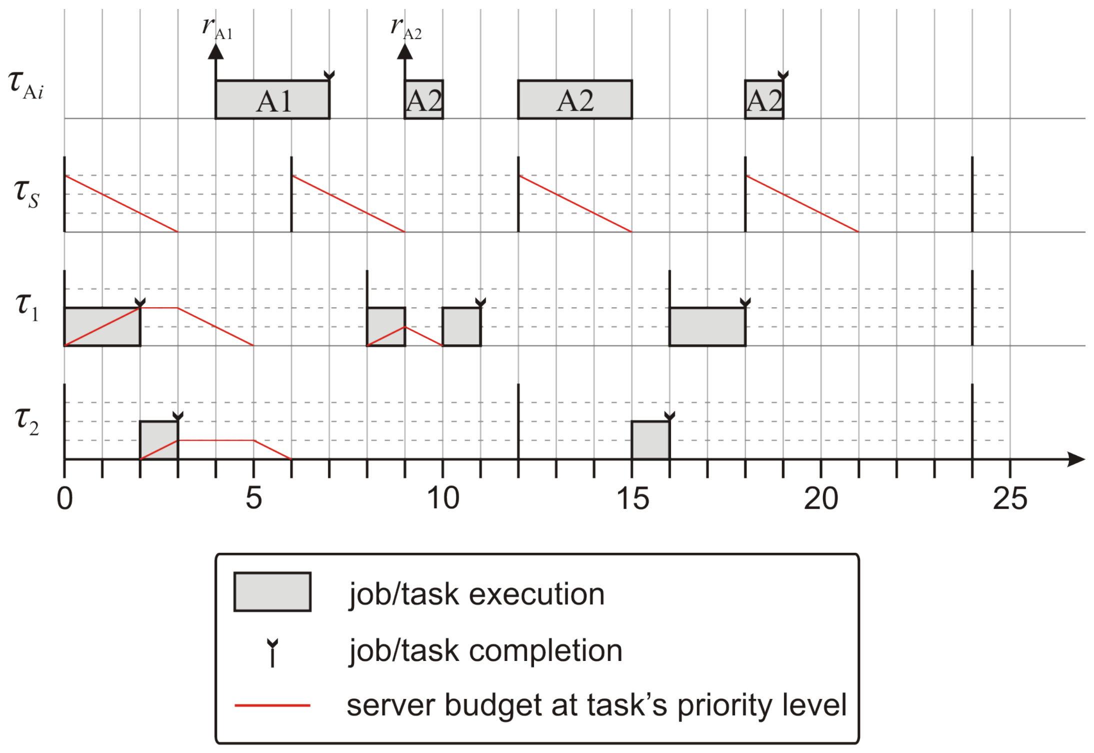


Figure 4.1: *Schedule for task set shown in Table 4.1 using PES.*

PES provides better response times than PS at cost of higher scheduling overhead. The disadvantage of this method is the complex implementation and capacity management.

Sprunt presented the Sporadic Server (SS) which is an FPS server overcoming the drawback of the complex capacity management of PES [SSL89, Spr90]. This server preserves its capacity at its own high priority level. Server capacity is restored when the server becomes active. When the server becomes idle or capacity is exhausted, the amount of capacity is calculated which will be increased at so-called "replenishment times" which are based on the server period. SS provides fast response times without jeopardizing deadlines of periodic tasks. Additionally, capacity is not lost and it has a lower overhead than PES. Saewong et al. showed how to determine the critical instant of the sporadic server replenishment and a schedulability analysis [SRLK02].

We refer the interested reader to [DB05], where Davis and Burns showed an FPS server performance analysis. As a result of the shown approaches before, the question arises how to determine the parameters budget and period for server tasks. Lipari and Bini answered the question on how to determine the "best" pair (budget, period) to schedule groups of tasks in [LB03]. They assumed a hierarchical scheduler with periodic servers, e.g., PS or SS, defined by the server budget Q_S and the server period T_S. Each server is assigned a set of tasks. The method aims at minimizing the needed utilization for these servers. The authors transformed the problem into (α, Δ)-space, i.e., $\alpha = \frac{Q_S}{T_S}$ and Δ is the maximum delay in the time slot distribution. The obtained values of the (α, Δ)-pair is then transformed back to server budget Q_S and server period T_S.

In [LBA98], Lipari et al. showed a two level DPS scheduler. They use one server for each application where target applications are soft real-time and multimedia applications. Instead of exact execuction times and inter-arrival times, they make of use "desired" inter-arrival times and reservations of a bandwidth for each application. Each job is assigned a deadline and put into single EDF queue.

Spuri and Buttazzo presented the Total Bandwidth Server (TBS) [SB94]. This EDF-based DPS server is characterized by its utilization $U_S = \frac{C_S}{T_S}$. On arrival of an aperiodic task or job of a sporadic task, a deadline is assigned based on the server utilization and the WCETof of the aperiodic (sporadic) task. The assigned deadline for the k-th aperiodic request is calculated according to the following equation:

$$d_{\mathrm{A}k} = max\left(r_{\mathrm{A}k}, d_{\mathrm{A}k-1}\right) + \frac{C_{\mathrm{A}k}}{U_S} \tag{4.1}$$

The server bandwidth until the deadline of the request is assigned to this aperiodic task and succeeding aperiodic requests are assigned the bandwidth available after this deadline.

Example. In the following, we show the behavior of TBS using the same example task set as used for PES which is shown in Table 4.2. TBS is characterized by its utilization which we calculate by $U_S = \frac{C_S}{T_S} = \frac{3}{6} = 0.5$. Figure 4.2 shows the resulting schedule serving aperiodic requests with TBS. The selection function of the scheduler works according to the earliest deadline first rule.

periodic tasks	WCET	period
τ_1	2	8
τ_2	1	12
τ_S	3	6

aperiodic tasks	WCET	release time
τ_{A1}	3	4
τ_{A2}	5	9

Table 4.2: *Task set for TBS example.*

When the first aperiodic request arrives at $t = 4$, a deadline of $d_{A1} = 10$ is assigned and is executed according to this deadline. Before the deadline of the first aperiodic task, another aperiodic task τ_{A2} arrives in the system. Although the first aperiodic task is already completely executed, the deadline is calculated based on the available bandwidth of the server after this deadline. This is done because the bandwidth until d_{A1} is already reserved for τ_{A1}. The schedule is then completed by scheduling the tasks according to EDF. ◊

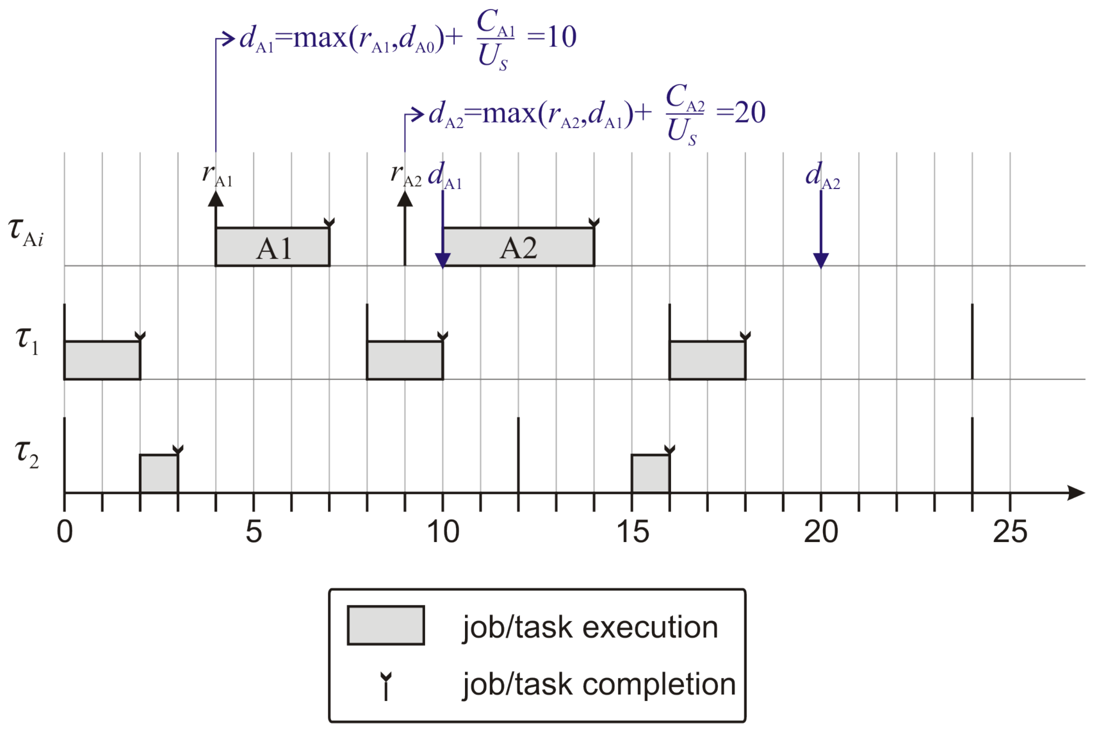


Figure 4.2: *Schedule for task set shown in Table 4.2 using TBS.*

TBS provides good response times for aperiodic and sporadic tasks and the implementation complexity is low.

Buttazzo and Sensini presented an improvement of the TBS called TB* [BS97]. TB* reduces the response times of aperiodic and sporadic requests by an iterative deadline calculation based on the estimated completion by the original deadlines calculated by

TBS. The result is an optimal solution at cost of an higher overhead for the deadline calculation. In their paper, the authors showed that already for few iteration steps TB* provides strong improvements in the response times. This results in a reduced overhead of deadline calculations.

The drawback of TBS and TB* is the fact that the deadline assignment is based on WCET of aperiodic and sporadic tasks. As a consequence, misbehaving, i.e., exceeding their WCET, aperiodic or sporadic tasks jeopardize the execution of periodic tasks. To overcome this problem, Constant Bandwidth Server (CBS) can be used which is shown later in this section.

Fohler et al. presented in [FLB01] how to handle soft aperiodic tasks using TBS. They included TBS into slot-shifting, which we will show in detail in Chapter 6, for soft aperiodic tasks. Before runtime, complex constraints of periodic tasks are resolved according to slot-shifting. The method creates an offline schedule such that the server utilization U_S is reserved for TBS. Then at runtime, slot-shifting, which takes decisions based on EDF, is used to schedule tasks. At the arrival of soft aperiodic tasks, a deadline is assigned based on the TBS bandwidth. The advantage of the method is the reduction in response times of soft aperiodic tasks and the handling of complex constraints, e.g., precedence constraints, of periodic tasks.

Another algorithm using server tasks is the Constant Bandwidth Server (CBS) which overcomes the problem of misbehaving aperiodic tasks affecting periodic tasks [AB98]. Abeni and Buttazzo characterize in their approach a server task τ_S by a maximum budget Q_S and a server period T_S. The server utilization is calculated by $U_S = \frac{Q_S}{T_S}$. At runtime, the currently available budget c_S is calculated and updated. Executing aperiodic tasks consumes this budget. If the currently available budget is exhausted, i.e., is equal to zero, then it is replenished to maximum budget Q_S and the server deadline is extended by one server period.

When the k-th aperiodic task arrives and if the condition

$$c_S \geq (d_{S,k} - r_{\mathrm{A}k})\, U_S \tag{4.2}$$

is fulfilled, the budget is replenished and the server deadline is set to $d_{S,k+1} = r_{\mathrm{A}k} + T_S$. If the condition is not satisfied, the aperiodic task is served with the current budget and deadline.

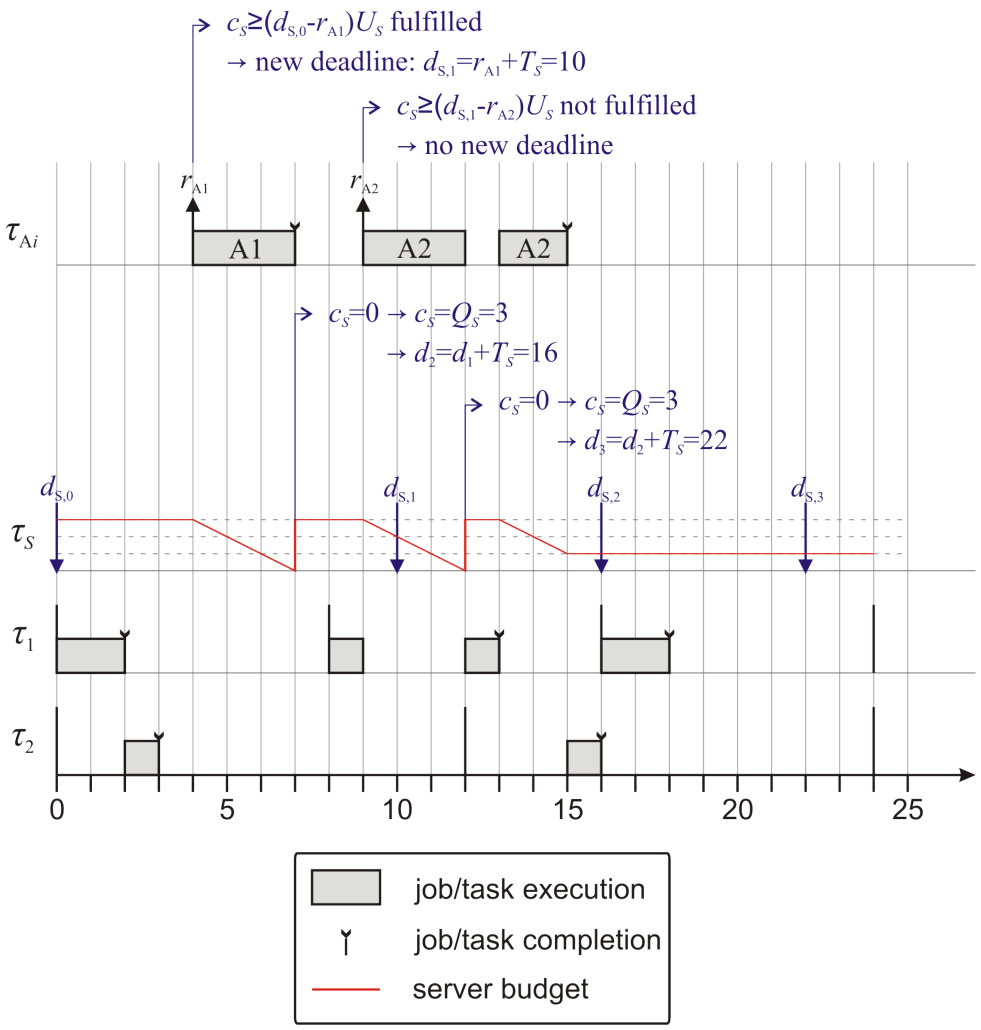
Example. We show the behavior of CBS using the same task set as before (for PES and TBS) which is depicted in Table 4.3.

periodic tasks	WCET / max. budget	period
τ_1	2	8
τ_2	1	12
τ_S	3	6

aperiodic tasks	WCET	release time
$\tau_{\mathrm{A}1}$	3	4
$\tau_{\mathrm{A}2}$	5	9

Table 4.3: *Task set for CBS example.*

Figure 4.3 shows the resulting schedule using EDF with CBS. At system start, the server task is initialized with maximum budget and deadline equal to zero. Periodic tasks are executed according to EDF until at $t = 4$ the first aperiodic request is triggered. We check the condition shown in Inequality (4.2) which is fulfilled. As a consequence, we set the currently available budget to maximum budget $Q_S = 3$ and set the server deadline to the aperiodic task's release time extended by the server period to $d_{S,1} = r_{\mathrm{A1}} + T_S = 4 + 6 = 10$. Aperiodic task τ_{A1} is then executed consuming the budget of the server until the currently available budget is zero, which is in this case when the tasks signals completion. Due to the exhausted budget, the budget is replenished and the server deadline is extended to $d_{S,2} = 16$. The next aperiodic request is triggered at $t = 9$. We check the condition in Inequality (4.2) which is not fulfilled and hence, we do not assign a new deadline but use the existing budget and deadline to serve the aperiodic task. At $t = 12$, the available budget is exhausted and we replenish it and extend the server deadline by one period to $d_{S,3} = 22$. The aperiodic task is served with the budget until its completion according to the server deadline. ◊



$c_S \geq (d_{S,0} - r_{A1}) U_S$ fulfilled
→ new deadline: $d_{S,1} = r_{A1} + T_S = 10$
$c_S \geq (d_{S,1} - r_{A2}) U_S$ not fulfilled
→ no new deadline
r_{A1}
r_{A2}
τ_{Ai}
A1
A2
A2
$c_S = 0 \rightarrow c_S = Q_S = 3$
$\rightarrow d_2 = d_1 + T_S = 16$
$c_S = 0 \rightarrow c_S = Q_S = 3$
$\rightarrow d_3 = d_2 + T_S = 22$
$d_{S,0}$
$d_{S,1}$
$d_{S,2}$
$d_{S,3}$
τ_S
τ_1
τ_2
0
5
10
15
20
25
job/task execution
job/task completion
server budget

Figure 4.3: *Schedule for task set shown in Table 4.3 using CBS.*

CBS shows good response times with a low complexity. Further, misbehaving tasks, i.e., tasks over-running their WCETs, do not jeopardize periodic tasks because CBS enforces its maximum bandwidth even if the actual execution time of aperiodic tasks exceeds their WCET and the actual execution time is unknown and maybe extremely large (or even unbounded).

In safety-critical applications not only the response times of aperiodic and sporadic events is important but also the response times of periodic tasks. Periods and hence, deadlines of tasks with implicit deadlines, affect the response times. Furthermore, higher sampling rates, i.e., shorter periods, can improve quality of results of periodic tasks. Additionally, periods in time-triggered (TT) systems affect the length of the hyper-period which is calculated by the least common multiple (LCM) of all periods in the system. The hyper-period determines the length of the schedule table and hence, a shorter hyper-period results in less memory consumption of the schedule table. In the following, we present methods to adjust task parameters before runtime to fulfill these goals.

Cottet and Babau showed a method based on Rate Monotonic in [CB96]. The authors assumed that all tasks' parameters are pre-determined except for one task. Their method calculates possible periods and deadlines for this one remaining task. As a result, an area with schedulable pairs of period and deadline is obtained and can be used to select period and deadline.

Burns and Davis presented a method for FPS consisting of periodic tasks [BD96]. They consider three different cases to calculate new periods. First, all releases of tasks have the same computation time. Second, task computation times differ but are the same for different releases of the same task. Third, each task has two computation times corresponding to whether an event is observed or not. The method reduces periods iteratively as long as task are schedulable based on a response time analysis. As a result, the method improves response times of periodic tasks and by improving the sampling rate, quality of results can be improved. On the contrary, increasing overheads by shorter periods, and hence, more invocations, are neglected.

So far, methods changed task parameters to optimize the performance of the system. The methods presented in the following focus on accommodating requirements by the hardware of the system, e.g., limited memory for storing the schedule table and limited computational capabilities, i.e., maximum utilization of the task set, of the system.

Brocal et al. assumed in [BBBR11] that periods are given as ranges of integer numbers to minimize the hyper-period. They presented the Fast Hyper-period Search (FHS) algorithm to efficiently minimize the hyper-period. We show the method in Section 4.7.4 in more detail as an application of our transformation method.

Ripoll and Ballester-Ripoll used the rationale of FHS to optimize the hyper-period assuming that periods are given as continuous ranges [RBR12]. In their method, periods are not absolute times values, e.g., a task is invoked every 30 ms, but tasks are invoked x-times within the hyper-period. For instance, a task is invoked $x = 3$ times within a hyper-period of $H = 100$ ms. Thus, the effective period of the task would be $33.\bar{3}$ ms. As a consequence, the scheduler is responsible to determine the exact activation time based on the granularity of time. The method determines overlapping intervals based on multiples of the original period ranges. If all intervals overlap, a minimum hyper-period is found.

Nasri et al. presented a method to select harmonic periods optimizing the system utilization [NFK14]. They assumed that periods are given as ranges. Based on integer multiples of the period ranges, the method searches for covering intervals of multiples of the period ranges to find harmonic periods. By using harmonic periods, the system utilization can be reduced and the hyper-period can be optimized.

4.2 Transformation of Sporadic Tasks

In this section, we show how we transform sporadic tasks into periodic reservation tasks. The goal of the transformation is to integrate reservation tasks into the offline scheduling process such that execution time is reserved for sporadic tasks arriving at runtime at unknown arrival times. In general, offline schedulers need knowledge about all task parameters for the scheduling process. We do not assume any specific offline scheduler. Our method is valid for an arbitrary offline scheduler able to schedule periodic tasks. At runtime, for each sporadic task, independent of its arrival time, there must be enough reserved execution time (RET) such that the task can execute for at least its WCET before its deadline. In contrast to the reactive server algorithms before, our method is proactive, i.e., the method accommodates for the demands of sporadic tasks already at design time. For the reason of simplicity, we present our transformation for the non-mixed-criticality case. Our method is also applicable for mixed-criticality task sets. We show how to adapt the basic method for the mixed-criticality case in Section 4.2.4.

4.2.1 Terms, Symbols, Notation and Assumptions

The set $P = \{\tau_1, \ldots, \tau_n\}$ represents n periodic tasks τ_i. Periodic tasks are characterized by the tuple of task parameters $\tau_i = \langle C_i, T_i, D_i \rangle$ with

- C_i: WCET, i.e., upper bound on the largest possible execution time;
- T_i: period, i.e., time interval between two consecutive releases;
- D_i: relative deadline, i.e., maximum time to complete execution relative to the release of a task instance;

The instances of periodic tasks are released at $r_{i,j} = (j-1)\,T_i$ with $j \in \{1, \ldots, \infty\}$.

The set $S = \{\tau_{\mathrm{S}1}, \ldots, \tau_{\mathrm{S}m}\}$ represents m sporadic tasks τ_i. Sporadic tasks are characterized by the tuple of task parameters $\tau_{\mathrm{S}i} = \langle C_{\mathrm{S}i}, T_{\mathrm{S}i}, D_{\mathrm{S}i} \rangle$ with

- $C_{\mathrm{S}i}$: WCET, i.e., upper bound on the largest possible execution time;
- $T_{\mathrm{S}i}$: period, i.e., *minimum* time interval between two consecutive releases;
- $D_{\mathrm{S}i}$: relative deadline, i.e., maximum time to complete execution relative to the release of a task instance;

Sporadic tasks have arbitrary release times with a period $T_{\mathrm{S}i}$ (also called minimum inter-arrival time of the sporadic task).

For each sporadic task, we create one reservation task. The set $R = \{\tau_{\mathrm{R}1}, \ldots, \tau_{\mathrm{R}m}\}$ represents m reservation tasks $\tau_{\mathrm{R}i}$ with implicit deadlines. Reservation tasks are characterized by the tuple of task parameters $\tau_{\mathrm{R}i} = \langle C_{\mathrm{R}i}, T_{\mathrm{R}i} \rangle$ with

- $C_{\mathrm{R}i}$: RET, i.e., reserved time in the offline schedule which can be used to execute sporadic tasks at runtime;
- $T_{\mathrm{R}i}$: period, i.e., time interval between two consecutive releases of the reservation task;

Reservation task instances are released at $r_{\mathrm{R}i,j} = (j-1)\, T_{\mathrm{R}i}$ with $j \in \{1, \ldots, \infty\}$.

In the following, we do not only show how to transform sporadic tasks into reservation tasks with the goal of including them into the offline scheduling process but also show how to improve the worst-case response time (WCRT) of sporadic tasks.

Definition 4.1. Worst-Case Response Time
The worst-case response time (WCRT) $R_{\mathrm{S}i}$ of a (sporadic) task is the longest possible time between the release $r_{i,j}$ of an (sporadic) task instance and the successful execution of the task instance, i.e., finishing time $f(\tau_{\mathrm{S}i,j})$ [LL73].

$$R_{\mathrm{S}i} = f(\tau_{\mathrm{S}i,j}) - r_{i,j} \tag{4.3}$$

◁

The utilization calculation of periodic tasks with implicit deadlines is shown in Equation (4.4).

$$U_i = \frac{C_i}{T_i} \tag{4.4}$$

The maximum workload of sporadic tasks is produced when they arrive with maximum frequency. In this case, sporadic tasks behave like periodic tasks [ABRW92]. The worst-case, i.e., maximum, utilization of sporadic tasks results then in:

$$U_{\mathrm{S}i} = \frac{C_{\mathrm{S}i}}{T_{\mathrm{S}i}} \tag{4.5}$$

The utilization of the entire (combined) task set $P \cup S$ results in:

$$U = \sum_{i=1}^{n} U_i + \sum_{i=1}^{m} U_{\mathrm{S}i} \tag{4.6}$$

After the transformation, the utilization of the task set for the offline schedule consists of the periodic tasks and the (periodic) reservation tasks, i.e., $P \cup R$:

$$U_{\text{offl. schedule}} = \sum_{i=1}^{n} U_i + \sum_{i=1}^{m} U_{\mathrm{R}i} \tag{4.7}$$

We do not assume any specific scheduler to generate the offline schedule table. Further, we assume that sporadic tasks have to be fully preemptive, because several reservation instances could be needed for runtime execution. The offline scheduler can be non-preemptive or preemptive.

4.2.2 Worst-Case Arrival of Sporadic Tasks

We assume deadlines of sporadic tasks are equal to the minimum inter-arrival time. For sporadic tasks with shorter deadlines, because we transform the sporadic tasks based on their execution window, the transformation can be adapted by using the deadline instead of the minimum inter-arrival time for the calculation of the reservation task parameters. We show the transformation based on deadlines which are equal to the minimum inter-arrival time for the reason of simplicity of the explanation but the method can also be applied to sporadic tasks with deadlines shorter than the minimum inter-arrival time. Guaranteeing sporadic tasks with $D_{\mathrm{S}i} \leq T_{\mathrm{S}i}$ will result in more reserved utilization than for sporadic tasks with $D_{\mathrm{S}i} = T_{\mathrm{S}i}$.
Within the execution window of every sporadic task $\tau_{\mathrm{S}i} \in S$, we reserve at least $C_{\mathrm{S}i}$ (split into $k_{\mathrm{U}i}$ slices) time units. A slice of a reservation task is the amount of execution time reserved within one period of the reservation task. The sum of $k_{\mathrm{U}i}$ slices is equal to the WCET of the sporadic task. This reservation has to be done for every possible release time of a sporadic task instance within the schedule independent of the scheduling pattern of the reservation task. If we can guarantee a sporadic task instance for its worst-case arrival, then we can also guarantee it in all other cases.

Figure 4.4 shows worst-case arrival: a sporadic task instance of $\tau_{\mathrm{S}i}$ arrives directly after the scheduled RET $C_{\mathrm{R}i}$. This RET before the release of the sporadic task instance is scheduled at the beginning of its period and thus, the time until the start of the next period ($T_{\mathrm{R}i} - C_{\mathrm{R}i}$) is maximum. This time interval until the next period elapses without execution of the sporadic task instance. All further slices are scheduled at the end of their periods such that the instances of the reservation task are scheduled as late as possible and the execution of the sporadic task instance completes just before its deadline. For the execution of the sporadic task instance, there are $k_{\mathrm{U}i}$ slices and hence, $k_{\mathrm{U}i}$ periods, plus the time interval $T_{\mathrm{R}i} - C_{\mathrm{R}i}$ without any execution needed. This time interval must not exceed the period, i.e., deadline, of the sporadic task which is shown in Equation (4.9a).

4.2.3 Transformation Method

For the transformation, we assume that the sporadic tasks arrive with maximum frequency and hence, behave like periodic tasks. Furthermore, for reasons of simplicity of representation, deadlines of sporadic tasks are assumed to be equal to the minimum inter-arrival time. For sporadic tasks with deadlines shorter than minimum inter-arrival time, the minimum inter-arrival time $T_{\mathrm{S}i}$ has to be replaced by $D_{\mathrm{S}i}$ in all equations. Assuming the worst-case arrival in Figure 4.4, we calculate the parameters of the periodic reservation tasks $\tau_{\mathrm{R}i}$. In the following, we set implicit deadlines ($D_{\mathrm{R}i} = T_{\mathrm{R}i}$) for all reservation tasks. The periodic reservation tasks are characterized by the tuple: $\tau_{\mathrm{R}i} = \langle C_{\mathrm{R}i}, T_{\mathrm{R}i} \rangle$, whereas we call the execution time $C_{\mathrm{R}i}$ of such reservation tasks RET. The RET $C_{\mathrm{R}i}$ of the reservation task is calculated according to Equation (4.8). We choose the period of the reservation task such that the time without execution

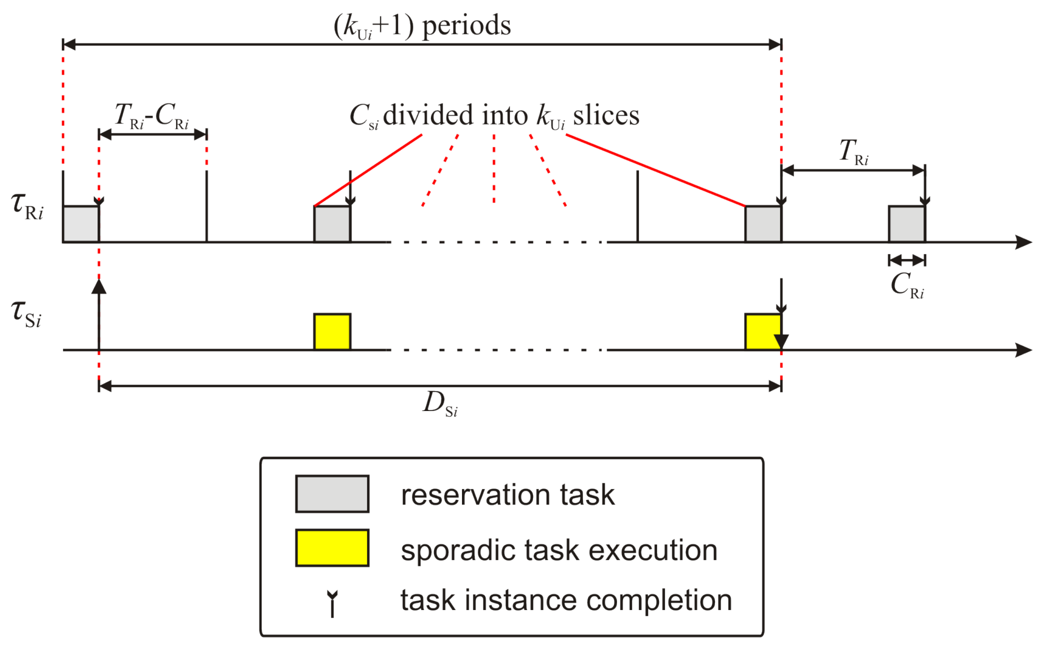


Figure 4.4: *Worst case arrival of a sporadic task instance.*

$(T_{Ri} - C_{Ri})$ in the first period and k_{Ui} periods are as long as the execution window, i.e., the relative deadline D_{Si}, of the sporadic task instance (see Figure 4.4); The resulting equation is shown in Equation (4.9a). We calculate the period of the reservation task using Equation (4.9b).

$$C_{Ri} = \frac{C_{Si}}{k_{Ui}} \quad \text{with } k_{Ui} \in \mathbb{N}\backslash\{0\} \tag{4.8}$$

$$\begin{aligned}
& (T_{Ri} - C_{Ri}) + k_{Ui} \cdot T_{Ri} = T_{Si} && \text{with } k_{Ui} \in \mathbb{N}\backslash\{0\} && (4.9a)\\
\Leftrightarrow\quad & (k_{Ui} + 1)\, T_{Ri} - C_{Ri} = T_{Si} \\
\Leftrightarrow\quad & T_{Ri} = \frac{T_{Si} + C_{Ri}}{k_{Ui} + 1} \\
\Leftrightarrow\quad & T_{Ri} = \frac{T_{Si} + \frac{C_{Si}}{k_{Ui}}}{k_{Ui} + 1} &&&& (4.9b)
\end{aligned}$$

As a consequence, the resulting utilization of reservation tasks is:

$$\begin{aligned}
U_{Ri} &= \frac{C_{Ri}}{T_{Ri}} \\
&= \frac{C_{Si}k_{Ui} + C_{Si}}{T_{Si}k_{Ui} + C_{Si}} \\
&= \frac{U_{Si}k_{Ui} + U_{Si}}{k_{Ui} + U_{Si}}
\end{aligned} \tag{4.10}$$

As Equation (4.10) shows, the utilization of a reservation task depends only on the utilization of the sporadic task and k_{Ui}. With increasing k_{Ui}, we can achieve a finer

granularity of the RET and hence, we can decrease the needed utilization to guarantee sporadic tasks (see Figure 4.5).
The extrema of the reservation task's utilization are calculated in Equations (4.11) and (4.12). Depending on $k_{\mathrm{U}i}$, the reservation task's utilization can be decreased down to the sporadic task utilization if the WCET of the sporadic task is divided into an infinite number of slices.

$$\begin{aligned} U_{\mathrm{R}i}(k_{\mathrm{U}i}=1) &= \frac{U_{\mathrm{S}i}\cdot 1 + U_{\mathrm{S}i}}{1+U_{\mathrm{S}i}} \\ &= \frac{2U_{\mathrm{S}i}}{1+U_{\mathrm{S}i}} \end{aligned} \tag{4.11}$$

$$\begin{aligned} \lim_{k_{\mathrm{U}i}\to\infty} U_{\mathrm{R}i} &= \lim_{k_{\mathrm{U}i}\to\infty} \frac{U_{\mathrm{S}i}k_{\mathrm{U}i} + U_{\mathrm{S}i}}{k_{\mathrm{U}i}+U_{\mathrm{S}i}} \\ &= \lim_{k_{\mathrm{U}i}\to\infty} \frac{k_{\mathrm{U}i}\left(U_{\mathrm{S}i}+\frac{U_{\mathrm{S}i}}{k_{\mathrm{U}i}}\right)}{k_{\mathrm{U}i}\left(1+\frac{U_{\mathrm{S}i}}{k_{\mathrm{U}i}}\right)} \\ &= U_{\mathrm{S}i} \end{aligned} \tag{4.12}$$

On the contrary, an infinite number of slices for one sporadic task instance execution

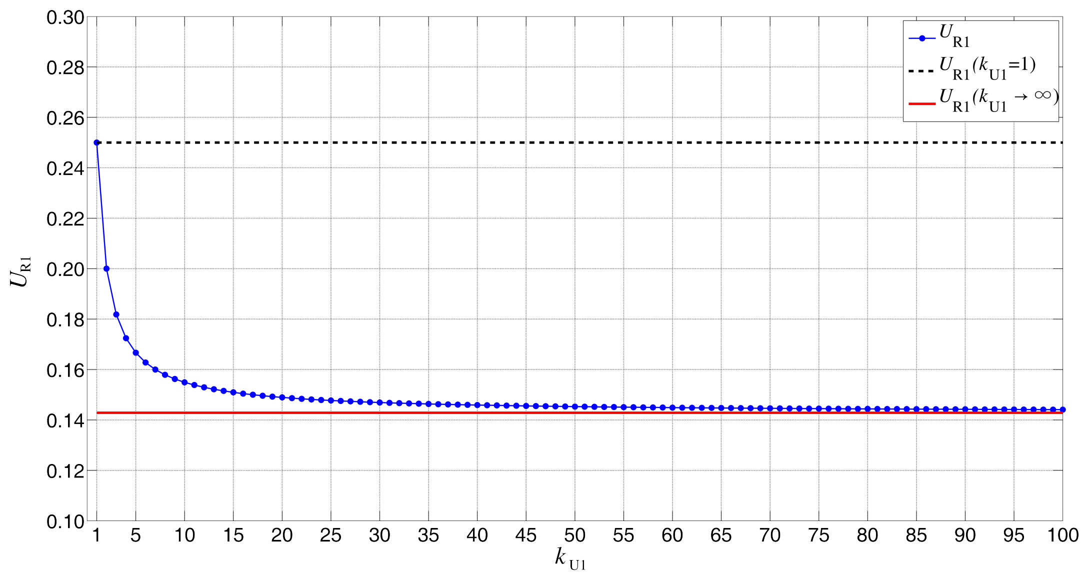


Figure 4.5: *Utilization of the reservation task for a sporadic task* $\tau_{S1} = \langle 10, 70\rangle$.

leads to an infinite number of preemptions of the sporadic task instance. We show the influence of this switching (preemption) overhead in Section 4.3.
Each sporadic task has its own reservation task. Hence, we guarantee each sporadic task independently from other sporadic tasks such that they do not interfere.

Theorem and Proof

In the following, we show that independent of the release time of a sporadic task instance, this method reserves enough execution time, such that the sporadic task instance can complete its execution within its execution window.

Theorem 4.1.
In each time interval $[t_1; t_2)$ with $t_2 - t_1 \geq T_{\mathrm{S}i}$, i.e. arrival $r_{i,j} \geq t_1$ and deadline $d_{\mathrm{S}i,j} \leq t_2$, if $C_{\mathrm{S}i}$ is the execution demand of the sporadic task instance (with relative deadline $D_{\mathrm{S}i}$), there will be at least $C_{\mathrm{S}i}$ time units reserved for the execution of the sporadic task instance. ◀

Proof 4.1.
To prove the theorem, we will show that there are at least $C_{\mathrm{S}i}$ time units reserved within the interval $[t_1; t_2)$:

$$\left\lfloor \frac{t_2 - t_1}{T_{\mathrm{R}i}} \right\rfloor C_{\mathrm{R}i} \geq C_{\mathrm{S}i}$$

Replacing $C_{\mathrm{R}i}$ by its definition as shown in Equation (4.8):

$$\left\lfloor \frac{t_2 - t_1}{T_{\mathrm{R}i}} \right\rfloor \frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}} \geq C_{\mathrm{S}i}$$

Multiplying with $\frac{k_{\mathrm{U}i}}{C_{\mathrm{S}i}}$:

$$\left\lfloor \frac{t_2 - t_1}{T_{\mathrm{R}i}} \right\rfloor \geq k_{\mathrm{U}i}$$

Hence, the interval $[t_1; t_2)$ must be greater than or equal to the product $k_{\mathrm{U}i} \cdot T_{\mathrm{R}i}$:

$$t_2 - t_1 \geq k_{\mathrm{U}i} T_{\mathrm{R}i}$$

Replacing $T_{\mathrm{R}i}$ by its definition as shown in Equation (4.9b):

$$\begin{aligned} t_2 - t_1 &\geq k_{\mathrm{U}i} \frac{T_{\mathrm{S}i} + \frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}}}{k_{\mathrm{U}i} + 1} \\ \Leftrightarrow t_2 - t_1 &\geq \frac{k_{\mathrm{U}i} T_{\mathrm{S}i} + C_{\mathrm{S}i}}{k_{\mathrm{U}i} + 1} \end{aligned}$$

We perform polynomial division for the right-hand side of the inequality:

$$(k_{\mathrm{U}i} T_{\mathrm{S}i} + C_{\mathrm{S}i}) : (k_{\mathrm{U}i} + 1) = T_{\mathrm{S}i} + \frac{C_{\mathrm{S}i} - T_{\mathrm{S}i}}{k_{\mathrm{U}i} + 1}$$

Thus, the inequality results in:

$$\begin{aligned} t_2 - t_1 &\geq T_{\mathrm{S}i} + \frac{C_{\mathrm{S}i} - T_{\mathrm{S}i}}{k_{\mathrm{U}i} + 1} \\ \Leftrightarrow \frac{T_{\mathrm{S}i} - C_{\mathrm{S}i}}{k_{\mathrm{U}i} + 1} &\geq T_{\mathrm{S}i} - (t_2 - t_1) \end{aligned}$$

With

$$\begin{aligned} & (t_2 - t_1) \overset{\text{def.}}{\geq} T_{\mathrm{S}i} \\ \Leftrightarrow \quad & T_{\mathrm{S}i} - (t_2 - t_1) \leq 0 \end{aligned}$$

follows:

$$\frac{T_{\mathrm{S}i} - C_{\mathrm{S}i}}{k_{\mathrm{U}i} + 1} \geq 0 \tag{4.13}$$

According to the definition of $k_{\mathrm{U}i} \in \mathbb{N}\backslash\{0\}$, Inequality (4.13) is greater than or equal to zero if the numerator of the fraction is greater than or equal to zero. The numerator is greater than or equal to zero if the WCET does not exceed the period of the sporadic task which is a trivial condition. Hence, we proved the assertion. ■

4.2.4 Adaption for Mixed-Criticality Task Sets

The transformation method we have shown before is based on WCETs and periods (i.e., minimum inter-arrival time) of sporadic tasks. In mixed-criticality systems, there are several WCETs per task. As a consequence, several questions have to be answered when thinking about applicability of the presented method for mixed-criticality systems. For reasons of simplicity and clarity, we consider the dual-criticality case with low and high criticality level. The issues shown in the following, are also valid for mixed-criticality systems with more than two criticality levels.

Criticality level of sporadic tasks. We can handle low criticality sporadic tasks without any problems. Low criticality tasks are not certified and not essential for survivability of the system. There is only one WCET per task, or the WCET is the same for low and high criticality level. In this case, we can transform sporadic tasks as shown before.

On the contrary, the unknown arrival times of high criticality sporadic tasks results in a possibly unknown schedulability. As a result, we need to ensure schedulability by using an offline guarantee algorithm to check whether sporadic tasks can always be included at runtime, e.g., [SRLK02, IF99]. Without this test, a guarantee and certification of high criticality tasks is useless.

Transformation of several WCETs. High criticality tasks have two (maybe) different WCETs which would result in different reservation tasks. Although using the same $k_{\mathrm{U}i}$, both the RET and the period of the reservation tasks can be different. In the following, we show the basic idea of three possible solutions for this issue:

1. A trivial approach is to use the largest, i.e., the high criticality, WCET to create the reservation tasks. As a result, the high criticality behavior is guaranteed already in low criticality system mode. The advantage is a simple transformation, but on the contrary, this assumption is very pessimistic and results in a possibly strongly under-utilized system.

2. In a two step approach, we can transform sporadic tasks based on their high criticality WCET. We obtain a value for the RET and the period of the reservation

task. In a second step, we use the obtained period to calculate a corresponding RET for the low criticality behavior. In this way, the transformation is less pessimistic and more flexible. The drawback of this approach is the additionally needed step in the transformation which is only a small problem because the transformation is done before runtime. Furthermore, the scheduling process is more complex to guarantee the high criticality requirements.

3. Another approach is to separate the WCET of the low criticality behavior from the *additionally* needed WCET in the high criticality case. We create two reservation tasks for one sporadic task. In Section 5.3, we show how we can separate the low criticality demand from the additionally needed demand in the high criticality case. This approach is less pessimistic and allows for a flexible scheduling process. On the contrary, more tasks can also increase the scheduling complexity.

4.3 Utilization of Reservation Tasks

In the following, we analyze the utilization of reservation tasks. Using the parameter $k_{\mathrm{U}i}$, we can adapt the utilization to the system requirements while including the introduced overhead of the transformation method.

In theory, we can divide the reservation tasks into an infinite number of infinitesimally short slices. But in practice, this is not possible because of the switching overhead (e.g., [SE04]). The switching overhead includes the time the scheduler/dispatcher needs to preempt the current task and to select and start the next executing task. As a consequence, the time consumed by this switching has to be included into the feasibility tests.
Besides the switching overhead, the length of a clock cycle is also a restricting factor for the minimum length of a slice.

The reservation tasks are periodic and hence, their utilization can be calculated by Equation (4.4). Splitting the RET for sporadic into several slices results in additional switching overhead which is not accommodated by the WCETs of the sporadic tasks. As a consequence, besides the RET of the reservation tasks, we have to consider the switching overhead λ which is added to the RET. For the guarantee of the sporadic tasks, we assume the worst-case for the switching overhead, i.e., for each instance of the reservation task the maximum switching overhead λ is needed. The effective utilization $U_{\mathrm{R}i}$ of the reservation tasks including the additional switching overhead is determined in Equation (4.14).

$$\begin{aligned} U_{\mathrm{R}i} &= \frac{C_{\mathrm{R}i} + \lambda}{T_{\mathrm{R}i}} \\ &= \frac{(C_{\mathrm{S}i} + k_{\mathrm{U}i}\lambda)(k_{\mathrm{U}i} + 1)}{k_{\mathrm{U}i}T_{\mathrm{S}i} + C_{\mathrm{S}i}} \end{aligned} \tag{4.14}$$

If we set $\lambda = 0$, we obtain the utilization mentioned earlier without overhead consideration, as shown in Equation (4.10). Figure 4.6 shows the comparison between a reservation task's utilization with and without overhead for an example sporadic task. Without considering the switching overhead, the utilization of the reservation task approximates the utilization of the sporadic task ($k_{\mathrm{U}i} \rightarrow \infty$). But in practice, the switching overhead is not negligible and we choose an integer value for $k_{\mathrm{U}i} \nrightarrow \infty$ close to the minimum of the function including the overhead (▲). For the example reservation task in Figure 4.6, the minimum utilization is obtained for $k_{\mathrm{U}i} = 9$. Due to the increasing impact of the switching overhead, the utilization exceeds the utilization of the first step ($k_{\mathrm{U}i} = 1$) at $k_{\mathrm{U}i} = 73$

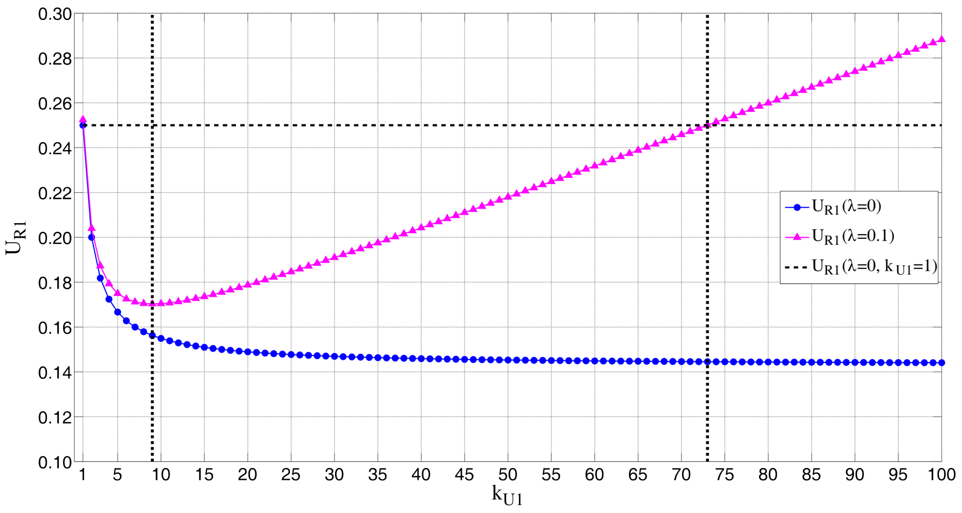


Figure 4.6: *Utilization considering switching overheads of a reservation task for a sporadic task $\tau_{S1} = \langle 10, 70 \rangle$.*

In Figure 4.7, we show the impact of varying switching overheads. As a reference, we use the reserved utilization for $\tau_{\mathrm{S}1}$ with the switching overhead used in Figure 4.6 (▲). Further, we show the reserved utilization including an overhead which is five times higher (◯), which corresponds to 5% of $C_{\mathrm{S}1}$, than the reference switching overhead of $\lambda = 0.1$ and reserved utilizations for two times higher (□), i.e, 2% of $C_{\mathrm{S}1}$, for two times smaller (◇), i.e, 0.5% of $C_{\mathrm{S}1}$, for five times smaller (▽) i.e, 0.2% of $C_{\mathrm{S}1}$, switching overhead.

We can observe that higher overheads result in higher reserved utilizations. In theory, for $k_{\mathrm{U}i} = 1$ only one slice is needed to execute a sporadic task instance and hence, no additional preemption is introduced. On the contrary in practice, although only the length of one slice is needed, a sporadic task instance can arrive during this reserved execution time such that e.g., 50% of current slice is used to execute and 50% of next reservation task instance is used to finish execution. As a result, one additional preemption is introduced and thus, already for $k_{\mathrm{U}i} = 1$ the reserved utilization is higher for higher switching overheads. For higher overheads, the impact of the switching overheads transcends the improvements of utilization reduction by finer granularity (more slices) already for smaller $k_{\mathrm{U}i}$, i.e., the sporadic WCET is reserved by fewer reservation slices.

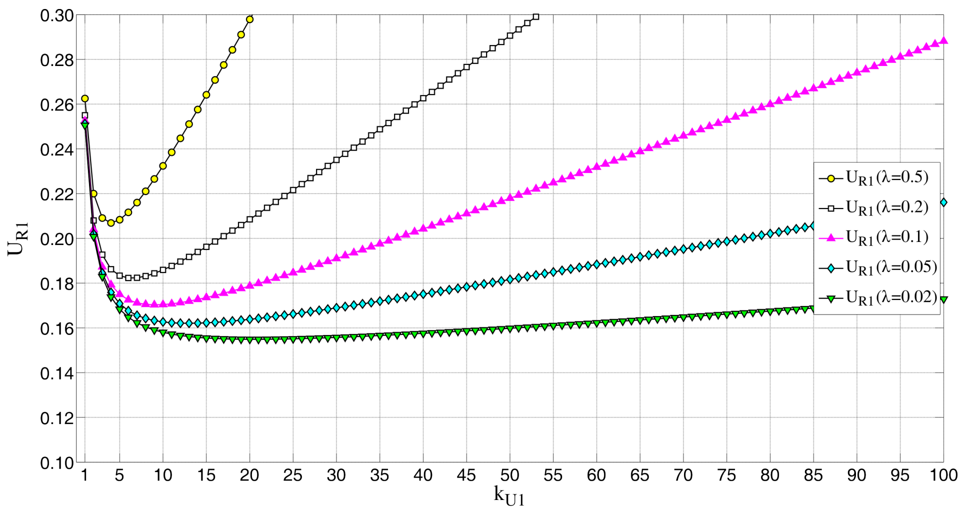


Figure 4.7: *Utilization considering different switching overheads of a reservation task for a sporadic task* $\tau_{S1} = \langle 10, 70 \rangle$.

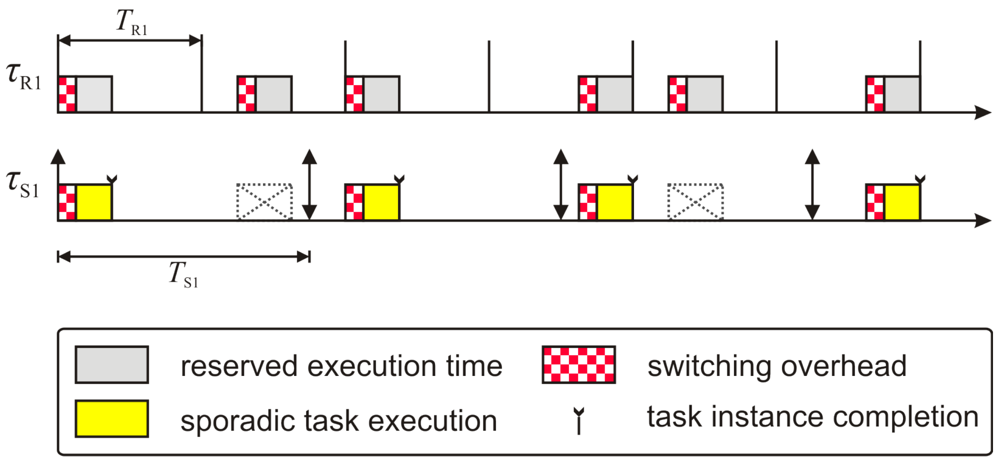


Figure 4.8: *Unused RET with* $k_{U1} = 1$.

As Figure 4.8 shows, we reserve more utilization than actually needed even if the sporadic tasks arrive with maximum frequency.

This over-provisioning is done because of the unknown arrival times of the sporadic tasks at runtime. In the following, we calculate the amount of *unused reserved utilization* Λ_i. For this calculation, we consider the arrival pattern which creates the highest workload, i.e., the sporadic task instances always occur with their minimum inter-arrival time. Equation (4.15b) calculates the amount of unused reserved utilization Λ_i of one sporadic task. We include the switching overhead into the reservation tasks such that Λ_i accounts for the switching overhead and the overhead by unused slices. The reserved utilization includes the switching overhead and the difference to the actual execution of the sporadic task results in Λ_i as shown in Equation (4.15a).

$$\Lambda_i\left(k_{\mathrm{U}i}\right) = \underbrace{\frac{C_{\mathrm{R}i}+\lambda}{T_{\mathrm{R}i}}}_{\text{reserved utilization}} - \underbrace{\frac{C_{\mathrm{S}i}}{T_{\mathrm{S}i}}}_{\text{sporadic task utilization}} \tag{4.15a}$$

$$= \frac{k_{\mathrm{U}i}^2\lambda + k_{\mathrm{U}i}\lambda + C_{\mathrm{S}i} - \frac{(C_{\mathrm{S}i})^2}{T_{\mathrm{S}i}}}{k_{\mathrm{U}i}T_{\mathrm{S}i} + C_{\mathrm{S}i}} \tag{4.15b}$$

Figure 4.9 shows the unused reserved utilization for the example task of Figure 4.8. For values of $k_{\mathrm{U}1}$ with $\Lambda_1(k_{\mathrm{U}1}) > \min(\Lambda_1(k_{\mathrm{U}1}))$, the switching overhead has a stronger influence on the utilization of the reservation task than the utilization reduction achieved by increasing $k_{\mathrm{U}1}$. The minimum unused reserved utilization is obtained for $k_{\mathrm{U}1} = 9$ which is also the $k_{\mathrm{U}1}$-value when the utilization of the reservation task is minimum.

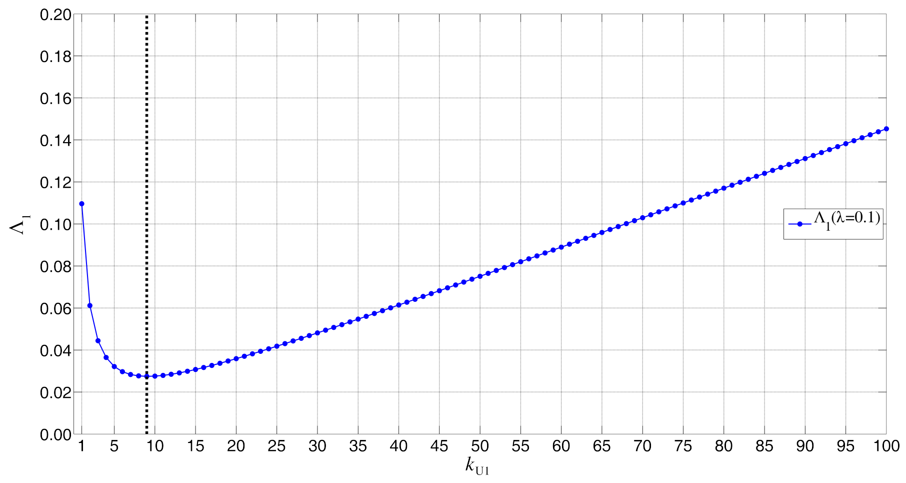


Figure 4.9: *Unused reserved utilization considering switching overheads of the reservation task for a sporadic task* $\tau_{S1} = \langle 10, 70\rangle$.

When choosing $k_{\mathrm{U}i}$ for a desired utilization of the reservation task, we must not forget that reservation tasks have to be schedulable. In Section 4.4, Equation (4.23) shows the necessary schedulability condition for the reservation tasks.

We showed before how to minimize the needed utilization of reservation tasks. It is possible that designer wants to choose a larger or shorter period than the one which results in minimum reserved utilization.

On the next pages, we show possible approaches to determine a feasible range for $k_{\mathrm{U}i}$ based on the reserved utilization:

1. Vertex approach on page 42;
2. First step approach on page 43;
3. Utilization limit approach on page 44;
4. Total utilization limit approach on page 45;

Vertex approach. Figure 4.10 visualizes the choices for $k_{\mathrm{U}i}$. The range is characterized by the minimum utilization of the reservation task, i.e., increasing $k_{\mathrm{U}i}$ also increases the reserved utilization. Inequality (4.16a) shows that we determine all $k_{\mathrm{U}i}$ such that with increasing $k_{\mathrm{U}i}$ the utilization does not decrease. Although the utilization for values of $k_{\mathrm{U}i}$ larger than the determined $k_{\mathrm{U}i}$ is still smaller than initial reserved utilization ($k_{\mathrm{U}i} = 1$), the number of preemptions increases without reducing the reserved utilization. This approach minimizes both utilization and number of preemptions. As a result, we choose $k_{\mathrm{U}i} \in [1, k_{\mathrm{U}i}^{\mathrm{vertex}}]$ whereas $k_{\mathrm{U}i}^{\mathrm{vertex}}$ is the largest $k_{\mathrm{U}i}$ satisfying Inequality (4.16b).

$$\begin{aligned}
& U_{\mathrm{R}i}\left(k_{\mathrm{U}i}\right) < U_{\mathrm{R}i}\left(k_{\mathrm{U}i}+1\right) && (4.16a)\\
\Leftrightarrow\quad & \frac{C_{\mathrm{R}i}(k_{\mathrm{U}i})+\lambda}{T_{\mathrm{R}i}(k_{\mathrm{U}i})} < \frac{C_{\mathrm{R}i}(k_{\mathrm{U}i}+1)+\lambda}{T_{\mathrm{R}i}(k_{\mathrm{U}i}+1)} &&\\
\Leftrightarrow\quad & \frac{\frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}}+\lambda}{\frac{T_{\mathrm{S}i}+\frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}}}{k_{\mathrm{U}i}+1}} < \frac{\frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}+1}+\lambda}{\frac{T_{\mathrm{S}i}+\frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}+1}}{k_{\mathrm{U}i}+2}} &&\\
\Leftrightarrow\quad & [\ldots] &&\\
\Leftrightarrow\quad & \lambda\left(T_{\mathrm{S}i}+\frac{2U_{\mathrm{S}i}T_{\mathrm{ST}}}{k_{\mathrm{U}i}}\right) - U_{\mathrm{S}i}T_{\mathrm{S}i}^{2}\frac{1-U_{\mathrm{S}i}}{k_{\mathrm{U}i}\left(k_{\mathrm{U}i}+1\right)} > 0 && (4.16b)
\end{aligned}$$

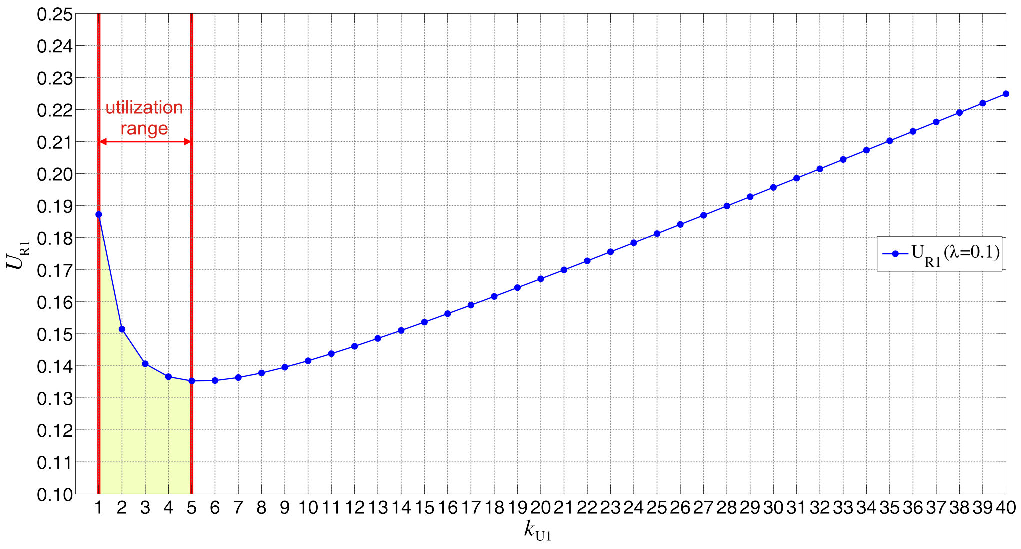


Figure 4.10: *Reserved utilization U_{R1} and utilization range according to vertex approach.*

First step approach. The range of this approach, shown in Figure 4.11, is characterized by the initial reserved utilization, i.e., utilization with $k_{\mathrm{U}i} = 1$. The reserved utilization increases for many slices because of the overhead. At some point, the reserved utilization transcends the initial reserved utilization. We choose values for $k_{\mathrm{U}i}$ satisfying Inequality (4.17b). This approach can be seen as special case of next approach: the utilization limit approach.

$$\begin{aligned}
& U_{\mathrm{R}i}\left(k_{\mathrm{U}i}\right) \leq U_{\mathrm{R}i}\left(k_{\mathrm{U}i}=1\right) && \text{(4.17a)}\\
\Leftrightarrow\quad & \frac{C_{\mathrm{R}i}(k_{\mathrm{U}i})+\lambda}{T_{\mathrm{R}i}(k_{\mathrm{U}i})} \leq \frac{C_{\mathrm{R}i}(k_{\mathrm{U}i}=1)+\lambda}{T_{\mathrm{R}i}(k_{\mathrm{U}i}=1)} \\
\Leftrightarrow\quad & \frac{\frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}}+\lambda}{\frac{T_{\mathrm{S}i}+\frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}}}{k_{\mathrm{U}i}+1}} \leq \frac{C_{\mathrm{S}i}+\lambda}{\frac{T_{\mathrm{S}i}+C_{\mathrm{S}i}}{2}} \\
\Leftrightarrow\quad & [\dots] \\
\Leftrightarrow\quad & \lambda T_{\mathrm{S}i}\left(k_{\mathrm{U}i}-1+U_{\mathrm{S}i}\left(k_{\mathrm{U}i}+1-\frac{2}{k_{\mathrm{U}i}}\right)\right)+ \\
& U_{\mathrm{S}i}T_{\mathrm{S}i}\left(\frac{1}{k_{\mathrm{U}i}}+1+U_{\mathrm{S}i}-\frac{U_{\mathrm{S}i}}{k_{\mathrm{U}i}}\right) \leq 0 && \text{(4.17b)}
\end{aligned}$$

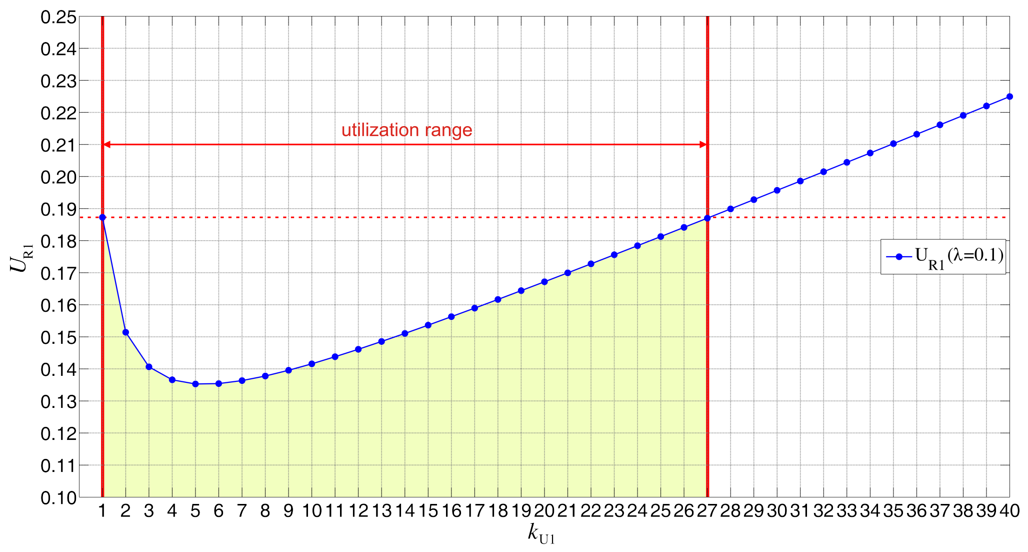


Figure 4.11: *Reserved utilization U_{R1} and utilization range according to first step approach.*

Utilization limit approach. As shown in Figure 4.12, the available utilization of each reservation task is limited (dotted line). Possible reasons for this can be that there is not more utilization available for this reservation task or the designer wants to privilege another sporadic task by granting more utilization to the corresponding reservation task and reducing the available utilization for this reservation task. As a consequence, we have to choose values for $k_{\mathrm{U}i}$ such that resulting utilization of reservation task is below the threshold U_{limit}, which is shown in Inequality (4.18a). The resulting values for $k_{\mathrm{U}i}$ have to satisfy Inequality (4.18b).

$$\begin{aligned}
& & U_{\mathrm{R}i} &\leq U_{\mathrm{limit}} & \quad (4.18\mathrm{a})\\
&\Leftrightarrow & \frac{C_{\mathrm{R}i}+\lambda}{T_{\mathrm{R}i}} &\leq U_{\mathrm{limit}} \\
&\Leftrightarrow & \frac{\frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}}+\lambda}{\frac{T_{\mathrm{R}i}+\frac{C_{\mathrm{R}i}}{k_{\mathrm{U}i}}}{k_{\mathrm{U}i}+1}} &\leq U_{\mathrm{limit}} \\
&\Leftrightarrow & & [\dots] \\
&\Leftrightarrow & \frac{\lambda k_{\mathrm{U}i}^2+\left(U_{\mathrm{S}i}T_{\mathrm{S}i}+\lambda\right)k_{\mathrm{U}i}+2\lambda U_{\mathrm{S}i}\left(U_{\mathrm{S}i}+1\right)}{T_{\mathrm{R}i}\left(k_{\mathrm{U}i}+U_{\mathrm{S}i}\right)} &\leq U_{\mathrm{limit}} & \quad (4.18\mathrm{b})
\end{aligned}$$

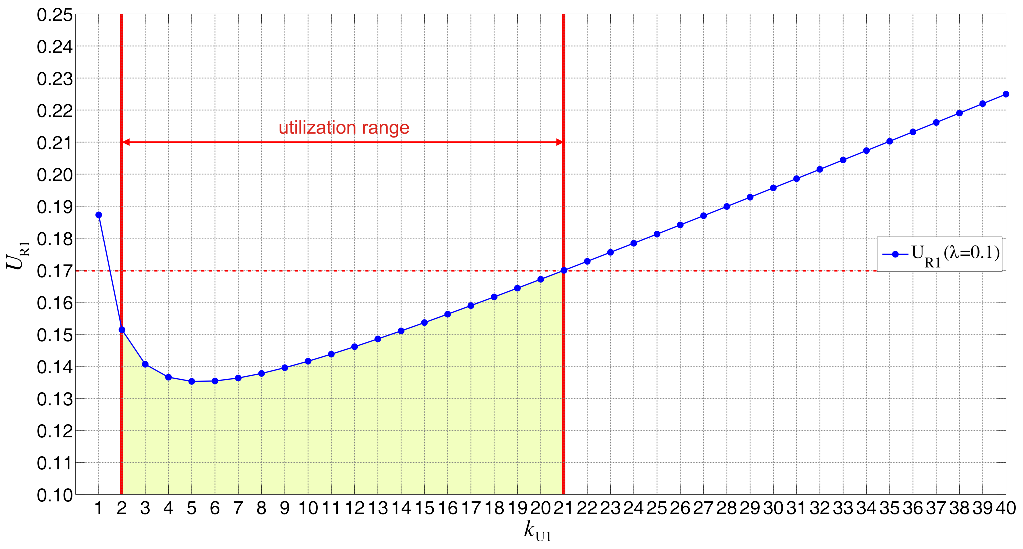


Figure 4.12: *Reserved utilization U_{R1} and utilization range according to utilization limit approach.*

Total utilization limit approach. Figure 4.13 visualizes the approach where the *total utilization* of all reservation tasks is limited. In contrast to selecting one value for each $k_{\mathrm{U}i}$ of every task, we select one (same) value $k_{\mathrm{U}i}$ for all tasks. This reduces the complexity of calculating $k_{\mathrm{U}i}$. k_{U} is the value used to determine the parameters of all reservation tasks. We choose a value for k_{U} such that resulting total utilization of all reservation tasks is below threshold $U_{\mathrm{total\_limit}}$. The values for k_{U} have to satisfy Inequality (4.19b).

$$\sum_{i=1}^{n} U_{\mathrm{R}i} \leq U_{\mathrm{total\_limit}} \tag{4.19a}$$

$$\Leftrightarrow \quad \sum_{i=1}^{n} \frac{C_{\mathrm{R}i}+\lambda}{T_{\mathrm{R}i}} \leq U_{\mathrm{total\_limit}}$$

$$\Leftrightarrow \quad \sum_{i=1}^{n} \frac{\frac{C_{\mathrm{S}i}}{k_{\mathrm{U}}}+\lambda}{\frac{T_{\mathrm{R}i}+\frac{C_{\mathrm{R}i}}{k_{\mathrm{U}}}}{k_{\mathrm{U}}+1}} \leq U_{\mathrm{total\_limit}}$$

$$\Leftrightarrow \quad [\ldots]$$

$$\Leftrightarrow \quad \sum_{i=1}^{n} \frac{\lambda k_{\mathrm{U}}^2+\left(U_{\mathrm{S}i}T_{\mathrm{S}i}+\lambda\right)k_{\mathrm{U}}+2\lambda U_{\mathrm{S}i}\left(U_{\mathrm{S}i}+1\right)}{T_{\mathrm{R}i}\left(k_{\mathrm{U}}+U_{\mathrm{S}i}\right)} \leq U_{\mathrm{total\_limit}} \tag{4.19b}$$

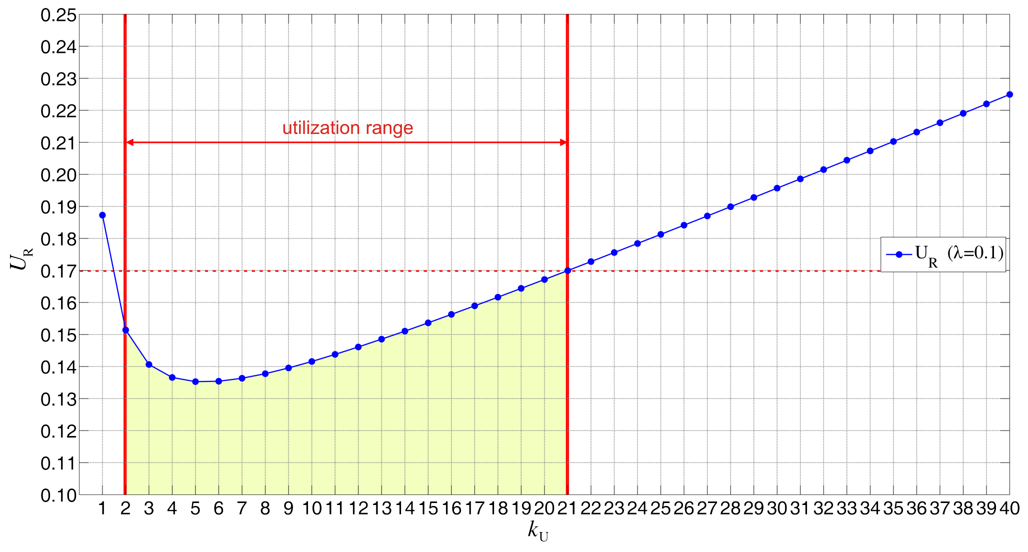


Figure 4.13: *Reserved utilization U_{R1} and utilization range according to total utilization limit approach.*

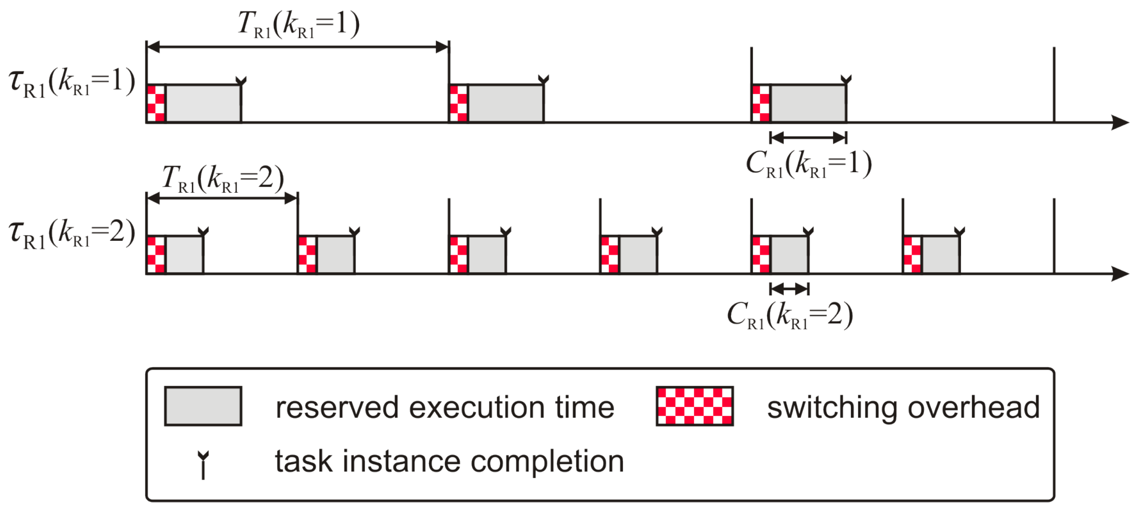
4.4 Worst-Case Response Time

Up to this point, we can guarantee sporadic tasks in the offline schedule. To achieve this, we split the sporadic task's WCET into $k_{\mathrm{U}i}$ slices. With shorter periods and shorter slices, we achieve a finer granularity and can reduce the WCRT of the sporadic task instances. As a consequence, we adapt Equations (4.8) and (4.9b), such that the periods and RETs are shortened.

4.4.1 Worst-Case Response Time Reduction

In the following, we shorten the period and the length of the slices of the reservation tasks to achieve a finer granularity, but we do not want to change the utilization of the reservation tasks. We introduce a new parameter $k_{\mathrm{R}i}$ to reduce the WCRT $R_{\mathrm{S}i}$ of the sporadic tasks. In contrast to $k_{\mathrm{U}i}$, that changes the utilization of the reservation task, $k_{\mathrm{R}i}$ only changes the granularity of the reservation tasks and hence, the WCRT of the sporadic task. As a secondary consequence, $k_{\mathrm{R}i}$ causes more preemptions which increases the switching overhead and thus, the increases the utilization.

The periods $T_{\mathrm{R}i}^{(4.9\mathrm{b})}$ presented in Equation (4.9b) and the length of the slices $C_{\mathrm{R}i}^{(4.8)}$ shown in Equation (4.8) of the reservation task $\tau_{\mathrm{R}i}$ are split into $k_{\mathrm{R}i}$ fractions with $k_{\mathrm{R}i} \in \mathbb{N}\backslash\{0\}$. As Figure 4.14 shows, the utilization of the reservation tasks is unchanged (ignoring the constant switching overhead) but the slices are more regularly distributed over the schedule.



$T_{\mathrm{R1}}(k_{\mathrm{R1}}=1)$
$\tau_{\mathrm{R1}}(k_{\mathrm{R1}}=1)$
$C_{\mathrm{R1}}(k_{\mathrm{R1}}=1)$
$T_{\mathrm{R1}}(k_{\mathrm{R1}}=2)$
$\tau_{\mathrm{R1}}(k_{\mathrm{R1}}=2)$
$C_{\mathrm{R1}}(k_{\mathrm{R1}}=2)$
reserved execution time
switching overhead
task instance completion

Figure 4.14: *Shortening period and length of slices of a reservation task.*

As a consequence, the equations calculating the period and RET (slice length) of the reservation tasks have to be adapted to these changes. Equations (4.20) and (4.21) show the adapted equations for the RET calculation and for the period calculation, respectively.

$$\begin{aligned} C_{\mathrm{R}i} &= \frac{C_{\mathrm{R}i}^{(4.8)}}{k_{\mathrm{R}i}} \qquad \text{with } \{k_{\mathrm{R}i}, k_{\mathrm{R}i}\} \in \mathbb{N}\backslash\{0\} \\ &= \frac{C_{\mathrm{S}i}}{k_{\mathrm{R}i} k_{\mathrm{U}i}} \end{aligned} \tag{4.20}$$

$$\begin{aligned} T_{\mathrm{R}i} &= \frac{T_{\mathrm{R}i}^{(4.9b)}}{k_{\mathrm{R}i}} \qquad \text{with } \{k_{\mathrm{R}i}, k_{\mathrm{R}i}\} \in \mathbb{N}\backslash\{0\} \\ &= \frac{T_{\mathrm{S}i} + \frac{C_{\mathrm{S}i}}{k_{\mathrm{U}i}}}{k_{\mathrm{R}i}\left(k_{\mathrm{U}i}+1\right)} \end{aligned} \tag{4.21}$$

Thus, the utilization of the reservations tasks can be adapted with $k_{\mathrm{U}i}$ and the WCRT can be adapted with $k_{\mathrm{R}i}$. An improvement of the WCRT will lead to an increasing utilization because of the switching overhead. Equation (4.22) shows the utilization of a reservation task including the adaptation for WCRT reduction.

$$\begin{aligned} U_{\mathrm{R}i} &= \frac{C_{\mathrm{R}i}^{(4.8)} + \lambda}{T_{\mathrm{R}i}^{(4.9b)}} \\ &= [\dots] \\ &= \frac{C_{\mathrm{S}i}\left(k_{\mathrm{U}i}+1\right) + \lambda k_{\mathrm{U}i} k_{\mathrm{R}i}\left(k_{\mathrm{U}i}+1\right)}{T_{\mathrm{S}i} k_{\mathrm{U}i} + C_{\mathrm{S}i}} \end{aligned} \tag{4.22}$$

The utilization of the reservation tasks restricts the schedulability of the entire task set. This results in the necessary schedulability condition shown in Equation (4.23).

$$\sum_{i=1}^{n} U_i + \sum_{i=1}^{m} U_{\mathrm{R}i} \leq 1 \tag{4.23}$$

The utilization of the periodic tasks is given by the designer and cannot be influenced. Changing $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$ influences the utilization of the reservation tasks, which includes the switching overheads introduced by the transformation. Figure 4.15 shows the utilization of one example reservation task versus the adaptation factors $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$. The utilization available for the guarantee of a specific sporadic task restricts the combinations of $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$, i.e., for a maximum available utilization for a reservation task only values $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$ can be chosen which are below the corresponding utilization level in the graph.

Figure 4.16 depicts an example which shows that WCRT can be reduced when splitting the period and RET into shorter slices. Both calculations of reservation task's parameters are based on the same sporadic task $\tau_{\mathrm{S}1} = \langle 10, 70 \rangle$. The switching overhead is shown disproportionately large to show how it influences the WCRT.

For $(k_{\mathrm{U}1} = 1, k_{\mathrm{R}1} = 1)$, the WCRT is 70 and with $(k_{\mathrm{U}1} = 1, k_{\mathrm{R}1} = 2)$ the WCRT can be decreased to 55, in this example. In the next step, we describe the influence of switching on the WCRT.

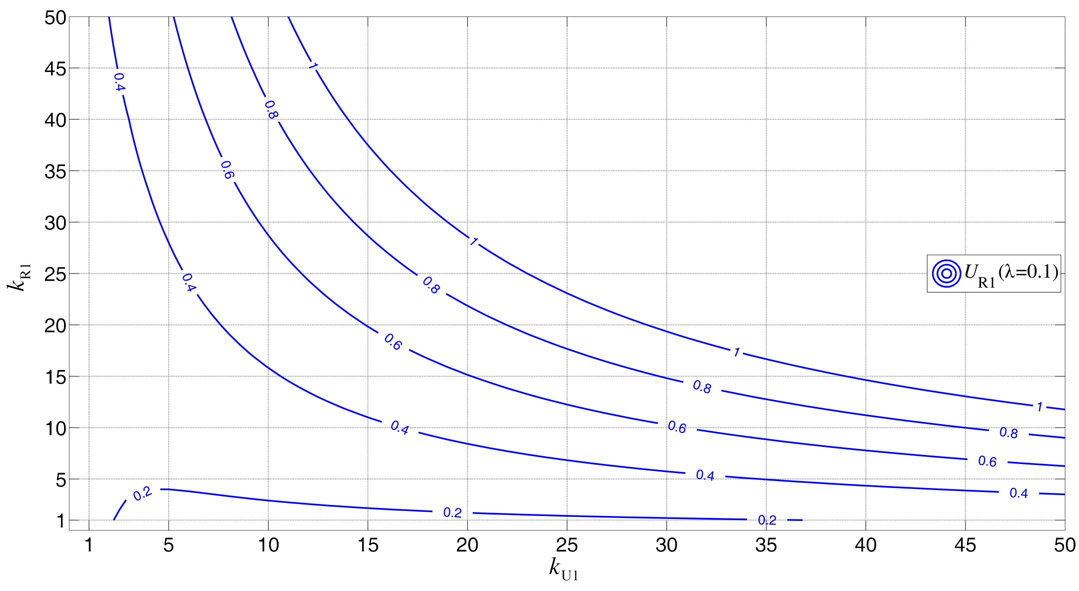


Figure 4.15: *Utilization of one reservation task.*

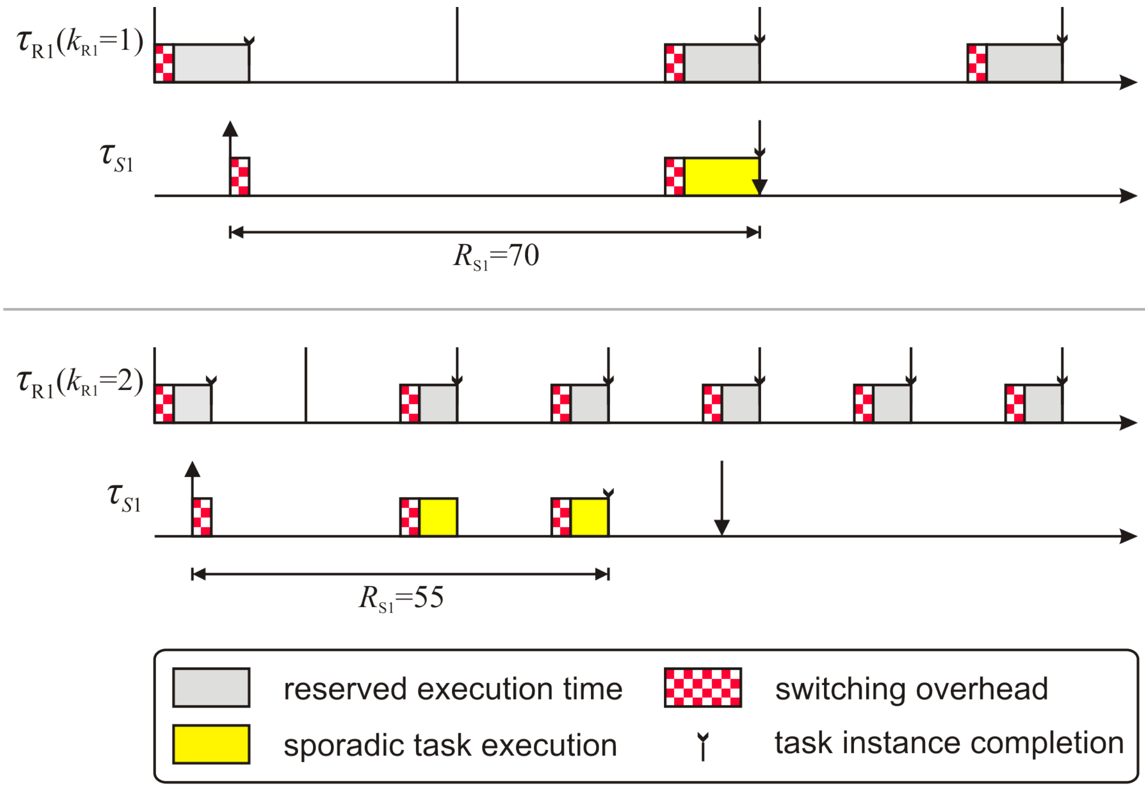


Figure 4.16: *Reservation tasks with $k_{R1} = 1$ (top) and $k_{R1} = 2$ (bottom) and resulting WCRT of sporadic task at runtime.*

4.4.2 Results of the Worst-Case Response Time Reduction

We assumed for the determination of the WCRT, that the sporadic task instance arrives at the end of a slice so that there is just enough time left for selecting the next task and fetching the data into the cache (switching overhead λ). This slice is scheduled at the beginning of its period and the following slices are scheduled at the end of their periods (see examples in Figure 4.16). Each time the data is removed from the cache so that it has to be fetched again from main memory. With this scenario, we can determine the WCRT which is shown in Equation (4.24). Before the execution of a sporadic task instance can start, the instance has to wait for the beginning of the next period because the RET of the current period in which the instance arrived is already gone. To complete the sporadic task execution, we need $(k_{\mathrm{U}i} \cdot k_{\mathrm{R}i})$ slices and hence, $(k_{\mathrm{U}i} \cdot k_{\mathrm{R}i})$ periods.

$$\begin{aligned} R_{\mathrm{S}i} &= \lambda + (T_{\mathrm{R}i} - C_{\mathrm{R}i} - \lambda) + k_{\mathrm{U}i} k_{\mathrm{R}i} T_{\mathrm{R}i} \\ &= \frac{k_{\mathrm{U}i} T_{\mathrm{S}i} + C_{\mathrm{S}i}}{k_{\mathrm{U}i} + 1} + \frac{1}{k_{\mathrm{R}i}} \cdot \frac{T_{\mathrm{S}i} - C_{\mathrm{S}i}}{k_{\mathrm{U}i} + 1} \end{aligned} \tag{4.24}$$

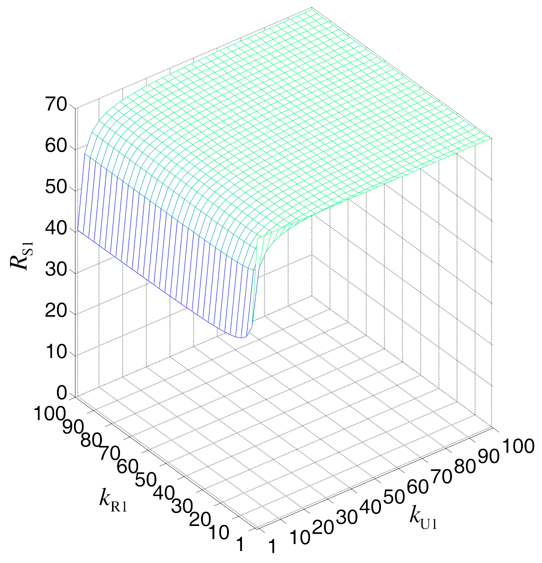
With Equation (4.24), we can determine the extrema of the WCRT shown in Equations (4.25a) - (4.25d).

$$R_{\mathrm{S}i}(k_{\mathrm{U}i} = 1, k_{\mathrm{R}i} = 1) = T_{\mathrm{S}i} \tag{4.25a}$$

$$\lim_{k_{\mathrm{U}i} \to \infty} R_{\mathrm{S}i}(k_{\mathrm{R}i} = 1) = T_{\mathrm{S}i} \tag{4.25b}$$

$$\lim_{k_{\mathrm{R}i} \to \infty} R_{\mathrm{S}i}(k_{\mathrm{U}i} = 1) = \frac{T_{\mathrm{S}i} + C_{\mathrm{S}i}}{2} \tag{4.25c}$$

$$\lim_{k_{\mathrm{R}i} \to \infty} \lim_{k_{\mathrm{U}i} \to \infty} R_{\mathrm{S}i} = T_{\mathrm{S}i} \tag{4.25d}$$



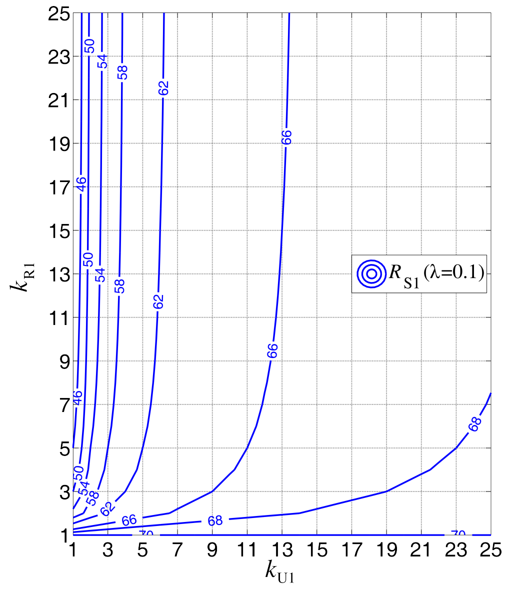


Figure 4.17: *WCRT of $\tau_{S1} = \langle 10, 70 \rangle$: 3D plot (left) and contour plot (right).*

As Equation (4.25c) shows, the shortest WCRT is achieved for $(k_{\mathrm{U}i} = 1, k_{\mathrm{R}i} \to \infty)$. Due to the fact, that the WCRT is always shorter than or equal to the deadline, we can again confirm the feasibility of our method. Figure 4.17 can be used to determine the WCRT of a sporadic task for chosen pairs of $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$.

4.5 Trade-Off between Utilization of Reservation Tasks and Worst-Case Response Times

Section 4.3 showed how we can reduce the utilization of reservation tasks. In Section 4.4, we showed how to reduce the WCRT. In this section, we depict the trade-off between utilization of reservation tasks and WCRT of the sporadic task.
As already mentioned, because of the unknown arrival times guaranteeing sporadic tasks causes overhead. Even when sporadic tasks arrive with maximum frequency, there is RET which the sporadic task instances will not use at runtime. By adapting $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$, we can change the utilization of reservation tasks such that the offline scheduled tasks will be feasible. Assuming the use of a scheduler which is able to schedule periodic task sets with a maximum utilization of U_{sched}, we have to determine pairs of $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$ so that Inequality (4.26) holds. For instance, EDF is capable of exploiting the full processor capacity and thus, U_{sched} is 1 for EDF.

$$\sum_{i=1}^{n} U_i + \sum_{i=1}^{m} U_{\mathrm{R}i} \leq U_{sched} \tag{4.26}$$

Within this constraint, we can reduce the WCRT for a given bound of the reservation tasks' utilization and vice versa. A trade-off between both parameters at the same time is also possible.

We show a feasibility analysis for this trade-off between the utilization of the reservation tasks and the WCRT of the sporadic tasks in the following. We determine pairs of $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$ for given bounds on the WCRT and the utilization of the reservation tasks. As an example, we use one sporadic task τ_1 with the parameters $C_{\mathrm{S}1} = 10$ and $T_{\mathrm{S}1} = 70$. Figure 4.18 shows a contour plot for the feasibility analysis with switching overhead set to 1% of the WCET of the sporadic task ($\lambda = 0.1$). Dashed lines depict the utilization $U_{\mathrm{R}1}$ of the reservation task $\tau_{\mathrm{R}1}$. The maximum available utilization for this reservation restricts the possible pairs of $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$. Solid lines show the WCRT of the sporadic task. The curves represent contour lines with the same values of that variable.
To check the feasibility of given bounds, we determine the area which is surrounded by the contour lines for the given bounds. If there is no pair of $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$ (remember both are natural numbers) for the given bounds, the transformation is infeasible for the given bounds. The procedure is shown in the example in Section 4.6.

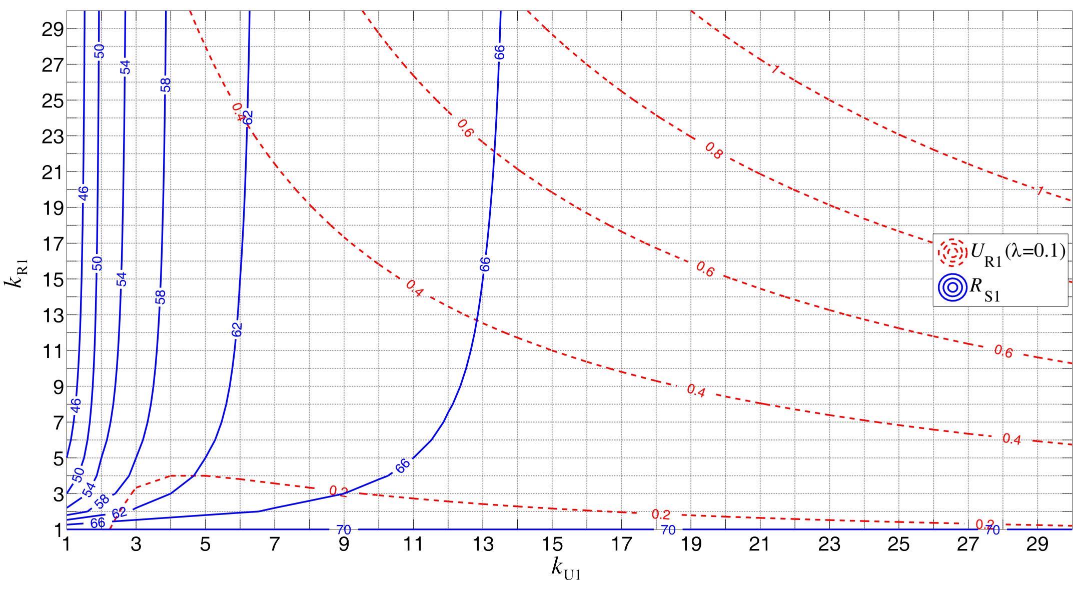


Figure 4.18: *Contour plot for feasibility analysis with* $\lambda = 0.1$.

4.6 Example of the Transformation of Sporadic Tasks

In this section, we present an example: first, we show that deadline misses can occur when sporadic tasks are not considered in the offline scheduling process. After this, we perform a feasibility analysis for given bounds on the utilization of the reservation tasks and the WCRT to determine feasible pairs of k_{Ui} and k_{Ri} for the reservation tasks. To conclude the example, we show that when including the reservation tasks into the offline scheduling process, deadlines are met.

In this example, we use a task set with three periodic tasks and two sporadic tasks, shown in Table 4.4. In a first step, the offline scheduler schedules the periodic tasks and at runtime the two sporadic tasks arrive at the beginning of the schedule, i.e., $t = 0$. Figure 4.19 shows the occurring deadline miss of the sporadic task τ_{S2}.

periodic tasks	WCET	period
τ_1	10	60
τ_2	20	95
τ_3	30	150

sporadic tasks	WCET	period
τ_{S1}	10	100
τ_{S2}	20	110

Table 4.4: *Example task set.*

In the following, we show how we can determine the parameters k_{Ui} and k_{Ri} for given bounds on the utilization and the WCRT. The utilization of the periodic tasks is about 58%. We want to use the remaining 42% to guarantee the sporadic tasks. We test the feasibility of the bounds $U_{R1} \leq 0.16$ and $R_{S1} \leq 100$ for τ_{S1} and $U_{R2} \leq 0.26$ and $R_{S2} \leq 95$ for τ_{S2}. Figure 4.20 shows the given bounds. The areas with feasible pairs of k_{Ui} and k_{Ri}

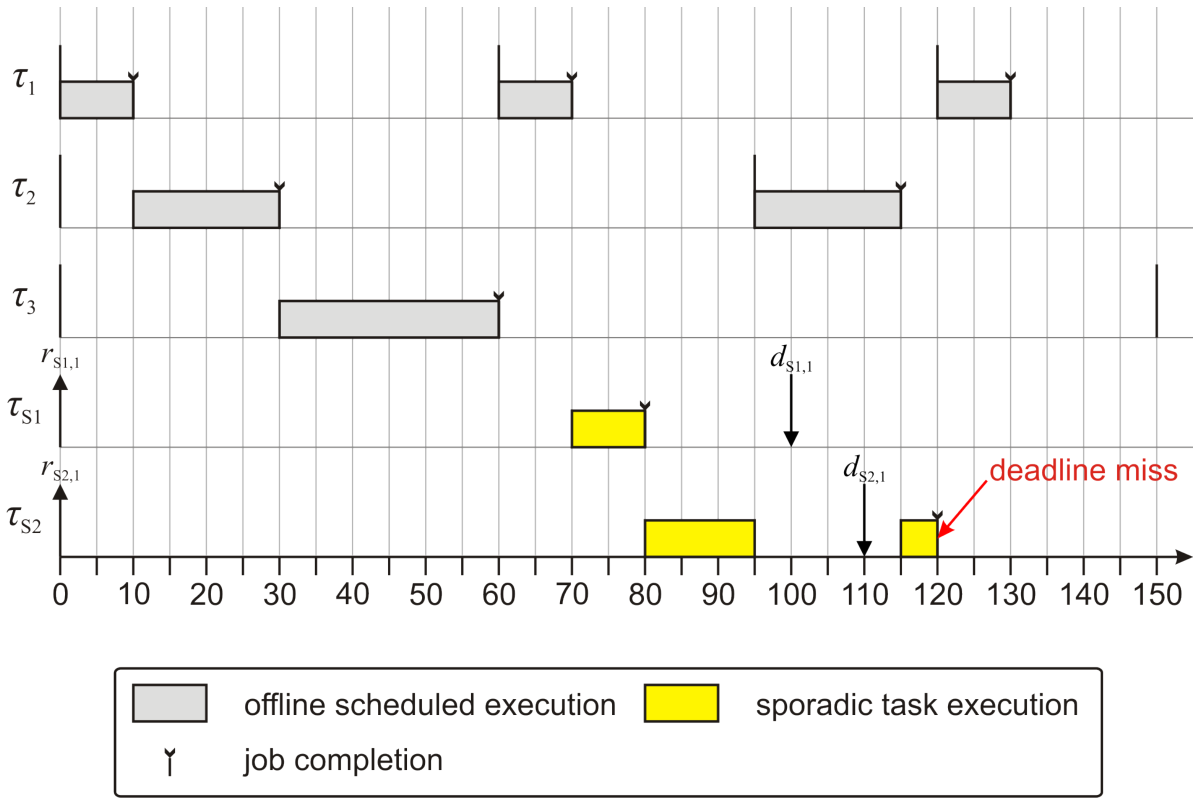


Figure 4.19: *Resulting schedule when sporadic tasks are not considered in the offline scheduling process.*

are highlighted. We choose the pairs of $k_{\mathrm{U}i}$ and $k_{\mathrm{R}i}$ such that the number of slices for each task is minimum. As a result, we choose $(k_{\mathrm{U}1}, k_{\mathrm{R}1}) = (2, 1)$ and $(k_{\mathrm{U}2}, k_{\mathrm{R}2}) = (2, 2)$ (marked in Figure 4.20).

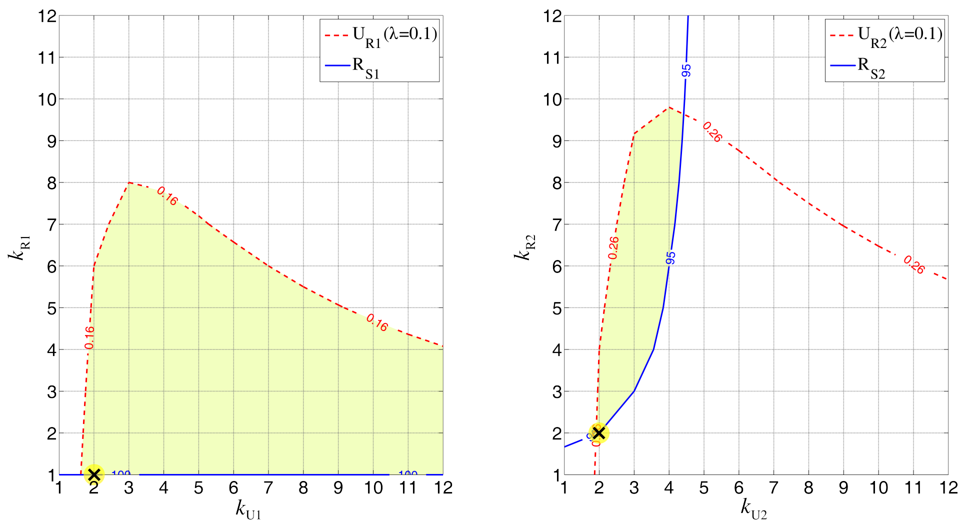


Figure 4.20: *Feasibility analysis of $\tau_{S1} = \langle 10, 100\rangle$ and $\tau_{S2} = \langle 20, 110\rangle$.*

Using the chosen parameters, we transform the sporadic tasks into periodic reservation tasks and schedule them together with the periodic tasks τ_1, τ_2, and τ_3, again. The resulting parameters for the reservation tasks are: $\tau_{R1} = \langle 5, 35 \rangle$ and $\tau_{R2} = \langle 5, 20 \rangle$. The utilizations of the reservation tasks ($U_{R1} \approx 0.143$, $U_{R2} = 0.250$) are below the desired bounds. Figure 4.21 (upper part and highlighted middle part) shows the resulting offline schedule. At runtime, both sporadic tasks are released at the beginning of the schedule, i.e., $t = 0$. Both tasks meet their deadline using the reserved execution time of their allotted reservation task which is shown in the figure (lower part).

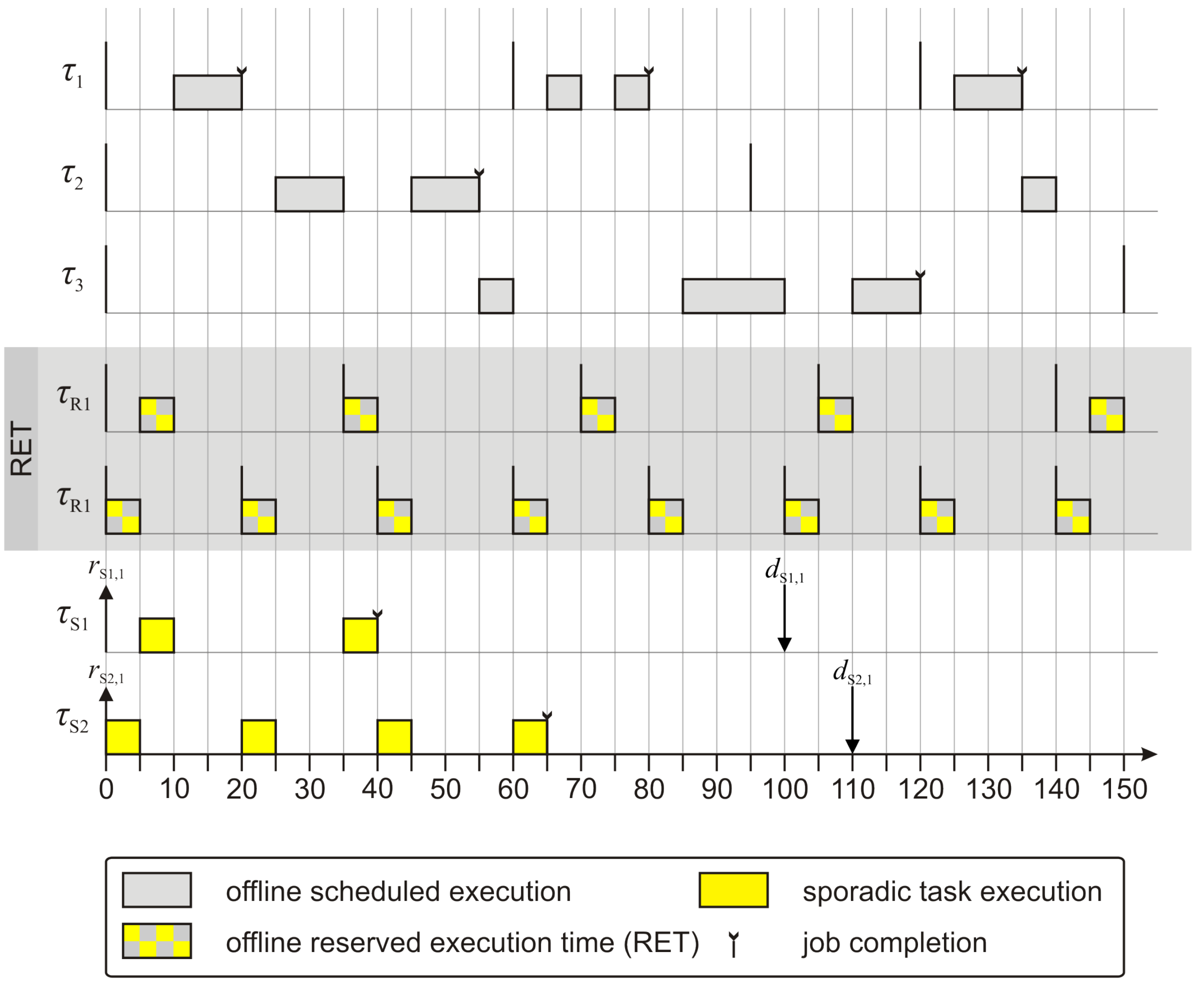


Figure 4.21: *Schedule including reservation tasks into the offline scheduling process.*

4.7 Evaluation: An Application of Parameter Transformation

In this section, we show a possible application of the transformation method shown before. The presented transformation guarantees sporadic tasks even when they arrive with maximum frequency. Sporadic tasks arriving with maximum frequency behave like periodic tasks [ABRW92]. As a consequence, we can also transform periodic task parameters. The disadvantage of transforming periodic tasks is that additional preemptions are introduced and hence, switching overhead is increased. A major reason to apply the transformation despite this disadvantage is the hyper-period. The schedule of synchronized periodic tasks repeats itself after the hyper-period H. The hyper-period is calculated by the LCM of the periods of all tasks. In TT schedule tables, the length of the schedule table is determined by the hyper-period. Thus, shortening the length of the schedule table, i.e., reducing the hyper-period, results in less memory consumption by the schedule table. A shortened hyper-period can be obtained by adjusting the periods or a smart selection of periods within a range, e.g., [Xu10, GM01, RBR12, BBBR11]. In this section, we use our transformation method to generate new periods (and WCETs) for periodic tasks. The idea is to use the presented reservation also on periodic tasks to make use of the additionally calculated periods. In the following, we assume that the feasibility of the determined periods (and WCETs) has been checked and we use only valid periods.

4.7.1 Motivation

Standard methods to calculate the hyper-period, i.e., the LCM, of two integer numbers a and b are, for instance:

Reduction by the greatest common divisor. Calculation of greatest common divisor (GCD) by using an algorithm like Euclidean algorithm and then calculating LCM by $\mathrm{lcm}(a,b) = \frac{|a \cdot b|}{\mathrm{gcd}(a,b)}$;

Prime factorization. Find a product of prime numbers for a and b; LCM is then the product of multiplying the highest power of each prime number of a and b;

There are further algorithms and table-based methods to calculate the LCM. These standard methods are quite computationally complex. Using given task periods can lead to extremely large hyper-periods and thus, to large memory demands. In the following, we show how to generate new periods based on the method presented in this chapter. Furthermore, we show the applicability of our method and the improvements by applying an advanced method to calculate the minimum hyper-period called Fast Hyper-period Search (FHS).

4.7.2 Creation of Periods

In Section 4.4, we showed the extension to include a WCRT optimization into the transformation. We neglect this possibility here for the sake of simplicity and clarity. The LCM is defined on integer numbers, hence, periods have to be integer. As a consequence, to get integer results from our transformation, we adapt the calculation of RETs and periods. Equations (4.27) and (4.28) show the adapted equations for the hyper-period optimization. As a result, the calculated parameters are integer numbers. Further, we replaced the WCET and period of sporadic tasks by the parameters of periodic tasks.

This requirement fits to actual hardware. In processor, granularity of time is the clock cycle, hence, times have to be integer multiples of the length of a clock cycle.

$$C_{\mathrm{R}i} = \left\lceil \frac{C_i}{k_{\mathrm{U}i}} \right\rceil \tag{4.27}$$

$$T_{\mathrm{R}i} = \left\lfloor \frac{T_i + \frac{C_i}{k_{\mathrm{U}i}}}{k_{\mathrm{U}i} + 1} \right\rfloor \tag{4.28}$$

Due to the fact that we consider several periods for one periodic task, we refer to the possible periods like this:

- T_i: original given period
- $T_i(1)$: calculated reservation task period based on $k_{\mathrm{U}i} = 1$
- $T_i(2)$: calculated reservation task period based on $k_{\mathrm{U}i} = 2$
- ...

The calculation of κ *additional* periods results in the set of periods

$$\Gamma_i(k_{\mathrm{U}i}) = \{T_i, T_i(1), T_i(2), \dots, T_i(\kappa)\}$$

of task τ_i. Performing these calculations for all tasks in the task set results in the set $\Gamma_i(k_{\mathrm{U}i})$ which contains $(\kappa + 1)$ elements (periods) per task. The assumptions for the remainder of this section are: Without loss of generality, for all tasks the same number κ of additional periods are calculated. Additionally, all possible periods of the periodic task set are stored in a period matrix $\Gamma(\kappa)$ shown in Equation (4.29).

$$\begin{aligned} \Gamma(\kappa) &= \begin{pmatrix} \Gamma_1(k_{\mathrm{U}1}) \\ \Gamma_2(k_{\mathrm{U}2}) \\ \vdots \\ \Gamma_n(k_{\mathrm{U}n}) \end{pmatrix} \\ &= \begin{pmatrix} T_1 & T_1(1) & T_1(2) & \dots & T_1(\kappa) \\ T_2 & T_2(1) & T_2(2) & \dots & T_2(\kappa) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ T_n & T_n(1) & T_n(2) & \dots & T_n(\kappa) \end{pmatrix} \end{aligned} \tag{4.29}$$

4.7.3 Exhaustive Search Algorithm

As a comparison for the later presented FHS algorithm, we first show a trivial approach to determine the minimum hyper-period based on a set of periods for each task. We calculate the hyper-period for each possible combination of periods which we refer to as Exhaustive Search Algorithm (ESA). Figure 4.22 shows a small example with two tasks with two additional, i.e., in total three, periods. We can see that there are nine possible combinations and hence, we calculate nine times the LCM of two periods. For n tasks with κ additional periods, there are $(\kappa+1)^n$ possible combinations, i.e., a complexity of $\mathcal{O}\left((\kappa+1)^n\right)$. The calculation of the LCM is highly data dependent, i.e., independent of the length of hyper-period, the time to calculate the LCM varies with the chosen periods, which we show in Section 4.7.5 in our experiments. As a result, the length of periods does not influence the length of the hyper-period, i.e., a longer periods of tasks can result in a shorter hyper-period.

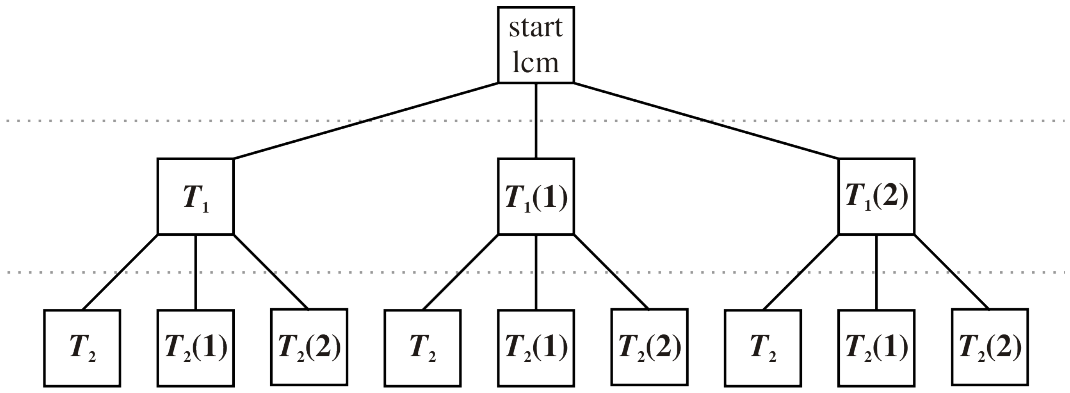


Figure 4.22: *Complexity of hyper-period calculation using ESA with two tasks with each two additional periods.*

The following example shows an exponential increase in calculation times for ESA. We create 10 random tasks with periods between 50 and 500 and generate $\kappa = 10$ additional periods for each task. As a result, the period matrix $\Gamma(\kappa = 10)$ is shown in Equation (4.30). Each row shows the original period and the additionally calculated periods of a task. As the sixth row in the matrix shows, due to rounding, the method can produce the same period several times. This phenomenon does not improve the results but also does not result in wrong results. This period matrix forms the basis of the following runtime measurements. When we measure the runtime to determine the minimum hyper-period for a task set with n tasks and κ additional periods, we select the first n rows and the first $\kappa+1$ columns of the matrix. In other words, the task set with n tasks is always the task set for $n-1$ tasks extended by one additional task.

$$\Gamma(\kappa = 10) = \begin{pmatrix} 410 & 205 & 136 & 102 & 82 & 68 & 58 & 51 & 45 & 41 & 37 \\ 454 & 227 & 151 & 113 & 90 & 75 & 64 & 56 & 50 & 45 & 41 \\ 72 & 36 & 24 & 18 & 14 & 12 & 10 & 9 & 8 & 7 & 6 \\ 458 & 230 & 153 & 114 & 91 & 76 & 65 & 57 & 50 & 45 & 41 \\ 320 & 160 & 106 & 80 & 64 & 53 & 45 & 40 & 35 & 32 & 29 \\ 57 & 28 & 19 & 14 & 11 & 9 & 8 & 7 & 6 & 5 & 5 \\ 146 & 73 & 48 & 36 & 29 & 24 & 20 & 18 & 16 & 14 & 13 \\ 278 & 139 & 92 & 69 & 55 & 46 & 39 & 34 & 30 & 27 & 25 \\ 480 & 240 & 160 & 120 & 96 & 80 & 68 & 60 & 53 & 48 & 43 \\ 483 & 241 & 161 & 120 & 96 & 80 & 69 & 60 & 53 & 48 & 43 \end{pmatrix} \tag{4.30}$$

We perform our experiments on an Intel XEON processor E5-2670 running at 2.60 GHz and with 4 GB equipped main memory. Based on shown period matrix, we calculate the minimum hyper-period using ESA and show the runtimes of it. Table 4.5 depicts the runtime results for 2 to 10 tasks. As the first column of the runtime results shows, the calculation of the LCM with only the original period per task takes only little time. With increasing number of periods and/or tasks, the runtimes increase drastically. Whereas increasing the number of tasks has – as expected – an exponential impact on the runtimes. For more than eight tasks, runtimes already go up to several seconds. The calculation of the minimum hyper-period for 10 tasks with 10 additional periods even takes more than 50 minutes.

no. of tasks	time unit	T_1	$T_1(1)$	$T_1(2)$	$T_1(3)$	$T_1(4)$	$T_1(5)$	$T_1(6)$	$T_1(7)$	$T_1(8)$	$T_1(9)$	$T_1(10)$
2		1	1	1	2	3	4	6	7	9	11	14
3		1	1	5	9	16	25	38	55	74	101	132
4	μs	1	4	15	41	95	184	335	551	846	1244	1789
5		1	7	46	166	472	1119	2237	4174	7229	11904	19145
6		1	15	137	638	2205	5997	14079	29987	58083	104820	182269
7		1	33	411	2496	11525	37614	101851	244623	537399	1083729	2115938
8		< 1	< 1	< 1	< 1	< 1	< 1	< 1	2	6	13	27
9	s	< 1	< 1	< 1	< 1	< 1	1	5	15	43	108	266
10		< 1	< 1	< 1	< 1	1	8	38	133	434	1199	3213

Table 4.5: *Runtimes to calculate the minimum hyper-period with ESA.*

In Figure 4.23, we depict the increasing runtimes for increasing number of additional periods. The y-axis shows the runtimes in microseconds on a logarithmic scale. Further, the figure also shows that although the figure depicts runtimes in relation to periods, but by the distance between curves, we can see an additional task also results in a runtime increase by about an order of magnitude. Further experiments confirmed the shown trend in this example and confirm the expected increase in runtimes. In summary, the results show that ESA is hardly applicable in real systems because of extremely long runtimes for more than just a few tasks.

4.7.4 Fast Hyper-Period Search Algorithm

As we showed before, we need a more efficient method to calculate the minimum hyper-period. Brocal et al. proposed an advanced method to calculate minimum hyper-period in [BBBR11]. They assume that tasks do not have a single period, but the period is given as range of integer numbers, e.g., period is not given as 70 but as a range of integers $[68, 72] \equiv \{68, 69, 70, 71, 72\}$. After the selection of a period at design time, this period is used at runtime for the lifetime of the tasks, i.e., there is no change of the periods at runtime. Their method to calculate the minimum hyper-period is called Fast Hyper-period Search (FHS).

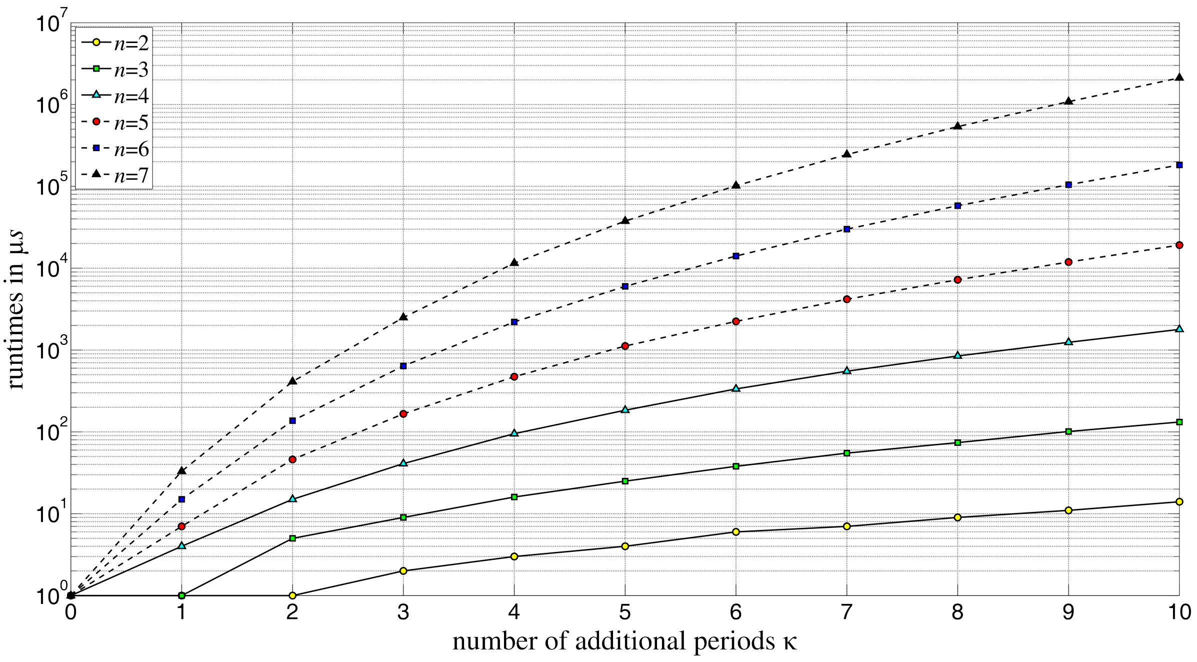


Figure 4.23: *ESA runtimes to calculate the minimum hyper-period.*

FHS performs the calculation of the minimum hyper-period in the following steps:

1. A hyper-period based on the original period of each task is calculated as first reference hyper-period.
2. All possible period combinations for the LCM of the first η tasks are calculated. Hence, the larger η is, the more time consuming these calculations are (see ESA in Section 4.7.3).
3. Next, the list of these possible hyper-periods is sorted in ascending order.
4. Each element of the list is used to check whether this a possible hyper-period of the remaining tasks. If there is a valid period for each of the remaining tasks, the element is set as the new shortest hyper-period. If there is at least one task in whose period range is not a valid divisor, the next element in the list is checked until the next element is greater than the currently shortest hyper-period. (reference hyper-period).
5. In this step, the list of sorted periods is multiplied by an iteration counter $d + 1$ (for original sorted list $d = 1$) and step 4 is repeated. This process is repeated until the first element of the multiplied sorted list is the new shortest hyper-period (reference hyper-period) or the first element is larger than the current reference hyper-period.

For more details about the algorithm, we refer the interested reader to [BBBR11]. For the first η tasks all possible hyper-periods are calculated which is analogue to ESA. In the next section, we show the runtime behavior of FHS for minimum η, i.e., $\eta = 2$, to illustrate the runtime improvements. Additionally, we show the influence of larger values for η. Finally, we show that by generating new periods, the hyper-period can be drastically reduced in a reasonable amount of time by using FHS.

4.7.5 Evaluation of the Effectiveness and Efficiency

We performed our runtime experiments on an Intel XEON processor E5-2670 running at 2.60 GHz. Each experiment consists of calculating the minimum hyper-period for $n \in \{2, ..., 60\}$ tasks with $\kappa = \{7, ..., 10\}$ additional periods. We generate task periods randomly between 50 and 1000 time units. The selection of tasks is done as before, i.e., a task set is always the task set as before extended by one additional task or period, respectively. The presented results are based on runtimes of 100 experiments. In addition, we measured runtimes with a granularity, i.e., also minimum measured runtime, of 1 μs. First test runs showed extremely long runtimes for small κ and shorter runtimes for increasing κ. Thus, we performed experiments for up to 60 tasks for 7 to 10 additional periods. Figure 4.24 shows mean runtimes results for $\eta = 2$ using FHS algorithm.

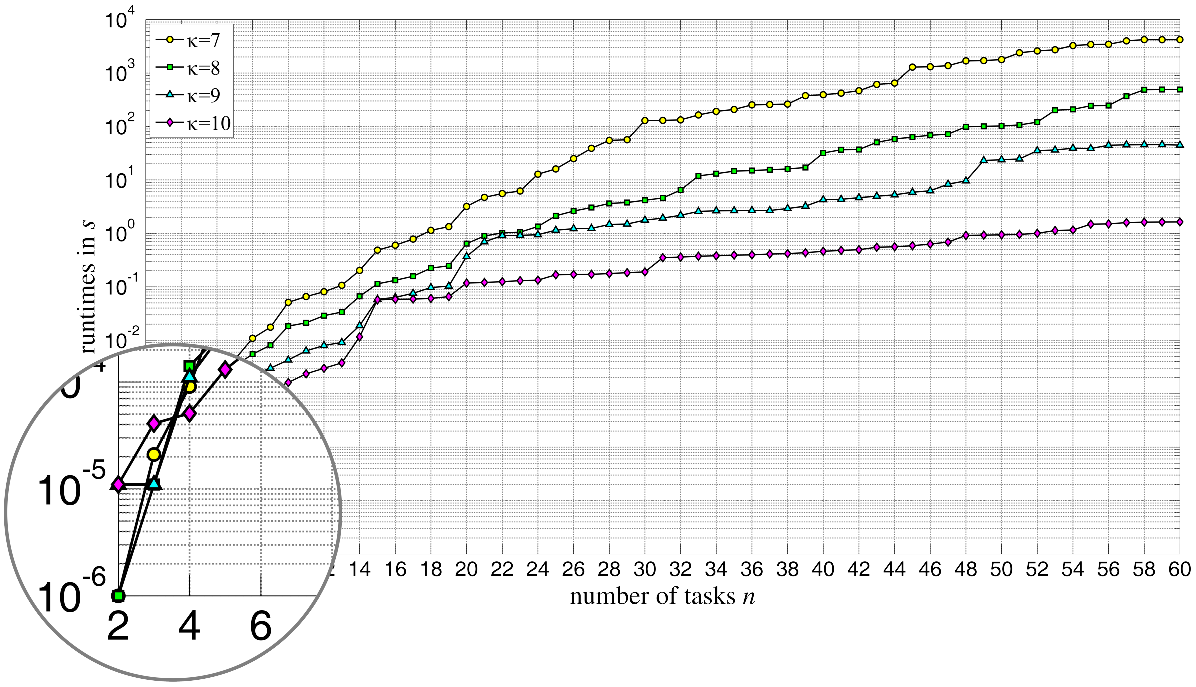


Figure 4.24: *Mean runtimes of FHS to calculate the minimum hyper-period.*

In general, runtimes strongly increase with increasing number of tasks. The slope of the curves flatten for higher number of tasks which can be explained by the restricted range of periods, i.e., at some point it is likely that a new period is a multiple or divisor of an already existing period and hence, does not strongly influence the hyper-period calculations. Furthermore, with increasing number of additional periods, the mean runtimes decrease. With more possible periods, the list of possible periods in step 2 is larger and hence, it is more likely to find a possible hyper-period in this set which results in shorter runtimes. As enlarged in the figure, for only a few tasks, runtimes are shorter with less additional periods, because the overhead of calculating the set of possible hyper-periods takes longer than the procedures afterwards.

As we mentioned before, the calculation of the hyper-periods is highly data-dependent. As an example, Figure 4.25 on page 61 shows a box plot of the measured runtimes including the mean runtimes for $\kappa = 8$ and Figure 4.26 on page 62 presents all measured runtimes in a scatter plot. The results for $\kappa = \{7, 9, 10\}$ are shown in the appendix in Section A.1. The majority of runtimes is even below the mean runtimes but several much larger exceptions, with a difference of one to two orders of magnitude, increase the mean runtimes. The range of runtimes is between four and seven orders of magnitude which confirms the data-dependency of the hyper-period calculation.

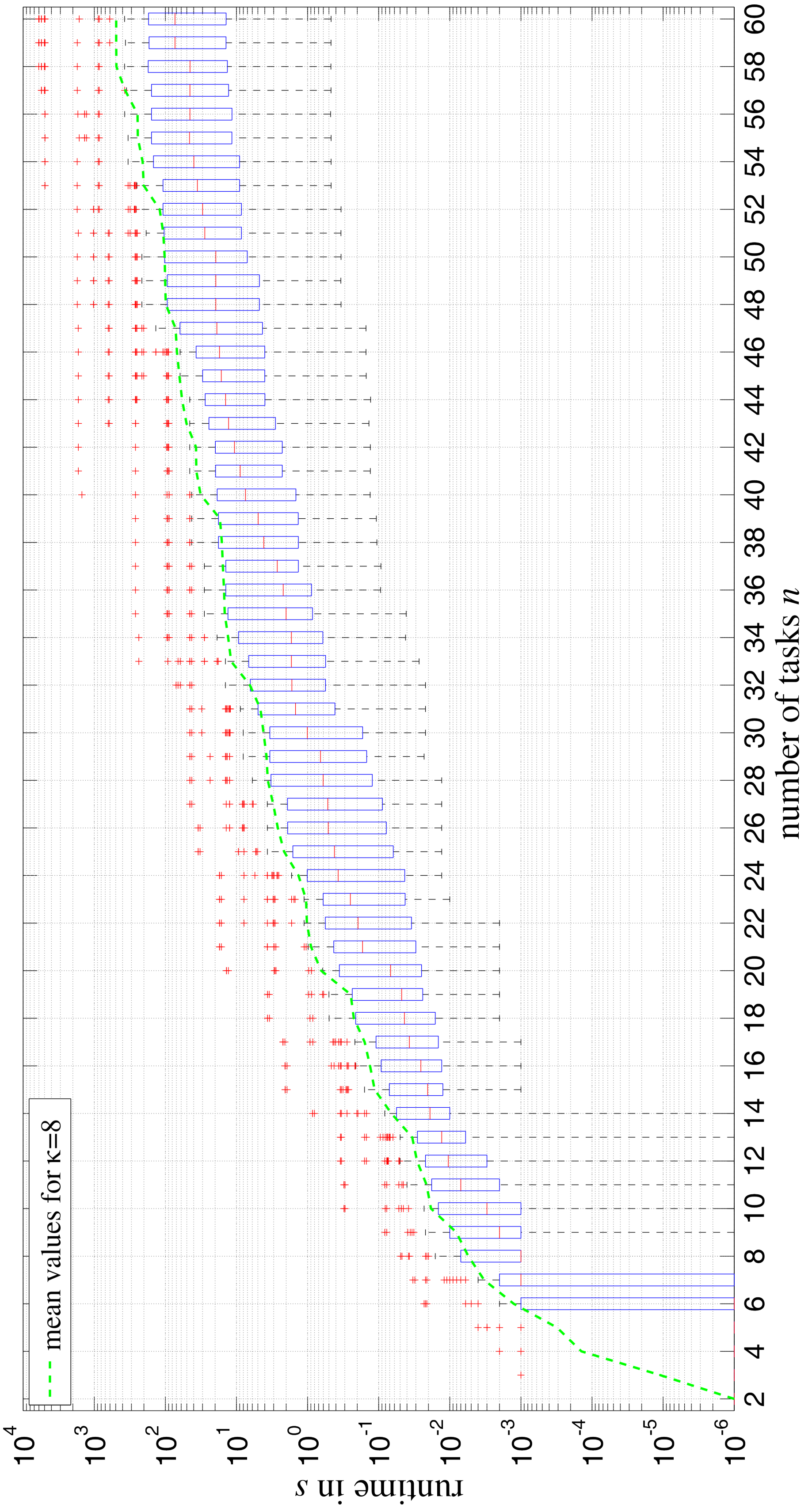


Figure 4.25: *Box plot of FHS runtimes and mean runtimes of FHS*

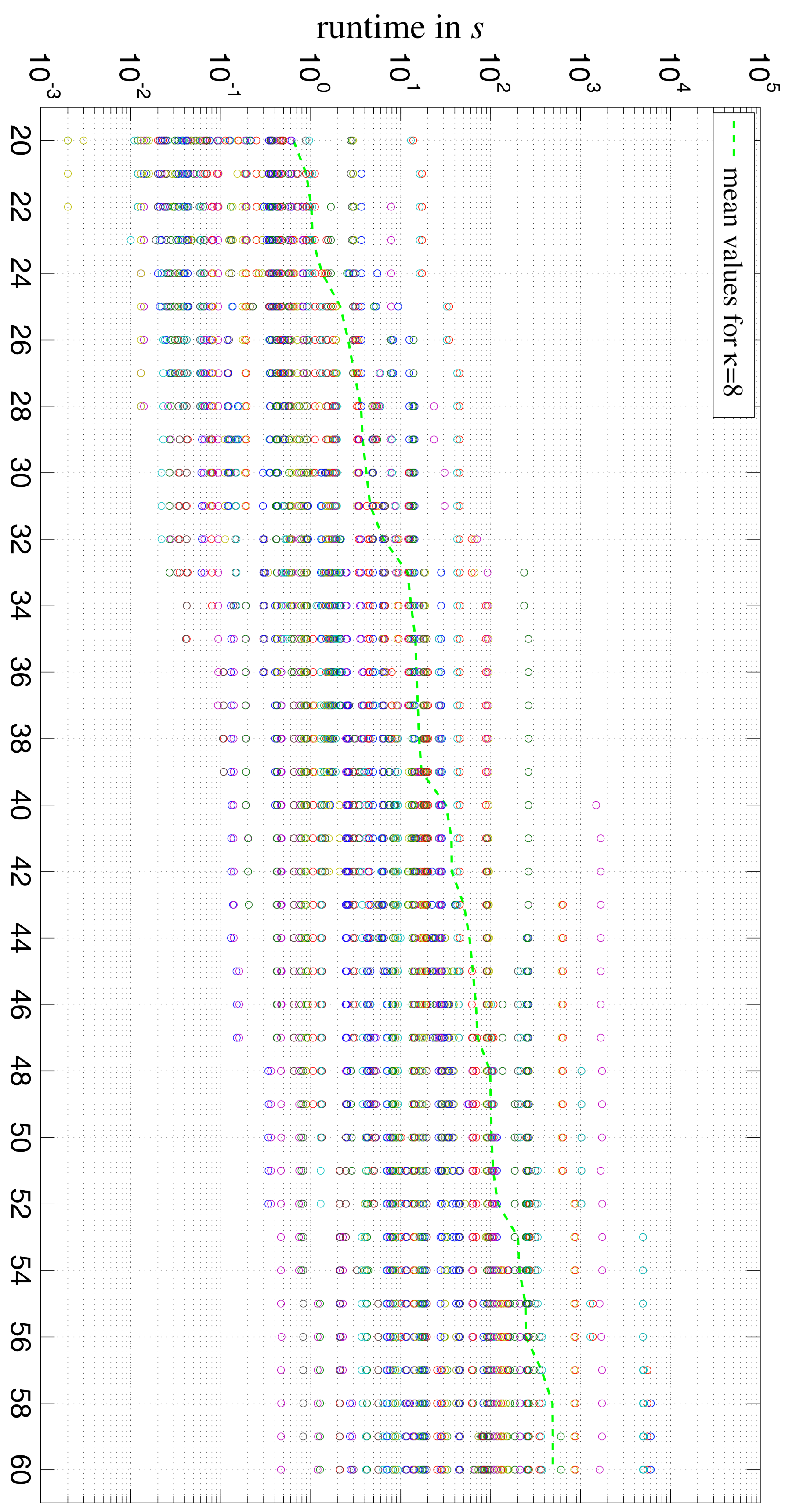


Figure 4.26: *Scatter plot of FHS runtimes and mean runtimes of FHS*

The results shown so far based on $\eta = 2$, i.e., minimizing the influence of the calculation of the set of possible hyper-periods by calculating all possible combinations. In Figure 4.27, we show the mean runtimes for $\kappa = 7$ and $\kappa = 8$ with $\eta \in \{2, 4, 5\}$. We show runtime results for at least 10 tasks to avoid just showing the runtimes of the calculation of the set of possible hyper-periods. Both graphs in the figure show that higher η results in longer runtimes. This results from the fact that in each iteration step the set of possible hyper-periods is multiplied by the iteration counter d which is time consuming. As a consequence, larger η results in a larger set of possible hyper-periods and hence, the runtimes are strongly affected by the multiplications. The impact of η is strong because of the number of possible hyper-periods in the initial set is $(\kappa + 1)^{\eta}$.

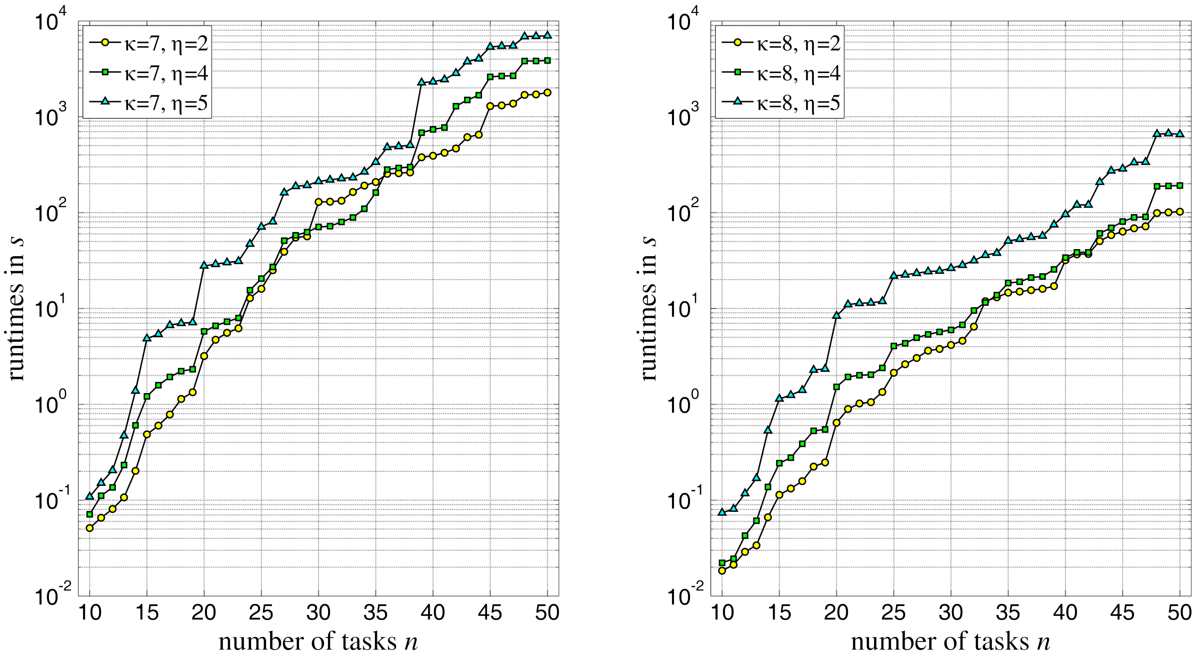


Figure 4.27: *Mean runtimes for FHS with varying η.*

Based on the results of the runtimes shown before, we can see that the minimum hyper-period can be calculated within seconds up to a few hours for up to 60 tasks with 10 additional periods. On the contrary, ESA showed already for 10 tasks with 10 additional periods runtimes of about one hour. As a consequence of using FHS, we can efficiently apply our transformation method of creating additional periods. In the following, we show the effectiveness of our method, i.e., that we can drastically improve the hyper-period and hence, reduce memory consumption of TT schedule tables. To illustrate the improvements, we show the results for an representative example with 30 tasks with $\eta = 2$ and $\kappa \in \{7, 8, 9, 10\}$. Figure 4.28 compares the minimized hyper-period for $\kappa \in \{7, 8, 9, 10\}$ with the hyper-period determined without additional periods. As a result,we can see that in all four cases, all 100 experiments showed an improvement in the length of the hyper-period.

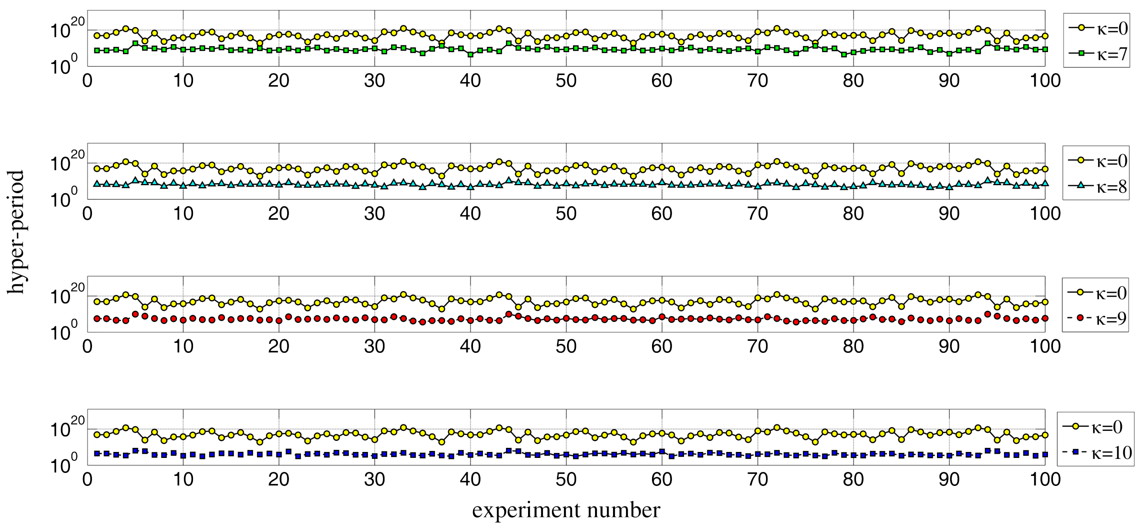


Figure 4.28: *Optimized hyper-period for $n = 30$ and $\eta = 2$: comparison to reference hyper-period.*

Figure 4.29 compares the hyper-periods for four different values of κ. We can confirm the expected result that with more additional periods the optimized hyper-period is never larger than the one with less additional periods. When comparing Figure 4.29 with Figure 4.28, we can also show the drastic improvement of the hyper-period. In summary, the optimized hyper-periods are below 10^{13} time units whereas the original hyper-periods are up to 10^{20} time units.

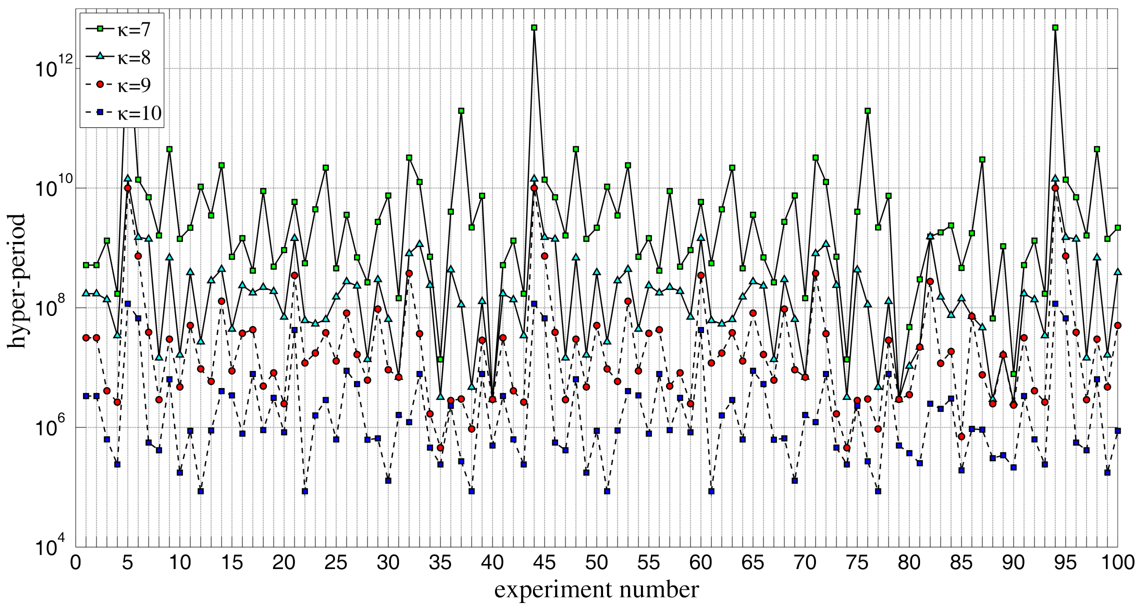


Figure 4.29: *Optimized hyper-period for $n = 30$ and $\eta = 2$: comparison for different number of additional periods κ*

To summarize the experiments in this section, our method to transform sporadic tasks can also be applied to transform periodic tasks. We gain new periods which can be used to optimized the hyper-period of TT schedule tables to reduce memory consumption and shorten scheduling process. Further, by applying FHS, we can make use of the additional periods to efficiently and effectively minimize the hyper-period.

Chapter 5

Time-Triggered Schedule Tables with Mode Changes

In this chapter, we show methods for the effective and efficient construction of time-triggered (TT) schedule tables. The methods focus on the construction of two modes, each with one schedule table, for mixed-criticality systems with two criticality levels, so-called dual-criticality systems. We present different methods that altogether form an algorithm to construct the schedule tables. The first method separates the demand of jobs determined by the designers from the additionally needed demand to fulfill the HI-criticality worst-case execution times (WCETs) of the Certification Authorities (CAs). We split HI-criticality jobs into two jobs to obtain jobs with only one WCET. In the next step, we construct the TT schedule tables. The schedule tables are created slot by slot starting in the LO-criticality table and then, scheduling the same slot in the HI-criticality table. An important step is then the calculation of the *leeway*. The leeway of a slot determines whether the job in this slot can meet its deadline, including future demand of this job, or not. Instead of scheduling until a deadline miss occurs, we can detect infeasible schedules earlier. By doing this, we can reduce the complexity of the scheduling process. If the leeway is negative, then we apply our backtracking method called swapping. We swap the scheduling decisions of two slots such that both jobs in the slots can meet their deadline. In contrast to standard backtracking, we do not need to check all possible scheduling decisions in the previous schedule but only slots with specific leeway values. The advantage of swapping is a strong reduction in the complexity of the backtracking process.

In Section 5.1, we describe the general behavior of TT mode changes. Next, we show our method to separate the designers' based demand from the demand introduced by CAs' pessimistic assumptions in Section 5.2. Further, Section 5.3 presents the method to construct the schedule tables. As mentioned before, a possible backtracking procedure is needed which is shown in Section 5.4. In Section 5.5, we illustrate our algorithm with an example. The evaluation in Section 5.6 shows the effectiveness and efficiency of our algorithm. Next, Section 5.7 and Section 5.8 discuss extensions and open questions. Finally, Section 5.9 concludes the chapter with a discussion.

5.1 Time-Triggered Mode Changes

Modern embedded systems are often composed of safety-critical applications which are subject to certification. In these systems, there is a trend to combine highly safety-critical with less safety-critical applications on the same platform. In avionics, for instance, design approaches are moving from the federated approach (FA) to the integrated modular avionics (IMA) approach. A common approach for these systems is a spatial and temporal isolation between activities, e.g., in ARINC [ARI03]. As a consequence of the complete isolation, resources can only sub-optimally be used. As a result, task and job sets are composed of a combination of tasks and jobs which are subject to certification and less critical tasks and jobs which are not subject to certificcation. Hence, the problem is to schedule mixed-criticality task and job sets. Thus, there is a need for guaranteeing tasks and/or jobs with different requirements and assumptions. In our approach, we make use of time-triggered (TT) schedule tables with mode changes to meet designer and certification requirements while making efficient use of resources.

In this chapter, we use the Vestal mixed-criticality model, shown in Chapter 3, with two criticality levels LO and HI. The presented methods work on job basis, hence, when we talk of jobs, we make no difference between single jobs or jobs which are instances of a periodic task. Further, we assume that all jobs have known release times, worst-case execution times (WCETs) and deadlines.

5.1.1 General Discussion and Assumptions

In the following, we describe and discuss the basic elements of our approach. First, we define the TT approach and continue with the mixed-criticality assumptions for our method. Finally, we show the issues we have to consider when implementing mode changes in our approach. TT architectures (TTA) [KB03] are often used for safety-critical applications. These architectures are characterized by global time base and complete determinism of the TT components.

Definition 5.1. Time-Triggered Architecture [Kop11]
A distributed computer architecture for real-time applications, where all components are aware of the progression of the global time and where most actions are triggered by the progression of this global time. ◁

In our approach, the execution of all jobs is only triggered by progression of time. This TT paradigm [Kop11] is characterized by complete determinism which simplifies verification and thus, certification.

We assume a dual-criticality system in which mixed-criticality jobs are characterized by two WCETs. The two WCETs represent the different assumptions of designers and certification authorities (CAs) about the confidence in the bound on the execution times. By the nature of the system, HI-criticality jobs have higher importance than LO-criticality jobs. For instance, flight-critical (HI-criticality) jobs, e.g., guidance, navigation, and control, in Unmanned Aerial Vehicles (UAVs) are more important than so-called mission critical (LO-criticality) jobs, e.g., video recording and streaming. Failing

LO-criticality jobs degrade the performance of the system but failing HI-criticality jobs can lead to destruction and loss of the system and even to dangers for human beings. As a result, meeting deadlines is important for both job types but in an emergency, HI-criticality jobs are prioritized. This prioritization is represented in the scheduler mode: schedule tables are constructed such that only HI-criticality jobs are executed in case of an unexpected behavior, i.e., one or several HI-criticality tasks exceeding their designer-based WCETs.

In the past, mode changes were used to accommodate for different behaviors of processes, e.g., aircraft control systems during different phases like take off, flight, and landing. In the context of mixed-criticality systems, we use mode changes to fulfill the requirements of the designers and the CAs. We construct one mode for each criticality level, here, for dual-criticality, we use two modes: one mode fulfilling designer requirements, i.e., all jobs must meet their deadlines with WCETs based on designers' assumptions; second mode fulfilling CAs' requirements, i.e., HI-criticality must meet their deadlines with WCETs based on CAs' pessimistic assumptions to obtain a high confidence in the execution time bounds. CAs certify only HI-criticality jobs and hence, are not interested in the behavior of LO-criticality jobs as long as they do not interfere with HI-criticality jobs.

In general, it is possible to construct modes such that during a mode change a transition mode is executed to switch from one to another mode. In our approach, we construct modes, i.e., schedule tables, such that there is no transition mode necessary. Additionally, we construct the schedule tables such that switching from LO-criticality to HI-criticality mode is possible at every time instant without violation of the certified HI-criticality jobs. In the following, we refer to this property as *switch-through* property. If modes are not constructed with switch-through property, we need to determine blackout slots [Foh94], i.e., slots in the schedule in which we cannot switch from the current mode to the destination mode.

In summary, we construct two schedule tables to meet designers' and CAs' requirements such that a change of operational mode from LO to HI is possible at every time instant.

5.1.2 Schedule Tables with Mode Changes

The pre-computed schedule tables represent schedule decisions for the entire runtime of the system. At the end of a schedule table, the table is repeated. Although switching back from HI-criticality to LO-criticality mode is not specified in the problem statement, e.g., in [BF11], we can safely switch back to LO-criticality mode at the end of a schedule table.

Baruah and Fohler showed that mode change schedulers, e.g., [Foh94], can be used to accommodate for demands of mixed-criticality job sets [BF11]. The different modes are used to accommodate the different requirements of LO- and HI-criticality behavior. They create for each mode one schedule table, i.e., for dual-criticality systems two schedule tables. Due to the fact that the table for HI-criticality behavior has to be certified, we have to guarantee not missing a deadline under CAs' pessimistic assumptions. For the certification, it is sufficient to guarantee only the HI-criticality jobs based on CAs' assumptions and hence, the HI-criticality schedule table (mode) needs only to contain HI-criticality jobs. In Baruah's and Fohler's approach [BF11], they include LO-criticality jobs into the HI-criticality mode to obtain the switch-through property.

Determining whether a mixed-criticality job set is schedulable by a TT scheduler, is highly intractable [BF11]. Thus, there can be no polynomial or pseudo-polynomial time algorithm for constructing the LO- and HI-criticality schedule table for any TT-schedulable mixed-criticality job set [BF11].

5.1.3 Runtime Behavior

Our goal is to construct the schedule tables such that we obtain schedule tables with switch-through property. In other words, at every time-instance t in the HI-criticality table, there must not be more time reserved for each job before t than in the LO-criticality schedule table. In the following, we refer to the LO-criticality schedule table as LO-*table* and to the HI-criticality schedule table as HI-*table*. Figure 5.1 presents a counter-example, i.e., two schedule tables without switch-through, to illustrate the problem of constructing the schedule tables. Switching from LO-*table* to HI-*table* at $t = 3$ leads to violation of J_i's required WCET for the HI-criticality case. At design time, the construction of the schedule tables has to ensure the switch-through property.

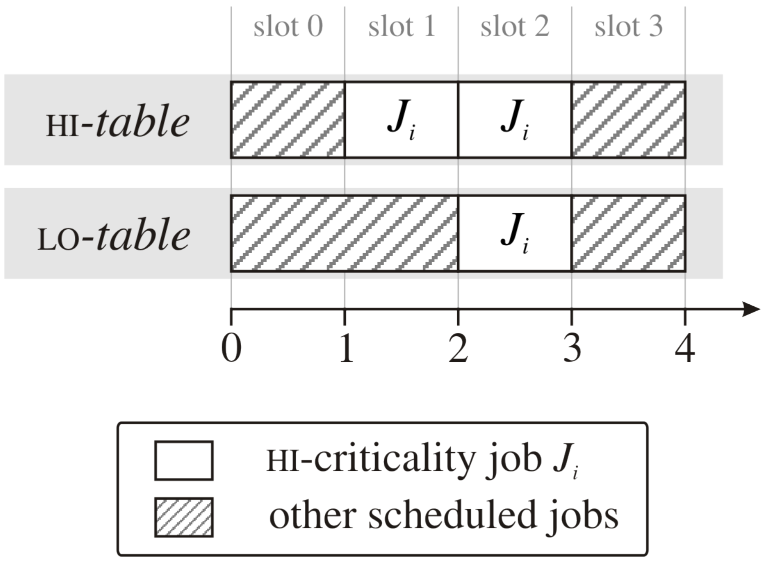


Figure 5.1: *Counter-example: although* HI-*table guarantees CAs' requirements for* J_i*, the missing switch-through property prevents correct system behavior.*

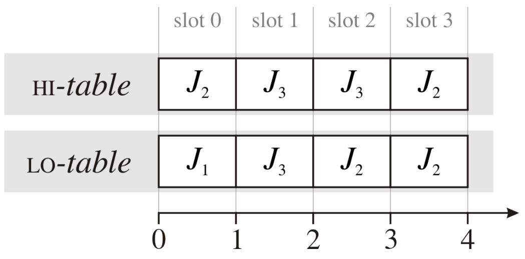
At runtime, the system starts in LO-criticality mode, i.e., using schedule table LO-*table* with designers' assumptions. Harming designers' assumptions, i.e., exceeding $C_i(\text{LO})$ of a HI-criticality job leads to a switch to HI-criticality mode, i.e., using schedule table HI-*table*. After the mode change, we can guarantee that all HI-criticality jobs are provided at least their $C_i(\text{HI})$ in total.

Example. In the following, we present an example to illustrate that obtaining the switch-through property is not trivial by constructing two schedule tables based on pure EDF. The example shows that traditional scheduling criteria, e.g., deadlines (based on EDF), are not enough to create the TT schedule tables. Table 5.1 shows the given job set for the example.

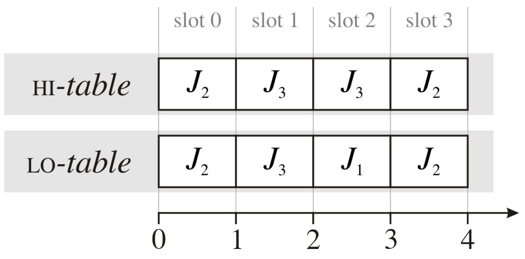
	χ_i	r_i	d_i	$C_i(\text{LO})$	$C_i(\text{HI})$
J_1	LO	0	3	1	1
J_2	HI	0	4	2	2
J_3	HI	1	3	1	2

Table 5.1: *Example mixed-criticality job set.*

The goal is to construct two TT schedule tables for which switching from LO-*table* to HI-*table* is possible at every time-instant in case of a HI-criticality job exceeding its $C_i(\text{LO})$. First, we create both schedule tables based on pure EDF. In Figure 5.2(a), HI-*table* is constructed independently of LO-*table*.



(a) Independently constructed schedule tables based on EDF.



(b) Schedule tables with necessary switch-through property.

Figure 5.2: *Schedule tables to illustrate that traditional scheduling criteria are not enough.*

Switching from LO-*table* to HI-*table* at $t = 2$, due to J_3 showing HI-criticality behavior, leads to not enough resources for J_2. As a consequence, traditional scheduling criteria as, e.g., deadlines, are not enough. When creating the schedule tables, we have to consider a possible HI-criticality behavior of HI-criticality jobs. Figure 5.2(b) shows a possible solution that allows for switching at every time-instant (switch-through property). ◊

5.2 Allocation of Jobs to Modes

In this section, we present how we separate demand based on designers' assumptions and *additional* demand by CAs' pessimistic assumptions for HI-criticality jobs. The construction of the schedule tables takes scheduling decision with the granularity of slots, i.e., the scheduler is preemptive at slot borders.

In a first step, we split HI-criticality jobs to separate the LO-criticality WCET and the additionally needed WCET in the HI-criticality case. As a result, we obtain jobs with only one WCET each. Remember that LO-criticality jobs can be excluded in the HI-criticality schedule table.

LO-criticality jobs J_i, which are only present in LO-*table*, are not split and the WCET C_i is set to $C_i(\text{LO})$. The remaining parameters of these jobs remain unchanged. Each HI-criticality job J_i is split into a job J_i^{LO}, which is the portion present in both tables and a job J_i^{Δ}, which is the portion that is additionally needed in HI-criticality case. The calculation of the WCETs, which we will use as input for our scheduler, are shown in Equations (5.1) - (5.3). Note that splitting HI-criticality job J_i results in two jobs J_i^{LO} and J_i^{Δ} both with the same index i. The difference is the super-script: LO for the portion present in both tables and Δ for the portion that is additionally needed in the HI-criticality case. Both jobs feature the same sub-scripts as the original job.

$$J_i : \qquad C_i = C_i(\text{LO}) \qquad \text{if } \chi_i = \text{LO} \tag{5.1}$$

$$J_i^{\text{LO}} : \qquad C_i^{\text{LO}} = C_i(\text{LO}) \qquad \text{if } \chi_i = \text{HI} \tag{5.2}$$

$$J_i^{\Delta} : \qquad C_i^{\Delta} = C_i(\text{HI}) - C_i(\text{LO}) \qquad \text{if } \chi_i = \text{HI} \tag{5.3}$$

In Figure 5.3, r_i and d_i represent the original release time and deadline, respectively. For the split jobs, we now derive new parameters based on the original parameters of the HI-criticality job. The release time r_i^{LO} of J_i^{LO} is equal to original release time r_i:

$$r_i^{\text{LO}} = r_i \tag{5.4}$$

The deadline of J_i^{LO} must be set early enough such that there remains enough time to schedule the additionally needed WCET in the HI-criticality case. As a result, the deadline of J_i^{LO} is set to

$$\begin{aligned} d_i^{\text{LO}} &= d_i - C_i^{\Delta} \\ &= d_i - \left[C_i(\text{HI}) - C_i(\text{LO})\right]. \end{aligned} \tag{5.5}$$

The WCET of J_i^{Δ} represents the temporal portion which is additionally needed in the HI-criticality case. The earliest possible release time of J_i^{Δ} occurs when its corresponding job J_i^{LO} is scheduled directly at the beginning of the execution window. Hence, we set the release time of a job J_i^{Δ} to

$$\begin{aligned} r_i^{\Delta} &= r_i + C_i^{\text{LO}} \\ &= r_i + C_i(\text{LO}). \end{aligned} \tag{5.6}$$

The deadline of J_i^{Δ} is equal to the deadline of its corresponding original job J_i:

$$d_i^{\Delta} = d_i \tag{5.7}$$

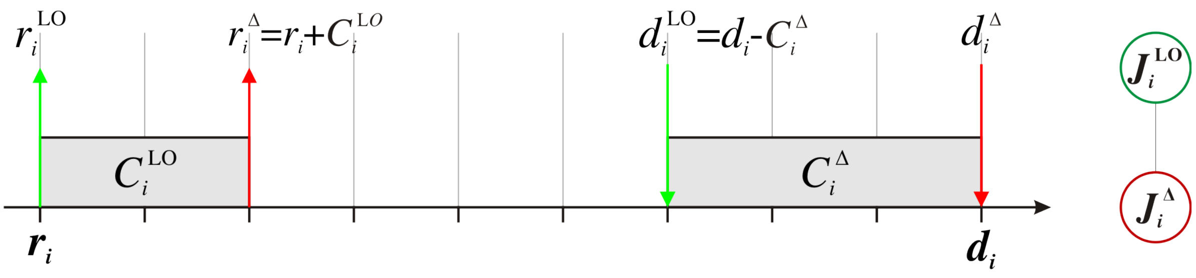


Figure 5.3: *Derivation of parameters for split* HI*-criticality jobs.*

This parameter assignment results in maximum slack for both J_i^{LO} and J_i^{Δ}. To avoid that J_i^{Δ} is scheduled before J_i^{LO}, we **add a precedence constraint** between them: $J_i^{\text{LO}} \prec J_i^{\Delta}$, i.e., J_i^{Δ} cannot be scheduled before J_i^{LO} is completely scheduled. As a consequence, we can guarantee with J_i^{Δ} that in HI-criticality case the additionally needed WCET is scheduled. The two resulting jobs J_i^{LO} and J_i^{Δ} of a split HI-criticality job J_i keep their criticality level HI. Table 5.2 gives an overview of the denomination of the original and the split jobs' parameters.

original jobs					**split jobs**				
job	release time	deadline	WCETs	criticality	job	release time	deadline	WCET	criticality
J_i	r_i	d_i	$C_i(\text{LO}) = C_i(\text{HI})$	$\chi_i = \text{LO}$	J_i	r_i	d_i	C_i	$\chi_i = \text{LO}$
J_i	r_i	d_i	$C_i(\text{LO}) \leq C_i(\text{HI})$	$\chi_i = \text{HI}$	J_i^{LO}	r_i^{LO}	d_i^{LO}	C_i^{LO}	$\chi_i^{\text{LO}} = \text{HI}$
					J_i^{Δ}	r_i^{Δ}	d_i^{Δ}	C_i^{Δ}	$\chi_i^{\Delta} = \text{HI}$

Table 5.2: *Overview of job parameters.*

The input job sets for the scheduler are represented by $S(\text{LO})$ for LO-criticality mode and by $S(\text{HI})$ for HI-criticality mode. The LO-criticality table LO-*table* contains all jobs in the set $S(\text{LO})$ while the HI-criticality table HI-*table* contains all jobs in the set $S(\text{HI})$ with:

$$S(\text{LO}) = \{J_i, J_k^{\text{LO}}\} \qquad \text{with } (\chi_i = \text{LO}) \wedge (\chi_k^{\text{LO}} = \text{HI})\, \forall \{i,k\} \in \{1,...,n\} \tag{5.8a}$$

$$S(\text{HI}) = \{J_i^{\text{LO}}, J_k^{\Delta}\} \qquad \text{with } (\chi_i^{\text{LO}} = \text{HI}) \wedge \left(\chi_k^{\Delta} = \text{HI}\right) \forall \{i,k\} \in \{1,...,n\} \tag{5.8b}$$

By splitting the jobs, we now can schedule the job sets $S(\text{LO})$ and $S(\text{HI})$ to satisfy the switch-through property. We ensure this by scheduling jobs J_i^{LO}, which are present in both modes, at the same time in both tables and obeying the precedence constraints.

5.3 Time-Triggered Schedule Table Generation

In the following, we create one schedule table for each of the two modes. The modes are characterized by their schedule table and thus, we use the terms schedule table and mode for LO-*table* and HI-*table* interchangeably.

Based on the job sets $S(\text{LO})$ and $S(\text{HI})$, we now show how to construct the schedule tables LO-*table* and HI-*table* for LO- and HI-criticality mode with switch-through property. We construct the schedule tables slot by slot, i.e., we implement preemptive scheduling with the slot as the atomic unit of execution. Additionally, construction of the scheduling tables is done concurrently for both tables, i.e., we schedule a slot i in both schedule tables (first in LO-*table* then in HI-*table*) before we proceed to the next slot $i+1$. The length of the schedule table is determined by the last deadline of the job set or the hyper-period in case of a job set derived from a periodic task set. Scheduling decisions are represented by a *search tree* which is based on iterative deepening [Kor85]. Each scheduling decision for a slot in both tables, i.e., a pair of selected jobs, is represented by an edge in the search tree. The levels of the search tree (the nodes) correspond to a scheduled slot, except for the root node which is the starting point for the first scheduling decision. Based on the history of decisions, a node represents a possible partial schedule of both tables at a given point in time.

In each node of a path, several scheduling decisions can be taken, leading to different (partial) schedules. All combinations of possible scheduling decisions form the complete search tree. Leaf nodes in a complete search tree represent complete schedules, both feasible and infeasible ones. The search for a feasible schedule is difficult due to the fact that we schedule mixed-criticality job sets with precedence constraints. Further, the table generation process requires the simultaneous construction of two schedule tables: LO-*table* and HI-*table*. Our selection of jobs in both tables uses a heuristic which is based on EDF and the criticality levels of the jobs. If a scheduling decision leads to infeasible schedule tables, we use a backtracking method, which we refer to as **swapping**, to search for another schedule table. In classic backtracking, all possible decisions in a search tree are considered which can be extremely complex. We use swapping, presented in Section 5.4, for backtracking which is based on the demand of HI-criticality jobs to reduce the complexity of backtracking.

5.3.1 Low Criticality Schedule Table

In the LO-criticality mode LO-*table*, we select a ready job of the set $S(\text{LO})$ with the earliest deadline. We introduce the concept of *leeway* $\delta(s)$ of a slot s as a heuristic function for our backtracking mechanism. The calculation of the leeway depends on the criticality level of the selected job in the current slot: for LO-criticality jobs J_i, the leeway represents the difference between the deadline of the selected job and the current time, i.e., end of current slot. For HI-criticality jobs J_i^{LO}, the leeway represents the difference between the deadline of the selected job J_i^{LO} and the current point in time **reduced** by the remaining, i.e., which have not been completely scheduled, demand of all jobs J_k^{Δ} with $k \in \{1, ..., n\}$ which have to be scheduled in HI-*table* until the deadline of J_i^{Δ}. The demand $g^{\Delta}(t)$ at time t represents the demand in the interval $[0, t]$ accumulated by all jobs J_i^{Δ}. The function $g_{sched}^{\Delta}(t)$ keeps track of the already scheduled demand of HI-criticality jobs J_i^{Δ} with deadlines until d_i^{Δ}. Equation (5.9) shows the leeway calculation based on the criticality level of the selected job. If there is no job scheduled in a slot, we define the leeway to be infinity.

$$\delta(s) = \begin{cases} d_i - (s+1) & \text{if } \chi_i = \textsc{lo} \\ \left[d_i^{\Delta} - (s+1)\right] - \left[g^{\Delta}(d_i^{\Delta}) - g^{\Delta}_{sched}(s)\right] & \text{if } \chi_i^{\textsc{lo}} = \textsc{hi} \\ \infty & \text{else} \end{cases} \tag{5.9}$$

The current slot s_c in the HI-criticality schedule table HI-*table* has not been scheduled yet, hence, $g^{\Delta}_{sched}(s_c)$ does not include the scheduled demand of HI-criticality jobs J_i^{Δ} in the current slot in HI-*table*. Due to the fact that $g^{\Delta}_{sched}(s_c)$ is only part of the equation if there is a HI-criticality job $J_i^{\textsc{lo}}$ scheduled, $g^{\Delta}_{sched}(s_c)$ will not be increased in this slot in HI-*table* because in HI-*table* there will be a job $J_i^{\textsc{lo}}$ scheduled which does not contribute to $g^{\Delta}_{sched}(s_c)$. We show the corresponding scheduling decisions for HI-*table* in Section 5.3.2. For each slot in the mode LO-*table*, we calculate the leeway. A non-negative leeway means it is possible to schedule remaining jobs J_i^{Δ} in the HI-criticality mode. Thus, we continue the scheduling process by scheduling the current slot in mode HI-*table*. If the leeway is negative, then independent of succeeding scheduling decisions, all paths in the search tree will lead to leaves representing infeasible schedules. As a consequence, we start backtracking based on our heuristic which we show in Section 5.4. Based on the heuristic function (leeway), a preceding node in the search tree is searched to apply for backtracking, i.e., we change the scheduling decision for that slot.

5.3.2 High Criticality Schedule Table

Based on the decision in LO-*table*, we select a job for HI-*table*. If the scheduled job in the current slot in schedule table LO-*table* has criticality level HI, i.e., $J_i^{\textsc{lo}}$, with $i \in \{1,..,n\} \wedge \chi_i^{\textsc{lo}} = \textsc{hi}$, then we schedule the same job $J_i^{\textsc{lo}}$ in the current slot in mode HI-*table*. If the scheduled job in the current slot in mode LO-*table* is a LO-criticality job J_i, or no job is scheduled, then we select a HI-criticality job J_k^{Δ}, with $k \in \{1,...,n\} \wedge \chi_k^{\Delta} = \textsc{hi}$, with fulfilled precedence constraints and earliest deadline. After scheduling a HI-criticality job J_k^{Δ}, we increase the amount of already scheduled demand $g^{\Delta}_{sched}(s_c)$ by one slot. After scheduling the current slot, we check whether a deadline miss of a HI-criticality job J_i^{Δ}, with $i \in \{1,...,n\} \wedge \chi_i^{\Delta} = \textsc{hi}$, occurred. In this case, the scheduling process is aborted. It is possible to extend the method by applying standard backtracking in this case to search for a solution at cost of increasing the complexity of scheduling again.

After scheduling this slot in the HI-criticality mode HI-*table*, we continue with scheduling the next slot.

5.4 Backtracking Procedure: Swapping

As shown before, it is possible that we have to change scheduling decisions which is done by applying a form of backtracking. In this section, we present our method to reduce the complexity of the backtracking mechanism. If the scheduler calculates a negative

leeway for a slot, we apply backtracking with our heuristic based on the leeway. Figure 5.4 describes the differences between classic backtracking and our applied methods. Solid arrows represent scheduling decisions and dashed arrows are backtracking steps.
In column (I), classic backtracking is applied. For instance, the fourth scheduling decision leads to a deadline miss (DL miss), then the scheduler goes back to a higher level to take another scheduling decision which is shown by the fifth and sixth arrow. If in the higher level all possible scheduling decisions are already exploited, here represented by arrows four and six, then the scheduler goes back to the next higher level in the tree – as shown by eighth arrow. The steps are repeated until a feasible schedule is found or the complete tree is explored.
In column (II), we integrate the leeway for an earlier detection that the current path will lead to a deadline miss. Here, a negative leeway is an exact indicator that independent of the next scheduling decisions a deadline miss will occur, e.g., after scheduling decision indicated by third arrow. Hence, we can already reduce the complexity just by applying the leeway for early detection of future deadline misses. Further steps follow the classic backtracking approach.
In column (III), we additionally integrate the backtracking procedure called swapping which allows for further reduction of the tree by guiding the backtracking. After the third scheduling decision, we check the slots already scheduled for a promising predecessor such that we can skip one or several backtracking steps, e.g., as shown by fourth arrow.

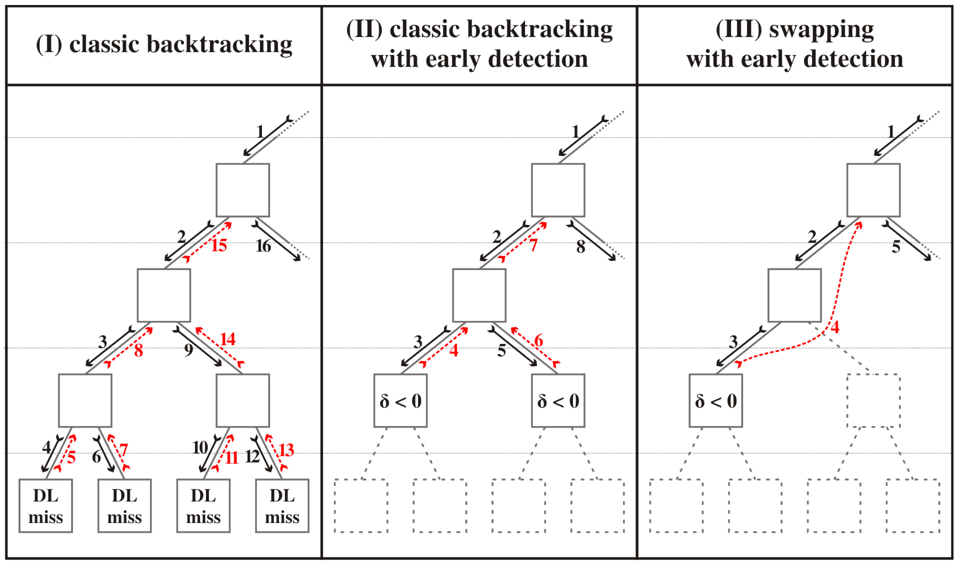


Figure 5.4: *Backtracking methods:* (I) *classic backtracking,* (II) *classic backtracking with early detection,* (III) *swapping with early detection.*

In the following, we describe our swapping procedure. Depending on the criticality of the jobs in the affected slots, there are different consequences for the swapping procedure. In Figure 5.5 column (I), scheduling decision (b) for current slot s_c leads to a negative leeway and hence, all succeeding scheduling decisions will yield infeasible schedule tables.

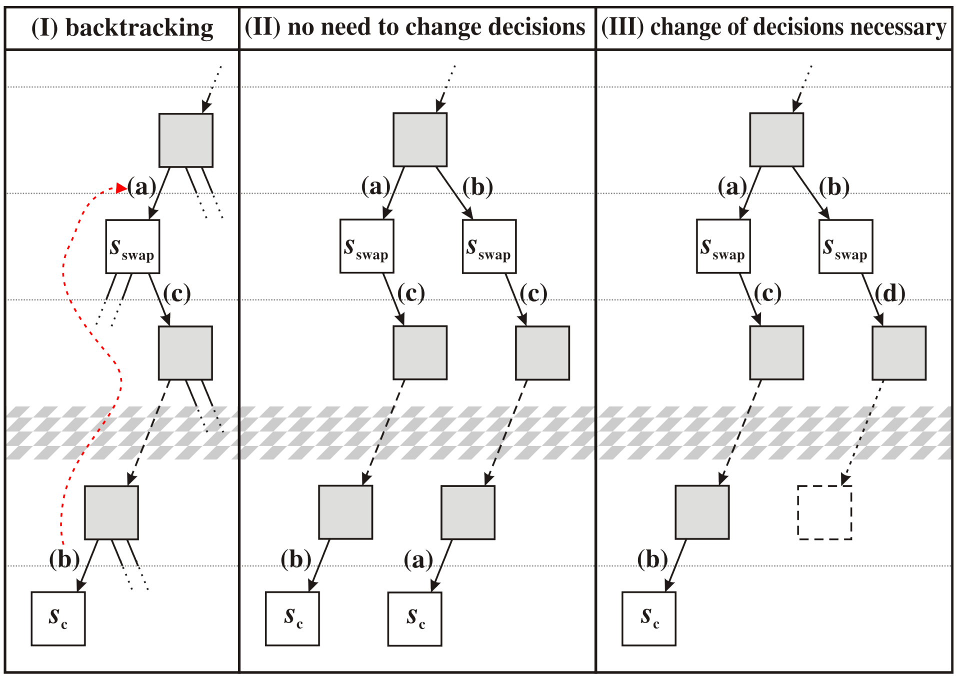

(I) backtracking
(II) no need to change decisions
(III) change of decisions necessary
(a)
(b)
(c)
(d)
s_{swap}
s_c

Figure 5.5: *Swapping in the search tree and consequences for the scheduling decisions.*

As a consequence, we may skip the search for a feasible schedule in this part of the search tree. By this early detection of paths leading to infeasible schedule tables, we save the time to check all succeeding decisions and thus, reduce the complexity.

Based on the leeway values, we look for a promising predecessor node to continue the scheduling process with a different scheduling decision for that node, as indicated by the dashed (red) arrow in Figure 5.5 column (I) from (b) to (a). We start in the current slot s_c and proceed upwards in the tree structure, checking based on the leeway of each slot whether it is a possible slot s_{swap} for which we can swap the scheduling decision. The conditions that define this swapping slot s_{swap} are: The swapping slot s_{swap} must be later than the release time of job scheduled by decision (b). Furthermore, the leeway of a candidate for the swapping slot must be greater than or equal to the difference in number of slots between the current slot and the candidate for the swapping slot. Inequalities (5.10a) and (5.10b) show these conditions. Additionally, the swapping slot has to be early enough such that the job in the current slot can meet its deadline which is represented by the condition in Inequality (5.10c). If these conditions are fulfilled, we can delay scheduling decision (a) of the swapping slot without violating the deadline. Further, the remaining demand of high criticality jobs in mode HI-*table* can be scheduled until the deadline of the job in the swapping slot.

$$\begin{cases} r_i \le s_{swap} & \text{if } \chi_i = \text{LO} \\ r_i^{\text{LO}} \le s_{swap} & \text{if } \chi_i^{\text{LO}} = \text{HI} \end{cases} \tag{5.10a}$$

$$\delta(s_{swap}) \ge s_c - s_{swap} \tag{5.10b}$$

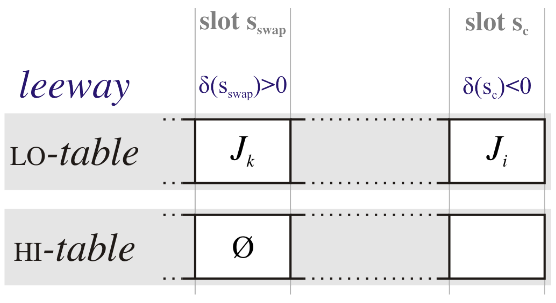
$$s_{swap} \le s_c + \delta s_c \tag{5.10c}$$

Once we found a slot which fulfills the swapping conditions, we now take scheduling decision (b) for s_{swap} and decision (a) for s_c, which can be seen in Figure 5.5 column (II). The scheduling decisions after the swapping slot, e.g., decision (c), remain unchanged. In other words, we only swap the decisions of the two slots s_c and s_{swap}. After swapping the decisions of the two slots, we have to recalculate the leeway values of these slots.

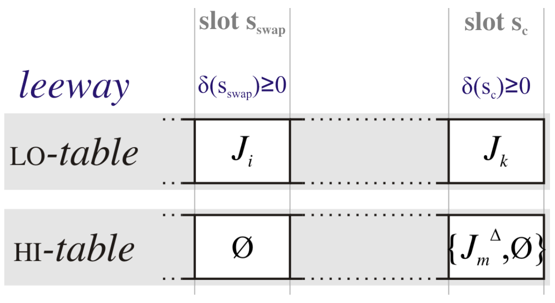
By swapping scheduling decisions (a) and (b), it is possible that fulfilled precedence constraints are not fulfilled anymore and/or scheduled demand of HI-criticality jobs J_i^{Δ} is changed. In this case, we cannot continue with scheduling decision (c) after s_{swap} and we continue the scheduling process after s_{swap} with a possibly different decision (d) for the next slot. This is shown in Figure 5.5 column (III). As a result, the current slot s_c is set to the swapping slot s_{swap} and we continue the scheduling process from that slot.

Depending on the scheduled jobs in the current slot and the swapping slot, there are six different cases which have to be considered when making swapping decisions. In the following, we present these cases and the consequences for the scheduling process. We refer to slots in mode LO-*table* by s^{LO} and slots in mode HI-*table* by s^{HI}. We group the different cases into the two categories already shown in Figure 5.5 in columns (II) and (III): for category NC (**N**o **C**hange), there is no need to change scheduling decisions between slot s_c and s_{swap}, whereas in category DC (**D**ecisions **C**hanged), we set $s_c := s_{swap}$ and continue scheduling from this point with possibly different scheduling decisions.

Case 1. In both slots s_c^{LO} and s_{swap}^{LO}, a LO-criticality job is scheduled. In s_{swap}^{HI}, no job is scheduled. The schedule which triggers swapping is shown in Figure 5.6(a). We swap slots in LO-criticality mode LO-*table* and update the leeway of s_c and s_{swap}. The swapping slot in the HI-criticality mode remains unchanged because if there was no HI-criticality job J_m^{Δ} with $m \in \{1, ..., n\}$ ready to be scheduled, swapping of two LO-criticality jobs does not influence this. In a last step, we schedule the current slot in the HI-criticality mode. Figure 5.6(b) depicts the situation after swapping the decision in slots s_c and s_{swap}. In this case, swapping does not change any precedence constraints and this refers to category NC.



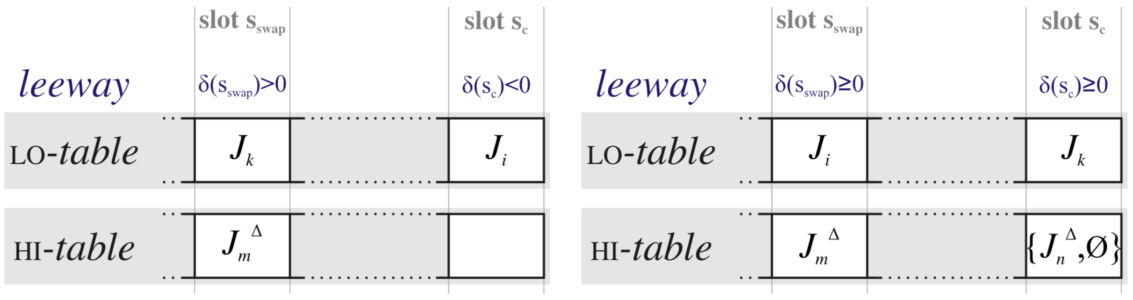
(a) Slots before swapping.



(b) Slots after swapping.

Figure 5.6: *Swapping case 1.*

Case 2. In both slots s_c^{LO} and s_{swap}^{LO}, a LO-criticality job is scheduled. In s_{swap}^{HI}, a HI-criticality job J_m^{Δ} is scheduled. Figure 5.7(a) shows this situation. We swap slots in LO-criticality mode LO-*table* and update the leeway of s_c and s_{swap}. Swapping two LO-criticality jobs does not change precedence constraints, and hence, the scheduled job in s^{HI}_{swap} remains unchanged. Finally, we schedule the current slot in the HI-criticality mode. The schedule after swapping is shown in Figure 5.7(b). Here, swapping refers to category NC.



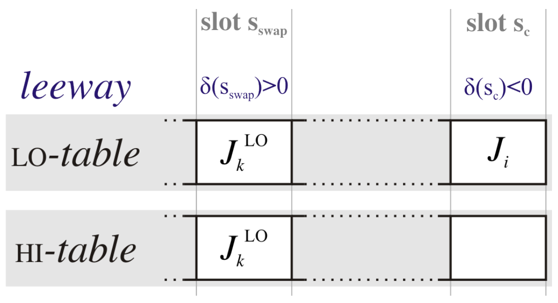
(a) Slots before swapping. (b) Slots after swapping.

Figure 5.7: *Swapping case 2.*

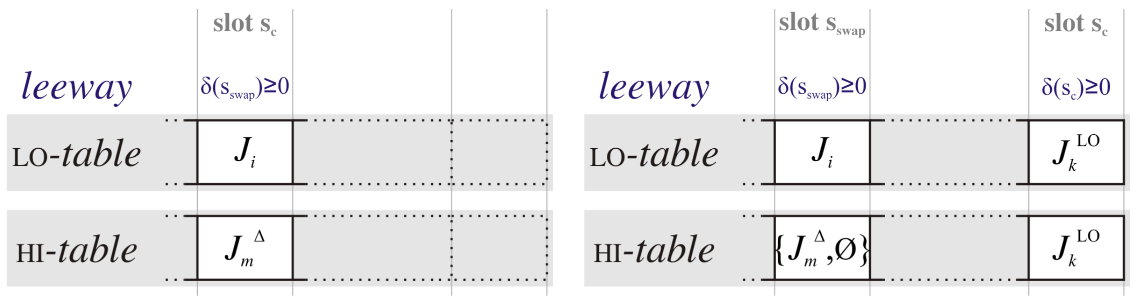
Case 3. In slot s^{LO}_c, a LO-criticality job J_i and in slot s^{LO}_{swap}, a HI-criticality job J^{LO}_k are scheduled. In s^{HI}_{swap}, the same HI-criticality job J^{LO}_k as in s^{LO}_{swap} is scheduled which is shown in Figure 5.8(a). We swap slots in LO-criticality mode LO-*table* and update the leeway of s_c and s_{swap}. Now we must check whether fulfilled precedence constraints have been changed by swapping, i.e., whether an already fulfilled precedence constraint between s_{swap} and s_c is now not fulfilled anymore or is fulfilled at a later slot.
If fulfilled precedence constraints have been changed, then we re-schedule slot s^{HI}_{swap} and set the swapping slot as current slot and continue the scheduling process at slot s_c. Figure 5.8(b) shows the schedule after this actions. This case refers to category DC.
If fulfilled precedence constraints have not been changed, then we swap s^{HI}_{swap} and s^{HI}_c (unscheduled yet) and re-schedule s^{HI}_{swap} based on the fulfilled precedence constraints at that time, as described in Section 5.3.2. Figure 5.8(c) depicts the schedule after swapping and the actions result in category NC.

Case 4. Figure 5.9(a) illustrates the situation which triggers the swapping: In slot s^{LO}_c, a HI-criticality job J^{LO}_i and in slot s^{LO}_{swap}, a LO-criticality job J_k are scheduled. In s^{HI}_{swap}, no job is scheduled. We swap slots in LO-criticality mode LO-*table* and update the leeway of s_c and s_{swap}. In slot s^{HI}_{swap}, we schedule the same job J^{LO}_i as in s^{LO}_{swap} (after swapping). As a consequence, we update $g^{\Delta}_{sched}(s)$ and leeway $\delta(s)$ for $s \in \{s_{swap}, ..., s_c\}$. Eventually, we schedule the current slot in the HI-criticality mode as depicted in Figure 5.9(b). This case refers to category NC.

Case 5. In slot s^{LO}_c, a HI-criticality job J^{LO}_i and in slot s^{LO}_{swap}, a LO-criticality job J_k are scheduled. In s^{HI}_{swap}, a HI-criticality job J^{Δ}_m is scheduled as shown in Figure 5.10(a). First, we check whether scheduling J^{Δ}_m in s^{HI}_c leads to a deadline miss.

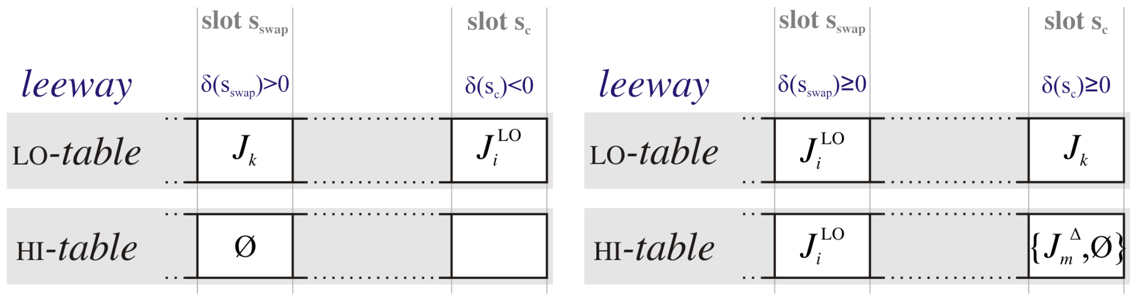


(a) Slots before swapping.



(b) Slots after swapping (changed fulfilled precedence constraints). (c) Slots after swapping (unchanged fulfilled precedence constraints).

Figure 5.8: *Swapping case 3.*

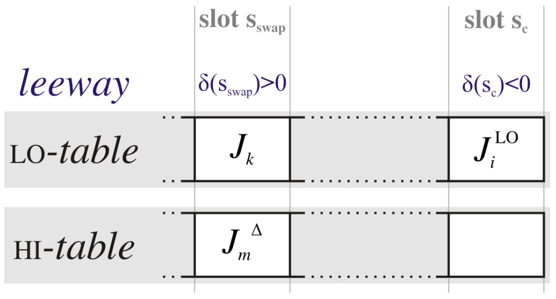


(a) Slots before swapping. (b) Slots after swapping.

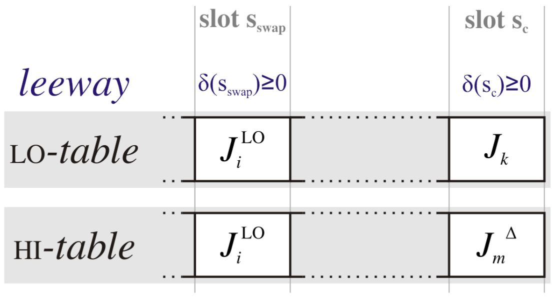
Figure 5.9: *Swapping case 4.*

If yes, then we have to search for another swapping slot.
If this does not lead to a deadline miss, then we swap slots in LO-criticality mode LO-*table*. Then, we swap s^{HI}_{swap} and s^{HI}_{c} (unscheduled yet) and schedule in s^{HI}_{swap} the same job J^{LO}_i as in slot s^{LO}_{swap} (after swapping in LO-*table*). As a consequence, we update $g^{\Delta}_{sched}(s)$ and leeway $\delta(s)$ for $s \in \{s_{swap}, .., s_c\}$. Figure 5.10(b) depicts the schedule after swapping. The category for this swapping case is NC.

Case 6. As Figure 5.11(a) shows, in both slots s^{LO}_c and s^{LO}_{swap}, HI-criticality jobs J^{LO}_i and J^{LO}_k are scheduled. Further, in s^{HI}_{swap}, the same HI-criticality job J^{LO}_k is scheduled as in slot s^{LO}_{swap}. We swap slots in LO-criticality modes and update the leeway of s_c and



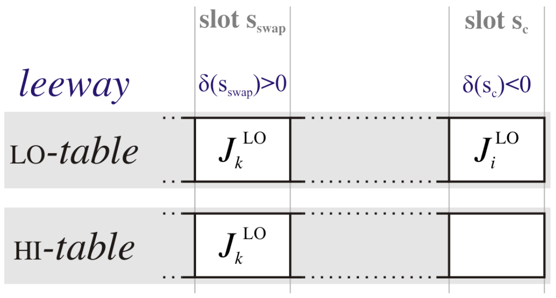
(a) Slots before swapping.



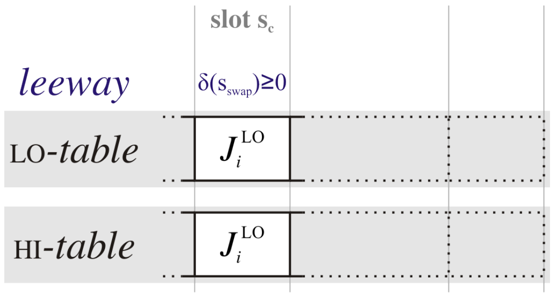
(b) Slots after swapping.

Figure 5.10: *Swapping case 5.*

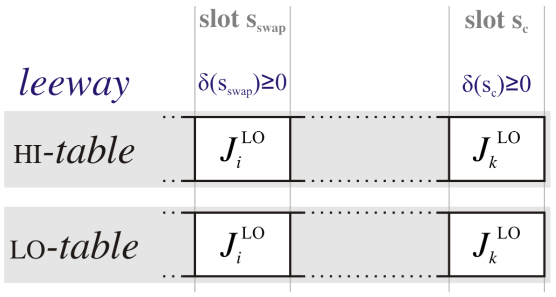
s_{swap}. Now we must check whether fulfilled precedence constraints have been changed by swapping.
If fulfilled precedence constraints have been changed, then we schedule J_i^{LO} in slot s_{swap}^{HI}, set the swapping slot as current slot, and continue the scheduling process. We show this in Figure 5.11(b). This refers to category `DC`.
If fulfilled precedence constraints have not been changed, then we schedule J_i^{LO} in slot s_{swap}^{HI} as presented in Figure 5.11(c). This situation is categorized into category `NC`.



(a) Slots before swapping.



(b) Slots after swapping (changed fulfilled precedence constraints).



(c) Slots after swapping (unchanged fulfilled precedence constraints).

Figure 5.11: *Swapping case 6.*

In summary, the leeway allows for an early detection of scheduling decisions that result in infeasible schedules. Further, we can reduce the complexity of backtracking by the application of our swapping procedure.

5.5 Example of Schedule Table Construction

In the following, we present a small example showing the construction of the TT schedule tables. Further, the example shows our backtracking method *swapping*. In Table 5.3, we show the job set used in this example. The job set consists of four jobs: two LO-criticality jobs and two HI-criticality jobs.

jobs	criticality level	LO-criticality WCET	HI-criticality WCET	release time	(absolute) deadline
J_1	LO	1	1	0	4
J_2	LO	2	2	0	6
J_3	HI	1	4	0	8
J_4	HI	1	4	0	8

Table 5.3: *Example job set for the schedule table construction.*

In a first step, we split the jobs as shown in Section 5.2. We apply Equations (5.1) - (5.7) to determine the parameters for the split jobs. LO-criticality jobs J_1 and J_2 are not split and we assign the LO-criticality WCET as new WCET for the job after the splitting procedure. The other parameters remain unchanged. HI-criticality jobs J_3 and J_4 are split into J_3^{LO} and J_3^{Δ}, and into J_4^{LO} and J_4^{Δ}, respectively. Table 5.4 depicts the split jobs.

jobs	criticality level	WCET	release time	(absolute) deadline
J_1	LO	1	0	4
J_2	LO	2	0	6
J_3^{LO}	HI	1	0	5
J_3^{Δ}	HI	3	1	8
J_4^{LO}	HI	1	0	5
J_4^{Δ}	HI	3	1	8

Table 5.4: *Job set after separating the demand of designers and additional demand introduced by CAs. This job set is used to construct the schedule tables.*

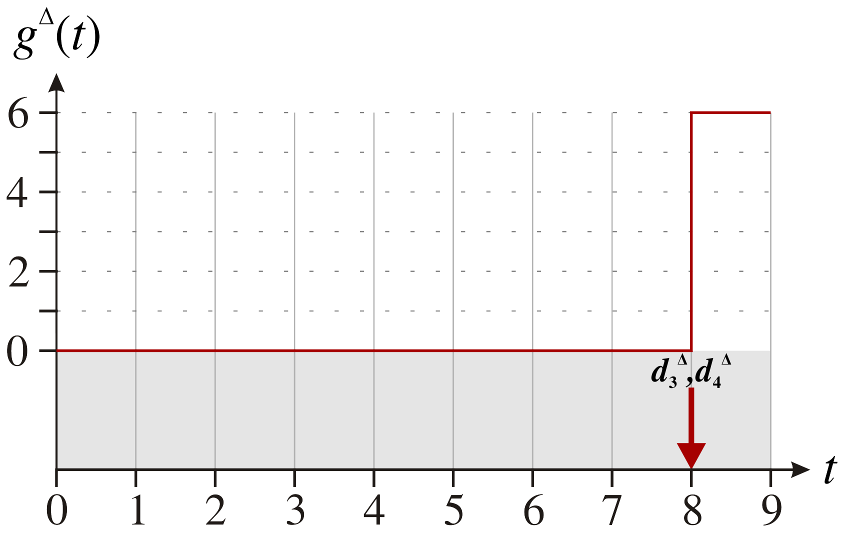


Figure 5.12: *Demand of* HI*-criticality jobs* J_i^Δ.

For the leeway calculations, we need the demand of HI-criticality jobs J_i^Δ which is depicted in Figure 5.12.

After we have split the jobs and calculated the demand, we can now start constructing the schedule tables LO-*table* and HI-*table*. We start in LO-*table* with slot 0. We select a job with earliest deadline, which is in this case J_1. We calculate the leeway for this slot according to Equation (5.9):

$$\begin{aligned}\delta(0) &= d_1 - (0+1)\\ &= 4 - (0+1)\\ &= 3\end{aligned}$$

Due to the fact, that we just scheduled a LO-criticality job in LO-*table*, we now schedule a job J_i^Δ with earliest deadline and fulfilled precedence constraints in HI-*table*. In this case, there is no job fulfilling these constraints and thus, we schedule an idle slot. The combination of these two scheduling decisions is represented by one edge in the scheduling tree. The succeeding node in the tree represents the resulting partial schedule. The partial schedule after scheduling slot 0 and the search tree are shown in Figure 5.13.

In the next slot, we start again in LO-*table* selecting a job by EDF. As a result, we select J_3^{LO} and calculate the leeway:

$$\begin{aligned}\delta(1) &= \left(d_3^\Delta - (1+1)\right) - \left(g^\Delta(d_3^\Delta) - g^\Delta_{sched}(1)\right)\\ &= (8 - (1+1)) - (6-0)\\ &= 0\end{aligned}$$

We scheduled a HI-criticality job and as a consequence, we schedule the same job J_3^{LO} in HI-*table* which is necessary to obtain the switch-through property. The combination of these two scheduling decisions is represented by a new edge in the scheduling tree.

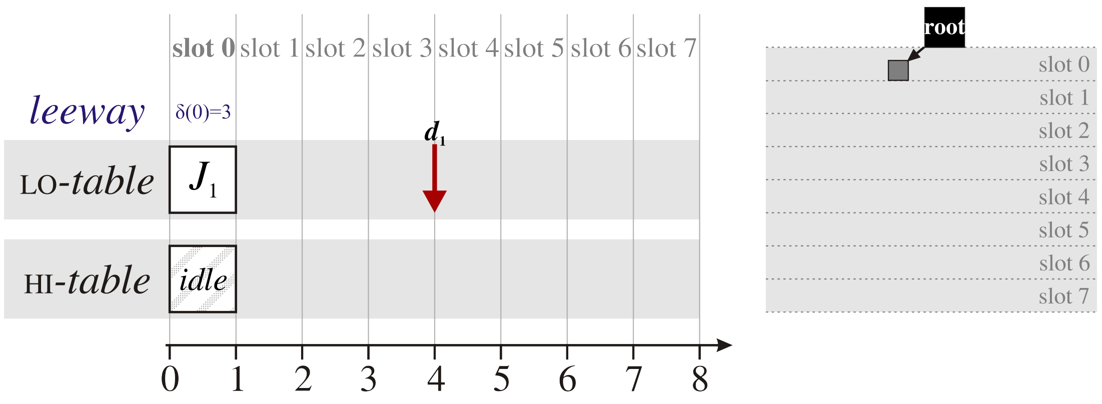


Figure 5.13: *Partial schedule after scheduling slot 0.*

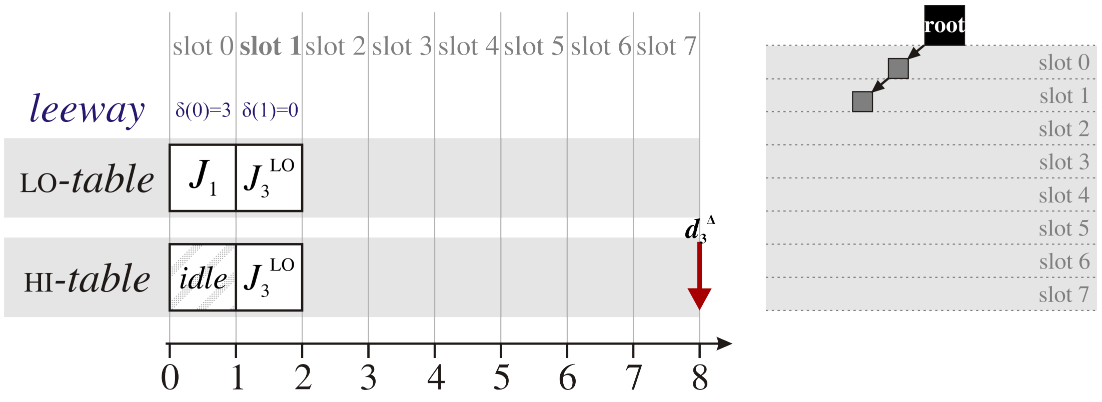


Figure 5.14: *Partial schedule after scheduling slot 1.*

The succeeding node in the tree represents the resulting partial schedule including slot 1. The partial schedule after scheduling slot 1 and the search tree are shown in Figure 5.14.

We continue with slot 2 in LO-*table*. J_4^{LO} is the job with the earliest deadline which is ready to execute. The leeway is calculated as follows:

$$\begin{aligned}\delta(2) &= \left(d_4^{\Delta} - (2+1)\right) - \left(g^{\Delta}(d_4^{\Delta}) - g_{sched}^{\Delta}(2)\right)\\ &= (8 - (2+1)) - (6 - 0)\\ &= -1\end{aligned}$$

The consequence of negative leeway is that we apply our backtracking method swapping. We start with the previous slot to search for a possible slot to swap scheduling decisions with the current slot, which is in this case slot 1. In slot 0, the leeway is 3 and this fulfills the conditions presented in Equations (5.10a) - (5.10c). The situation is depicted in Figure 5.15 and refers to swapping case 4 which is shown in Section 5.4.

We swap slot 2 with slot 0 in both schedules LO-*table* and HI-*table*. Figure 5.16 shows which slots are swapped in the schedules and the decisions in the search tree which are swapped.

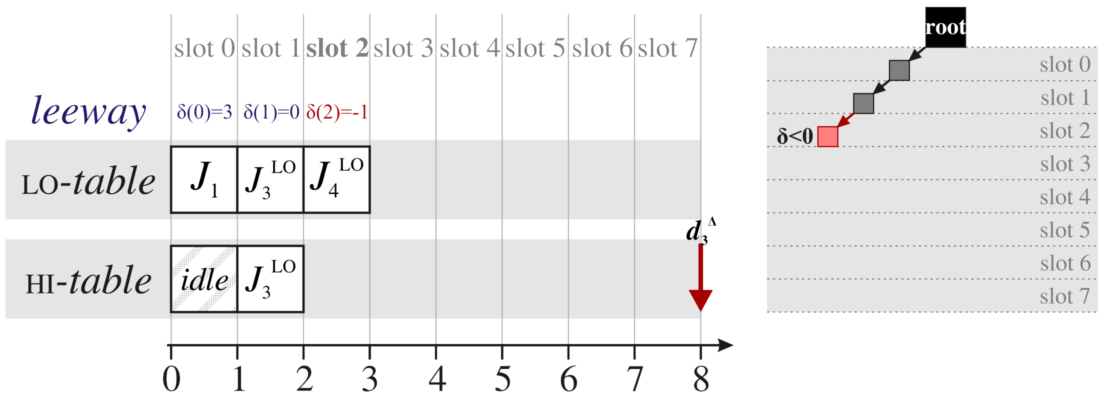


Figure 5.15: *Partial schedule before swapping slot 2.*

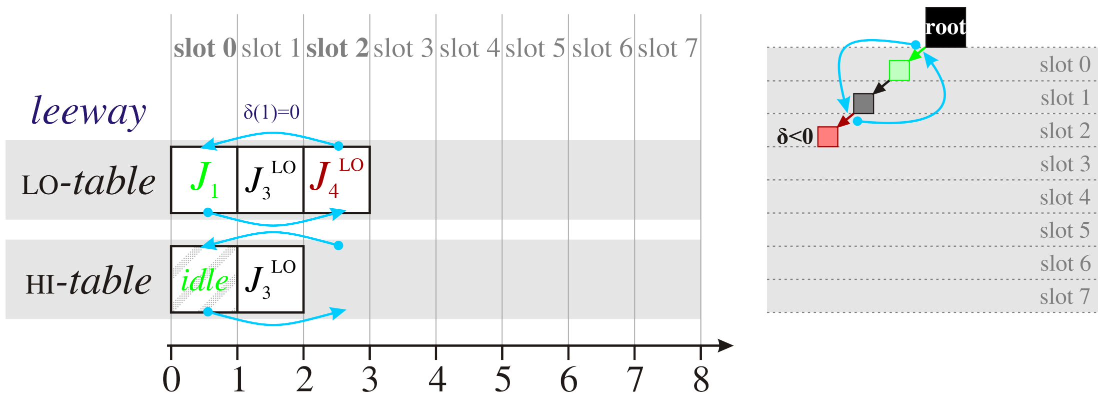


Figure 5.16: *Swapping slot 2 with slot 0.*

In the search tree, swapping the scheduling decisions in these slots corresponds to a new branch in the tree. Now, we re-calculate the leeway values for both slots 0 and 2. Both leeway values are now non-negative and we can continue the scheduling process.

After swapping, we finish scheduling of the slot which we started with scheduling J_4^{LO} in LO-*table*. As a consequence, we schedule J_4^{LO} in slot 0 in HI-*table*. The re-calculated leeway values, the additional path in the search tree, and the completed schedule decision of slot 0 is shown in Figure 5.17.

Due to swapping slots whereas one slot is an idle slot, we now can check whether we can re-schedule this slot. In slot 2 in HI-*table* we can now schedule a job J_i^Δ with earliest deadline and fulfilled precedence constraints. This results in scheduling J_3^Δ in this slot. Figure 5.18 shows the partial schedule after swapping and completing the scheduling decisions in slot 2.

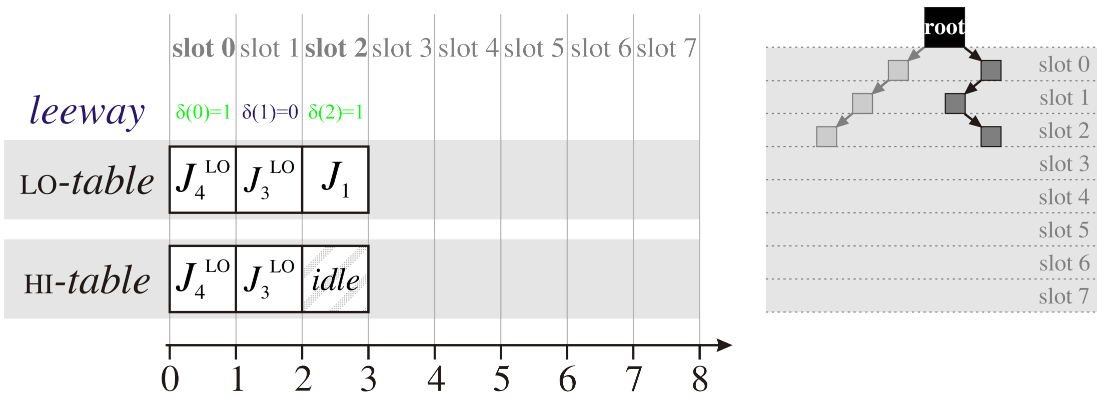


Figure 5.17: *Partial schedule after swapping and completing scheduling in slot 0.*

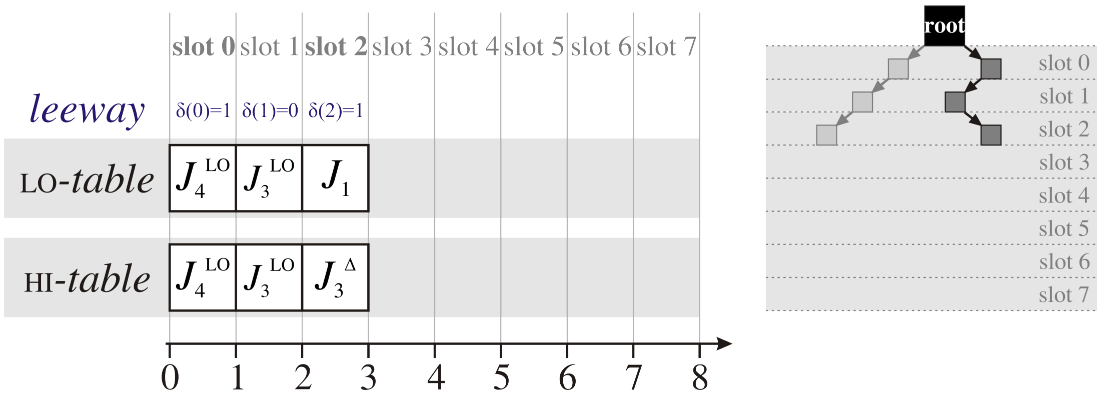


Figure 5.18: *Partial schedule after completing scheduling in slot 2.*

In the next slot 3, we schedule J_2 which has the earliest deadline of ready jobs in LO-*table*. We calculate the leeway as follows:

$$\begin{aligned}\delta(3) &= d_2 - (3+1)\\ &= 6 - (3+1)\\ &= 2\end{aligned}$$

As a consequence of the LO-criticality job in LO-*table*, we schedule a job J_i^{Δ} with earliest deadline in HI-*table*, here J_3^{Δ}. In slot 4, we ready jobs result in repeating the same scheduling decisions as in slot 3, whereas the leeway of slot 4 results in:

$$\begin{aligned}\delta(4) &= d_2 - (4+1)\\ &= 6 - (4+1)\\ &= 1\end{aligned}$$

The partial schedule after scheduling slot 3 and 4 is shown in Figure 5.19.

We scheduled all jobs allocated to LO-*table* and hence, we schedule idle slot in LO-*table* until we also scheduled all jobs in HI-*table*. The leeway for these slots is set to infinity. In HI-*table*, we schedule J_4^{Δ} in slots 5-7 such that it completes its execution. After doing

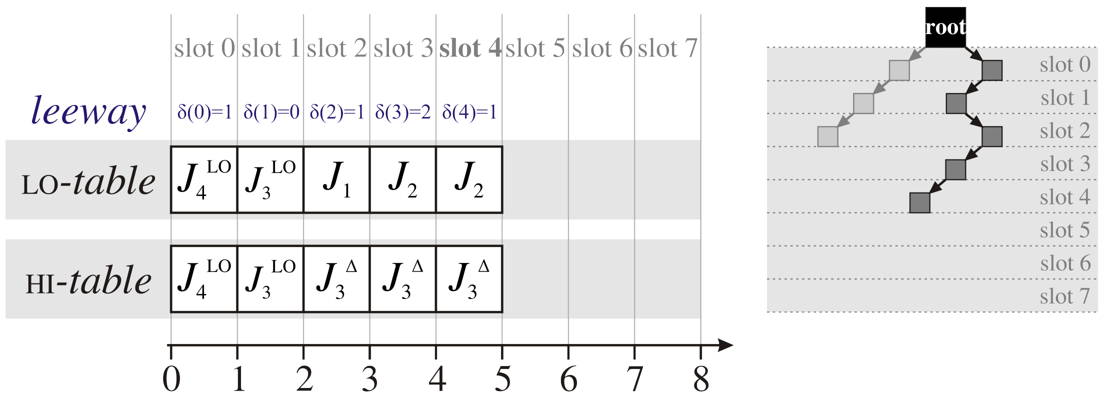


Figure 5.19: *Partial schedule after scheduling slot 3 and 4.*

this, all jobs in both schedule tables are scheduled. The resulting (complete) schedule tables are shown in Figure 5.20. The branch in the search tree which results in the feasible complete schedule is also shown in Figure 5.20.

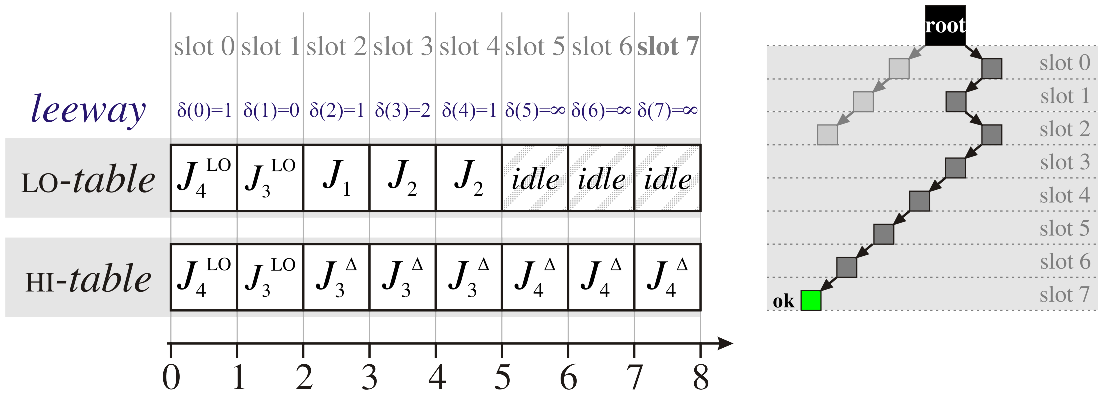


Figure 5.20: *Complete schedule after completing scheduling in slot 7.*

5.6 Evaluation

In this section, we evaluate the efficiency and effectiveness of our tree-based method of TT schedule table construction presented in Sections 5.2 - 5.4. We implemented the scheduling algorithm including the leeway to determine paths in the search that lead to infeasible schedules for early application of backtracking. To reduce the complexity of backtracking, we implemented our backtracking method called swapping. For evaluation, we compare our method with the fixed priority scheduling (FPS) approach presented by Baruah and Fohler in [BF11].

In the following, we briefly review Baruah's and Fohler's FPS algorithm to construct TT schedule tables with mode changes. After that, we describe the setup of our experiments and evaluate our scheduling algorithm. Further, we show the results of the empirical comparison in the ability of the two approaches to schedule example job sets.

We compare the algorithms regarding the influence of the utilization of LO-criticality jobs. Further, we analyze the results with respect to the introduced pessimism by the CAs, i.e., by extension of the WCETs, and the ratio of HI-criticality jobs in the job set. Finally, we present and evaluate the runtimes of both algorithms.

5.6.1 Fixed Priority Schedule Table Construction

Baruah's and Fohler's algorithm to construct two schedule tables is based on a priority ordering. The algorithm is sufficient but not necessary, i.e., it generates schedule tables not for all possible mixed-criticality job sets, but if the generation is successful, then the table is correct. The algorithm works in the following steps: 1) It assigns priorities and if this is not successful, the method cannot schedule the job set. 2) The schedule table LO-*table* is created based on this priority ordering using $C_i(\text{LO})$ values for all jobs. 3) Based on same priority ordering and $C_i(\text{HI})$ values for all jobs, whereas $C_i(\text{LO}) = C_i(\text{HI})$ for LO-criticality jobs, the algorithm constructs the HI-criticality table HI-*table*.

Assigning job priorities. Baruah's and Fohler's approach is based on an optimal recursive priority ordering method presented by Audsley [Aud91, Aud93] to assign priorities. This approach looks for a job which can be assigned the lowest priority and if it finds one, it continues with the remaining set of jobs. A job J_i may be assigned the lowest priority if criticality level of J_i is LO and there are at least $C_i(\text{LO})$ time units between release time and deadline of this job if all other jobs have a higher priority than J_i. If job J_i has criticality level HI, then at least $C_i(\text{HI})$ must be available in its execution window under the assumptions that all other jobs have a higher priority and execute for their $C_i(\text{HI})$. Remember that $C_i(\text{LO}) = C_i(\text{HI})$ for LO-criticality jobs. This procedure is repeated until the priority ordering includes all jobs or in an iteration step, a job which can have lowest priority does not exist. In the latter case, the priority assignment fails and the job set cannot be scheduled.

Constructing the Schedules. Based on the priority ordering determined before, we now show how to construct the TT schedule tables. The algorithm simulates a fixed priority scheduler for both LO-*table* and HI-*table*. In LO-*table*, all jobs are scheduled assuming jobs execute exactly for $C_i(\text{LO})$. In HI-*table*, all jobs are scheduled assuming jobs execute exactly for $C_i(\text{HI})$. A consequence of including all jobs into the HI-criticality schedule table is the possible over-provisioning as shown in the following Example.

Example. To illustrate the over-provisioning, which is a property independent of the priority assignment, we use a simplified priority assignment instead of the more complex Audsley-approach. We assign priorities $p_i \in \mathbb{N}\backslash\{0\}$ with 1:= highest priority. HI-criticality jobs have higher priority than LO-criticality jobs, and we use as priority ordering within the criticality levels the index. Table 5.5 shows the example job set.

Figure 5.21 shows the resulting schedule tables whereas jobs are scheduled for $C_i(\text{LO})$ time units in LO-*table* and for $C_i(\text{HI})$ time units in HI-*table*. Only HI-criticality jobs can trigger a mode change from LO-*table* to HI-*table*, i.e., in this example jobs J_1 and J_3. If J_1 exceeds its $C_1(\text{LO})$ at t_1, i.e., it does not signal completion, then the mode is changed to HI-*table* and both J_1 and J_3 can execute for their HI-criticality WCETs $C_1(\text{HI})$ and

jobs	criticality level	LO-criticality WCET	HI-criticality WCET	release time	deadline	priority
J_1	HI	1	2	0	6	1
J_2	LO	1	1	0	6	3
J_3	HI	1	2	2	6	2

Table 5.5: *Example job set for the schedule table construction.*

C_3(HI), respectively. However, if J_3 exceeds its C_3(LO) at t_2, then the mode is changed to HI-*table* and J_3 can execute for 3 time units in total which is more than C_3(HI). ◊

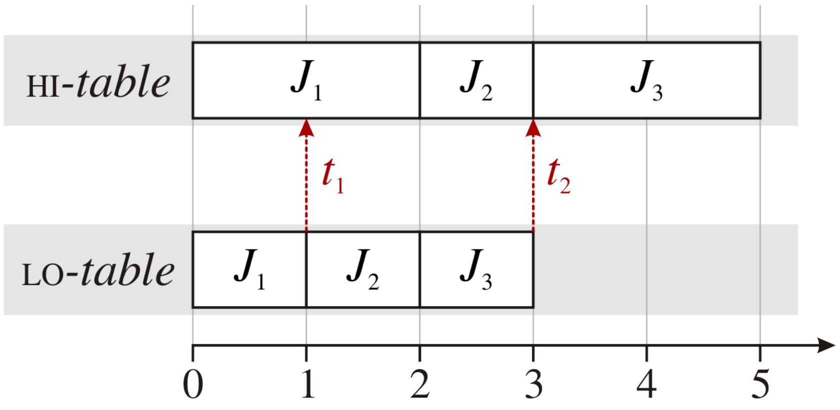


Figure 5.21: *Example: schedule tables constructed by simulating FPS [BF11].*

As a result, independent of the time instant when a mode change is triggered, all HI-criticality jobs can execute at least for C_i(HI) time units, which results in a correct schedule. However, there is the major disadvantage of over-provisioning which decreases the effective utilization.

5.6.2 Experiment Description

We generated job sets with an uniformly distributed utilization based on UUniFast algorithm [BB05]. The algorithm generates periodic tasks which are then unrolled into set of jobs. Further, jobs are randomly assigned a criticality level LO or HI according to ratio of HI-criticality jobs (HI-*ratio*). Synthetic workloads are used to show the correctness and a comparison of the methods. All parameters are rounded to full slot size values which causes errors. We only consider job sets with an utilization error up to 3%. Based on the assumption that the system is designed to meet designers' worst case scenario, job sets with LO-criticality parameters are feasible and then the pessimism of CAs' is added to WCET values.

We varied the following parameters for the evaluation: impact of CA's pessimistic assumptions, LO-criticality utilization and ratio of HI-criticality jobs; which is in line with other mixed-criticality work in the field, e.g., [BBD11]. For each combination of input parameters, we generated 1,000 random job sets.

5.6.3 Impact of Certification Authorities' Pessimistic Assumptions on Schedulability

In the following, we evaluate the impact of CA's pessimistic assumptions. The pessimism of CAs is expressed by the HI-criticality WCET which is $C_i(\text{HI}) \geq C_i(\text{LO})$. We generated job sets with the same input parameters for different degrees of pessimism, i.e., how much larger is $C_i(\text{HI})$ than $C_i(\text{LO})$. Based on $C_i(\text{LO})$, we randomly determine a value for $C_i(\text{HI})$ in the range of $[C_i(\text{LO}), hsf \cdot C_i(\text{LO})]$, whereas the high scale factor hsf represents the degree of pessimism introduced by CAs. The evaluation considers the following high scale factor values: $hsf = \{2, 3, 5\}$. We present job sets for LO-criticality utilizations $\sum_{i=1}^{n} U_i(\text{LO}) = \{10\%, 20\%, ..., 80\%\}$, i.e., workload based on $C_i(\text{LO})$. The ratio of HI-criticality jobs (HI-*ratio*) is set to 25%. Later, we explain the reason for this HI-*ratio* choice. We compared the scheduler presented in [BF11], in the following referred to as FPS, with the approach presented in this chapter, in the following referred to as SWAP. The results compare the number of job sets for which the given schedulers can create a correct schedule table (in percent based on a test set of 1,000 job sets).

Table 5.6 presents the results of the comparison of the success ratio for both FPS and SWAP. Both algorithms show a decrease in the success ratio for an increasing LO-criticality utilization. Additionally, the success ratio also decreases for increasing hsf. The reason for the decreasing success ratio of both algorithms is that increasing the LO-criticality utilization and increasing the high scale factor hsf results in a higher HI-criticality utilizations. FPS, which includes all jobs in HI-*table*, is more affected by these increases than SWAP.

		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$															
success		10%		20%		30%		40%		50%		60%		70%		80%	
ratio [%]		SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS
hsf	2	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.6	98.5	93.7	76.7	52.9	44.0	4.5	16.2	0.0
	3	100.0	100.0	100.0	100.0	100.0	99.4	89.8	87.3	52.9	37.1	25.2	5.4	10.0	0.4	2.9	0.0
	5	100.0	100.0	100.0	100.0	71.7	71.3	30.4	9.4	10.8	0.6	*	*	*	*	*	*

**** no job sets generated for this case***

Table 5.6: *Results: success ratio for* HI-*ratio=25%.*

In the following, we explain the results in more detail. Assume that "25% of jobs are HI-critical" corresponds to "25% of LO-criticality utilization is caused by HI-criticality jobs". We now calculate the theoretical utilization of all jobs in the HI-criticality case, i.e., the maximum utilization which can be obtained for the input parameters based on $C_i(\text{HI}) \forall i \in \{1, ..., n\}$. Consider a LO-criticality utilization of $\sum_{i=1}^{n} U_i(\text{LO}) = 60\%$. HI-*ratio* of 25% means that HI-criticality jobs have a LO-criticality utilization (utilization based on $C_i(\text{LO})$) of 15%. Further, the remaining LO-criticality jobs cause the remaining 45% utilzation. For a high scale factor of $hsf = 2$, the utilization based on $C_i(\text{HI})$

doubles for HI-criticality jobs, such that their HI-criticality utilization is 30%. Adding the unchanged 45% utilization of the LO-criticality jobs results in a total theoretical HI-criticality utilization of 75%. This theoretical HI-criticality utilization of 75% is also obtained if $hsf = 3$ for a LO-criticality utilization of $\sum_{i=1}^{n} U_i(\text{LO}) = 50\%$. Furthermore, if $hsf = 5$ the theoretical HI-criticality utilization can be 75% for a LO-criticality utilization of $\sum_{i=1}^{n} U_i(\text{LO}) = 37.5\%$.

The result of the theoretical observation of high utilizations, for instance, the presented 75%-case before, in HI-*table* can be confirmed by the drastically dropping success ratios of FPS at $\sum_{i=1}^{n} U_i(\text{LO}) = 60\%$ for $hsf = 2$, at 50% if $hsf = 3$, and at 40% if $hsf = 5$. The problem of the FPS approach is that it includes all jobs in HI-*table* and the high utilizations make scheduling difficult. A similar drop of the success ratios can be observed for SWAP at $\sum_{i=1}^{n} U_i(\text{LO}) = 70\%$ if $hsf = 2$, at 60% if $hsf = 3$, and at 40% if $hsf = 5$. The theoretical HI-criticality utilizations are above 80% in these cases. As a result, SWAP is less affected by an increasing HI-criticality utilization. We also investigated HI-*ratio*s over 25% which results in a drop-off point shift to lower LO-criticality utilizations and lower hsf. Hence, we select HI-*ratio*=25% to show the behavior of the algorithm because the differences are visible for all three high scale factors. For higher HI-*ratio*s, high utilization job sets are hardly schedulable and the success ratios are almost 0%.

Figure 5.22 depicts the success ratios for FPS and SWAP as already shown in Table 5.6. The graph confirms the aforementioned drop of the success ratio. The curves for $hsf = 2$ ($\bigcirc$), $hsf = 3$ ($\square$), and $hsf = 5$ ($\triangle$) show a similar trend: for a specific LO-criticality utilization, the success ratio drastically drops. Further, we see that a higher hsf-value shifts this drop-off point to lower LO-criticality utilizations.

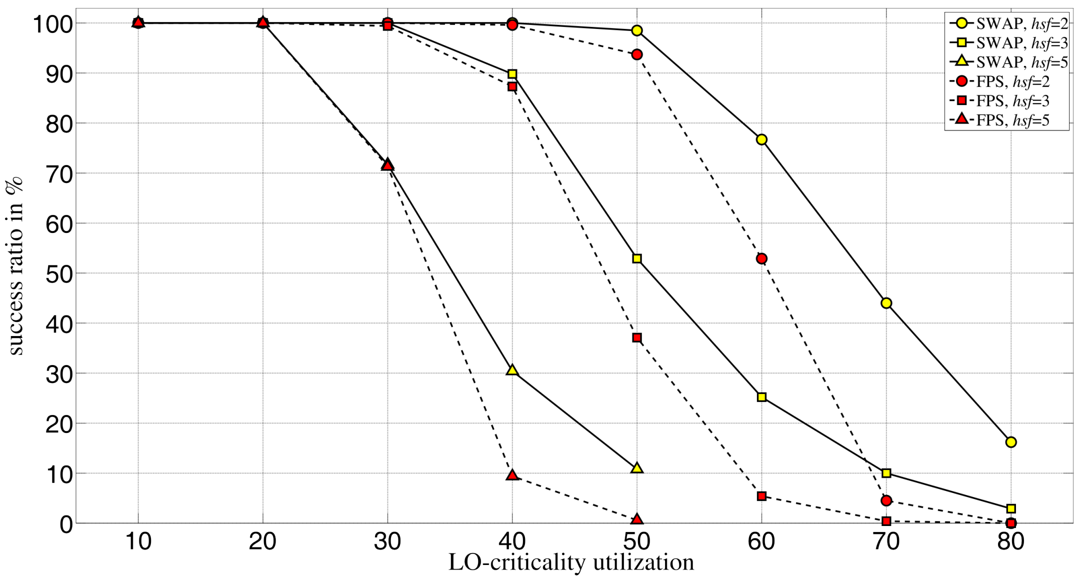


Figure 5.22: *Success ratio for varying high scale factor hsf.*

The assumption above is only an approximation to the actual HI-criticality utilizations because the job set generator does not ensure that "25% of jobs are HI-critical" corresponds to “25% of LO-criticality utilization is caused by HI-criticality jobs”. Actually , HI-criticality utilizations are mostly even higher. Based on this assumption, the utilization of HI-criticality jobs can exceed 100% for $hsf = 5$ and $\sum_{i=1}^{n} U_i(\text{LO}) \geq 60\%$ and HI-*ratio* $= 25\%$. The job set generator limits the utilization to 100%. As a consequence, if the HI-criticality utilization would exceed 100%, this correspond to job sets with lower hsf, results are not reliable and not shown (* in Table 5.6). Based on the observations above for different hsf, we present a detailed analysis of the schedulers in the following by investigating the success ratios and runtimes.

5.6.4 Schedulability Analysis

In the following, we investigate the success ratios and runtimes for FPS and SWAP for varying LO-criticality utilizations and varying HI-*ratio*. For all further experiments, hsf is set to 3 . We group experiments into three utilization categories: low (10%, 20%, 30%), medium (40%, 50%, 60%) and high (70%, 80%, 90%) LO-criticality utilizations. In each category, we used the same input parameters for the job set generator to generate the target LO-criticality utilizations $\sum_{i=1}^{n} U_i(\text{LO})$.

In Table 5.7, we present the success ratios for the corner cases HI-*ratio*=0% and HI-*ratio*=100%, i.e., for a standard (non-mixed-criticality) scheduling problem. For HI-*ratio*=0%, both schedulers can always schedule the job sets. In this case, SWAP works as a pure EDF scheduler and FPS uses an optimal priority assignment.On the contrary for HI-*ratio*=100%, FPS shows much better results in each utilization category than SWAP. Remember that each category has different input parameter for job set generation and thus, results are not comparable to other categories. For low LO-criticality utilizations, the HI-criticality utilization is still not too high and as a result, the success ratios are 100%. Whereas for LO-criticality utilizations above 30%, HI-criticality utilizations increase drastically to values above 95% and the success ratios drop rapidly. SWAP exploits slots with LO-criticality jobs and idle time to schedule additionally needed execution time in HI-criticality case, i.e., execution time of jobs J_i^{Δ}. For HI-*ratio*=100%, there are no slots with LO-criticality jobs and high HI-criticality utilizations reduce the amount of idle time slots. As a result, SWAP suffers more from the high utilizations than FPS. These corner cases do not represent mixed-criticality job sets because all jobs have the same criticality level, hence, this refers to a standard scheduling problem. As a consequence, schedulers optimized for mixed-criticality, here SWAP, cannot exploit their advantages.

		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$																	
success		10%		20%		30%		40%		50%		60%		70%		80%		90%	
ratio [%]		SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS
HI-*ratio*	0%	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	100%	100.0	100.0	100.0	100.0	100.0	100.0	19.7	88.3	0.0	0.8	0.0	0.5	10.6	66.8	3.4	66.3	1.8	73.7

Table 5.7: *Success ratios for corner cases and* $hsf = 3$.

In the following, we analyze and compare FPS and SWAP for job sets with 25%, 50% and 75% ratio of HI-ciritcality jobs.

Table 5.8 shows the success ratios for low LO-criticality utilizations. In case of these low utilizations, both algorithms can schedule (almost) all job sets. FPS shows a slight decrease in the success ratio already for LO-criticality utilizations of 30%. Including all jobs into HI-*table* increases the utilization and results in the observed decrease of success ratio.

		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$					
success		10%		20%		30%	
ratio [%]		SWAP	FPS	SWAP	FPS	SWAP	FPS
HI-*ratio*	25%	100.0	100.0	100.0	100.0	100.0	98.3
	50%	100.0	100.0	100.0	100.0	100.0	96.0
	75%	100.0	100.0	100.0	100.0	100.0	96.0

Table 5.8: *Success ratios for low LO-criticality utilizations and $hsf = 3$.*

Table 5.9 shows the success ratios for medium LO-criticality utilizations. Further, the results are also depicted in Figure 5.23. An increasing LO-criticality utilization and/or HI-*ratio* show a strong impact for both algorithms. On the one hand, an increasing HI-*ratio* has a stronger affect on the success ratio than increasing the LO-criticality utilization in case of SWAP. On the other hand for FPS, an increasing LO-criticality utilization affects the success ratio stronger than increasing the HI-*ratio*. The EDF-driven selection function of SWAP advantages high utilizations in contrast to Audsley's-fixed-priority-approach for FPS. Due to the fact that SWAP uses LO-criticality execution and idle time, SWAP suffers from high HI-*ratio* values. In the range of the presented medium LO-criticality utilizations with HI-*ratio*=50%, the HI-criticality utilizations are already in the range of more than 60%. For higher LO-criticality utilizations and higher HI-*ratio* values, this effect becomes even more important. In summary, the results show better success ratios for SWAP.

		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$					
success		40%		50%		60%	
ratio [%]		SWAP	FPS	SWAP	FPS	SWAP	FPS
HI-*ratio*	25%	94.4	62.4	54.1	7.3	14.6	0.4
	50%	79.8	37.7	9.9	1.0	0.3	0.0
	75%	51.1	25.4	0.2	0.1	0.0	0.0

Table 5.9: *Success ratio for medium LO-criticality utilizations and $hsf = 3$.*

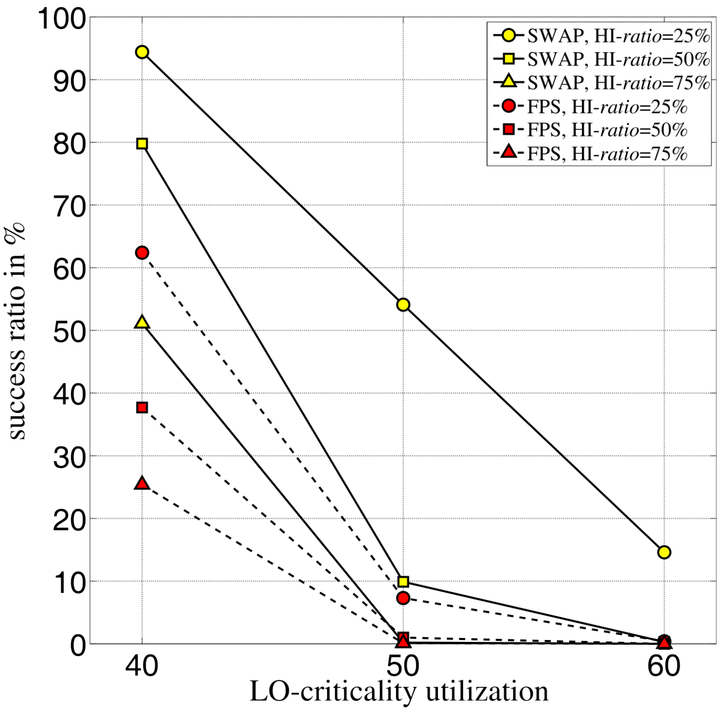


Figure 5.23: *Success ratio for medium* LO*-criticality utilizations.*

In Table 5.10, we show the success ratios for medium LO-criticality utilizations which are also shown in Figure 5.24. Keep in mind that input parameters are changed to create high utilization job sets and hence, the success ratio results are incomparable with the results of low/medium LO-criticality utilization job sets. FPS suffers from the fixed priority assignment and as a consequence, high LO-criticality utilizations result in very low success ratios. On the contrary, our EDF-based method SWAP allows for scheduling higher utilizations, but there is also a drastic drop of success ratios for SWAP for these high LO-criticality utilizations. As seen before, an increasing LO-criticality utilization has strong impact on success ratio of FPS. Similarly, an increasing HI-*ratio* has strong impact on success ratio of SWAP. In summary, $hsf = 3$ and LO-criticality utilization of 70% and more result in extremely high HI-criticality utilizations which are very difficult to schedule.

		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$					
success		70%		80%		90%	
ratio [%]		SWAP	FPS	SWAP	FPS	SWAP	FPS
HI-*ratio*	25%	67.5	10.4	42.1	1.1	18.4	0.1
	50%	20.3	5.4	4.8	0.7	1.0	0.2
	75%	7.4	3.7	1.4	0.8	0.1	0.2

Table 5.10: *Success ratio for high utilizations and* $hsf = 3$.

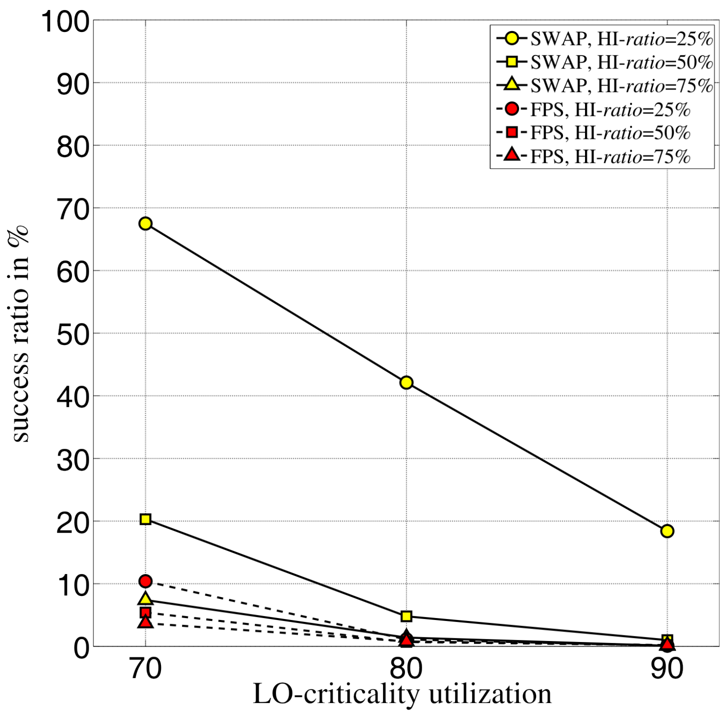


Figure 5.24: *Success ratio for high* LO*-criticality utilizations.*

5.6.5 Runtime Analysis

In the following, we compare FPS and SWAP by their runtimes. We measured the runtimes of each scheduler for the experiments presented before within the three utilization groups. All runtime measurements are performed on an Intel Xeon E5345 processor running at 2.33 GHz. The memory consumption of both algorithms are negligible (only a few megabytes). As before, the results are grouped into low, medium and high LO-criticality utilizations.

In Table 5.11, we present the mean runtimes for low LO-criticality utilizations. The runtimes for both algorithms are up to approximately 1 second. Assigning the job priorities in FPS is the dominating factor for its runtimes and the complexity of it depends on the number of jobs. For the low LO-criticality utilizations, only few iteration steps for FPS and no/few backtracking steps for SWAP are necessary which results in the small runtimes.

Table 5.12 presents the mean runtimes for medium LO-criticality utilizations. The runtimes of FPS drastically increase with increasing LO-criticality utilization and/or increasing HI-*ratio*. If there is no priority assignment, FPS checks all possible jobs for a lowest priority job. In the worst case, in each iteration step the job checked at last can have lowest priority until in the last iteration step no job can have lowest priority. The runtimes of SWAP even decrease with increasing LO-criticality utilization and/or increasing HI-*ratio*. The reason for the decreasing runtimes is that the higher the utilization the lower the success ratio is and with an uniformly distributed high utilization an early deadline miss is likely. To illustrate this with an example which

mean runtime [s]		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$ 10% SWAP	10% FPS	20% SWAP	20% FPS	30% SWAP	30% FPS
HI-*ratio*	25%	0.095	0.699	0.083	0.567	0.115	1.006
	50%	0.110	0.626	0.107	0.688	0.146	1.071
	75%	0.117	0.617	0.131	0.752	0.148	0.960

Table 5.11: *Mean values of runtime for low* LO*-criticality utilizations and* $hsf = 3$.

cannot be scheduled, we assume that a job with earliest deadline cannot be scheduled. The SWAP scheduler detects the deadline miss and checks the slots which have already been scheduled which is only a small part of entire schedule. Hence, SWAP can early detect that the scheduling process cannot be completed. On the contrary, the FPS scheduler checks whether this job can have lowest priority. If this not the case, the scheduler checks all other jobs for a possible lowest priority assignment. Maybe another job can have lowest priority and hence, many other job will be checked before the scheduler knows that job with earliest deadline cannot be scheduled. As a result, if both schedulers cannot find a solution, SWAP stops earlier than FPS which checks many possible priority assignments before stopping the process.

mean runtime [s]		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$ 40% SWAP	40% FPS	50% SWAP	50% FPS	60% SWAP	60% FPS
HI-*ratio*	25%	0.388	5.219	0.233	7.301	0.074	12.322
	50%	0.385	5.618	0.071	17.955	0.013	116.500
	75%	0.300	6.308	0.014	170.840	0.005	2150.005

Table 5.12: *Mean values of runtime for medium* LO*-criticality utilizations and* $hsf = 3$.

Table 5.13 presents the mean runtimes for high LO-criticality utilizations. The runtime results for high LO-criticality utilizations confirm the results of medium LO-criticality utilizations. All comments for medium LO-criticality utilizations are also valid for this case.

In the following, we compare FPS and SWAP by dividing the runtimes of FPS (t_{FPS}) by the runtimes of SWAP (t_{SWAP}). Table 5.14 presents the mean values for the runtime fraction (RTF), i.e., the quotient runtime FPS divided by runtime SWAP. For each job set, we calculate RTF and the mean values for each combination of LO-criticality utilization and HI-*ratio* are shown in the table. For low LO-criticality utilizations, when both schedulers can schedule (almost) all job sets, FPS needs about three to four times

mean runtime [s]		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$					
		70%		80%		90%	
		SWAP	FPS	SWAP	FPS	SWAP	FPS
HI-*ratio*	25%	0.127	1.155	0.082	1.335	0.039	1.946
	50%	0.055	5.697	0.019	9.822	0.007	28.774
	75%	0.250	52.073	0.008	75.017	0.004	128.418

Table 5.13: *Mean values of runtime for high* LO*-criticality utilizations and* $hsf = 3$.

$RTF = \frac{t_{FPS}}{t_{SWAP}}$		$\sum_{i=1}^{n} U_i(\text{LO})$								
		10%	20%	30%	40%	50%	60%	70%	80%	90%
HI-*ratio*	25%	3.6	3.6	4.0	14	114	485	24.7	62.3	160
	50%	3.1	3.1	3.6	51	1153	12430	493	1374	5094
	75%	2.7	2.9	3.2	139	18564	35204	8006	14560	28732

Table 5.14: *Mean values of runtime FPS divided by runtime SWAP.*

longer than SWAP. The difference between FPS and SWAP drastically increases with high LO-criticality utilizations and increasing HI-*ratio*s.

Figure 5.25 shows the cumulative distribution function (CDF) representing of the runtime fractions RTF for all 27,000 job sets, i.e., 1,000 job sets per parameter combination LO-criticality utilization (10%, 20%, ..., 90%) and HI-*ratio*s (25%, 50%, 75%). In 50% of the cases, FPS needs at least nine times as long as SWAP. Further, there are also cases where FPS needs more 100,000 times as long as SWAP. The cases, where FPS needs drastically longer than SWAP, are job sets which are not schedulable. Because of the iterative search for a priority assignment, FPS needs a lot of time to check whether there is a priority assignment.

Figure 5.26 shows a histogram showing the distribution of RTF below 10,000 which is about 90% of all results. Each bar represents the number of resulting RTF values which are between its borders. 2507 values ($\equiv$ 9.3%) are above 10,000 and are not shown in the figure. In summary, 96.9% of the results are above 2, i.e., FPS takes at least twice as long as SWAP. Further, 99.5% of the results are above 1.5. Finally, no run of FPS was quicker than the same job set scheduled by SWAP (no value for $RTF < 1$).

5.6.6 Summary of Results

Each success ratio and mean runtime presented in this section are based on 1,000 randomly generated job sets. An increasing utilization, both for LO-criticality and HI-criticality assumptions, makes scheduling harder. Further, an increasing hsf, i.e., intro-

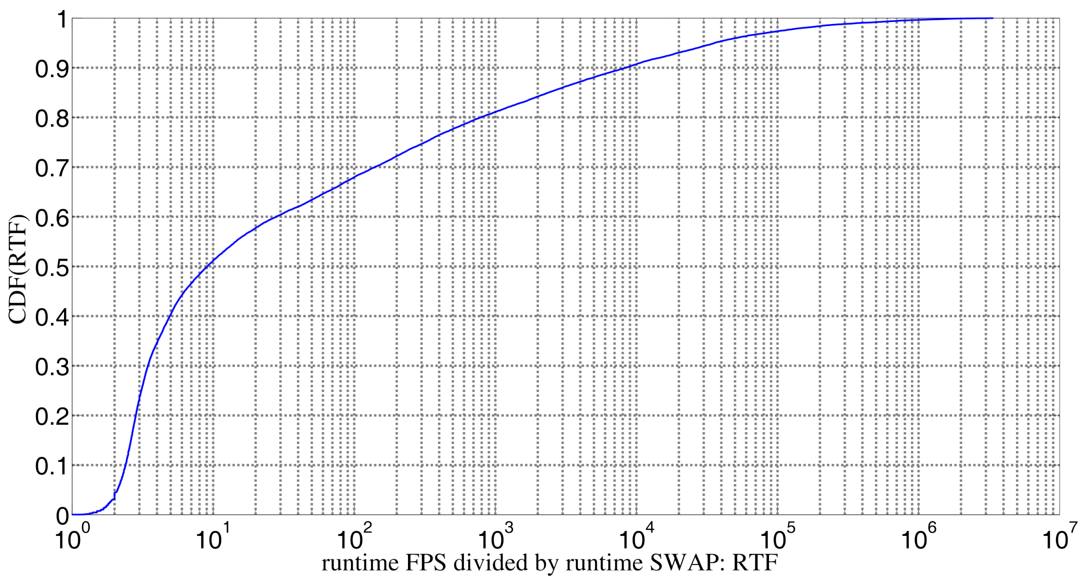


Figure 5.25: *Cumulative distribution function (CDF) of the runtime fractions RTF.*

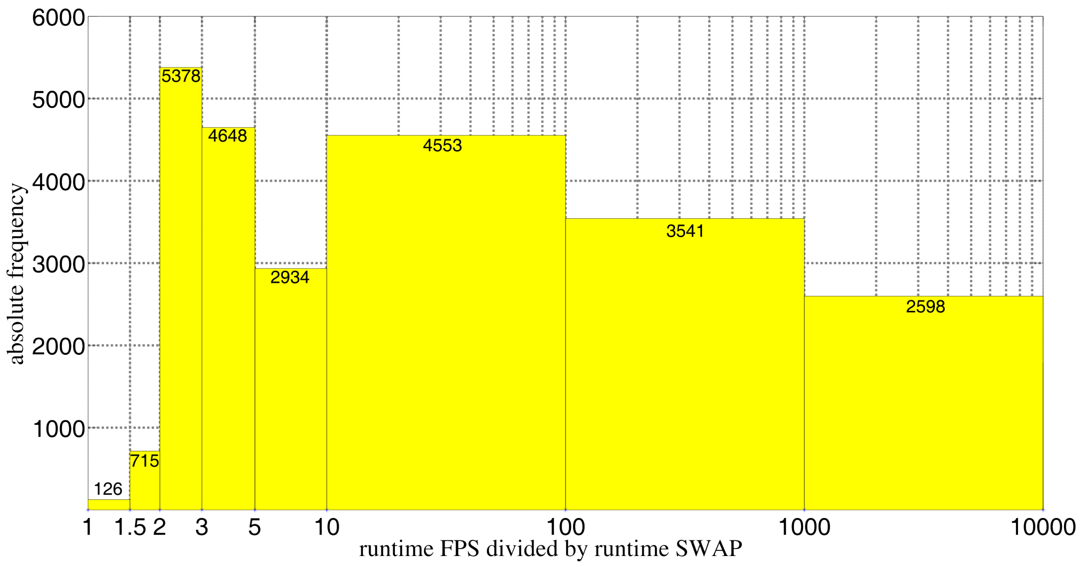


Figure 5.26: *Histogram of the runtime fractions RTF.*

ducing more pessimistic WCETs for HI-criticality behavior, corresponds to an increasing HI-criticality utilization. Also, a higher HI-*ratio* increases the HI-criticality utilization.

We showed corner cases with only LO-criticality jobs and only HI-criticality jobs. For only LO-criticality jobs, both algorithms perform well. This case corresponds to a standard scheduling problem with an optimal priority assignment or the optimal EDF scheduler. On the contrary, for only HI-criticality jobs, SWAP cannot exploit its advantages which are based on idle time and scheduled execution of LO-criticality jobs. FPS is less affected by the HI-*ratio* because only the utilization is important and not which criticality level creates the workload. These two corner cases do not represent mixed-critical systems because there is only one criticality level in this case.

For the general cases, i.e., job sets with more than one criticality level, SWAP's performance decreases with increasing HI-*ratio* whereas FPS is stronger affected by increasing LO-criticality utilization. Additionally, the runtime of SWAP is much better than the runtime of FPS. In particular, for job sets which cannot be scheduled by both algorithms, SWAP aborts the scheduling process earlier while FPS iterates through the job set to check for possible priority assignments. SWAP is at least twice as fast as FPS in 96.9% of presented cases. Furthermore, for no tested job set FPS is quicker than SWAP. FPS shows decreasing success ratio while runtime is increasing. When observing decreasing success ratio in SWAP, runtimes are decreasing. If we exclude the corner cases, which are not mixed-criticality job sets, SWAP performs better than FPS w.r.t. the success ratio and the runtime. We repeated all experiments with different input parameters which confirm the presented results. We show the further results in in the appendix in Section A.2.

5.7 Extensions

We developed our method for uniprocessor dual-criticality systems. The ideas presented so far are generic such that an extension based on the shown methods is possible. In the following, we present the fundamental information to extend our methods to systems with more than two criticality levels. Further, we show the basic idea to extend the methods to multiprocessor systems.

As mentioned earlier, the presented methods work on job-basis and the extension to handle periodic tasks is simple. Tasks are unrolled into a sequence of jobs with release time and deadline. For the WCET assignment, it is reasonable to assume that all jobs of a periodic tasks share the same WCETs, i.e., $C_i(\text{LO})$ and $C_i(\text{HI})$ of tasks for all their jobs.

For a mixed-criticality system with k criticality levels, we create k schedule tables, i.e., one schedule table for each criticality level. In the following, we assume as lowest criticality level $\chi = 1$. As a result, the mode m_1 with criticality level 1 includes all jobs. In general, mode m_c includes all jobs J_i with criticality level $\chi_i \geq m_c$ with WCET $C(m_c)$. The construction of the schedule tables is analogue to our presented method: the creation of the schedule table for mode m_1 is based on EDF. Modes with criticality level greater than 1 are based on modes with lower criticality level. A job J_i with $\chi_i = c$ is scheduled in the c lowest schedule tables. For schedule tables with higher criticality, we schedule a job representing additional needed execution time for the criticality case greater than m_i with fulfilled precedence constraints, lowest criticality level and earliest deadline. For instance, in mode m_4 given a job J_i with criticality level $\chi_i = 6$ is scheduled with its WCET of $C_i(6) - C_i(5)$ in all criticality modes greater than m_4 and less than or equal to criticality level of the job; here 6. We repeat this procedure until mode m_k is scheduled. For each slot, we calculate $k-1$ leeway values. The leeway includes additional workload for each criticality level between 1 and $k-1$, i.e., $\delta^1(s)$ includes additional workload of all jobs with criticality level between 2 and k, and so on until $\delta^{k-1}(s)$ which includes additional workload of jobs with criticality level k. The backtracking method swapping is more complex due to the fact that several leeway values and precedence constraints have to be considered. We have to check whether swapping two slots in all modes can lead to an infeasible schedule table in the future. In other words, we check whether the job in the swapping slot can be delayed until the current slot without resulting in a negative leeway. This has to be checked for the jobs in the swapping slot in all modes. As a consequence, many different cases have to be handled when swapping. A simplification by using the minimum leeway values of the slots for backtracking results in worse results because this can be very pessimistic. As a result of this schedule table construction, there can be a maximum number of $k-1$ mode-changes.

In general, multiprocessor systems can be classified into partitioned and global approaches (and also hybrid versions). In a partitioned multiprocessor system, uniprocessor methods can be applied based on a given job partitioning. As a result, our presented methods can be applied without any change. For global multiprocessor scheduling, we assume a system with M processing units. We select M jobs with earliest deadlines to schedule them in the LO-criticality schedule tables (one job for each processing unit). As a consequence, the switch-through condition can be relaxed: the same job J_i^{LO} need not to be schedule in same slot in HI-*table* on the same processing unit but only on any processing unit in the system in this slot. However, this results in an increasing complexity for backtracking with the number of processing units. A scheduling decision resulting in a negative leeway can be swapped with another job in the same schedule table or with a job in another schedule table on a different processing unit. As a result, we get an improvement of schedulability at cost of an increasing complexity.

5.8 Open Questions

In the following, we discuss open problems to improve the scheduling process. Further, we justify the rationale of our method and show alternatives.

5.8.1 Reducing the Number of Scheduling Decisions

The granularity of our scheduler is based on the length of a slot. Smaller slot sizes increase the percentage overhead within a slot and hence, increase the total accumulated overhead at runtime. On the contrary, larger slot sizes increase unused time in a slot when a job does not need the full slot size for computation or finishes computation in the middle of a slot, respectively. Based on a predetermined slot size, the scheduler can create a schedule table based on "chunks" of slots. In other words, instead of scheduling slot by slot, we schedule several slots together instead of selecting the same job several times in sequence. As a result, we can reduce the scheduling complexity. As a consequence the problem of a more complex backtracking mechanism occurs because of different "chunk" sizes. Dividing these "chunks" into fractions with the size of one or several slots to swap jobs, and thus, improving results, also increases the complexity of the backtracking.

Another possibility is to schedule the jobs non-preemptively and only divide jobs into slot-sized fractions if the leeway is negative and thus, indicates that the process will result in an infeasible schedule table. In this case, we have to consider several new aspects of the problem: First, we start with scheduling a job non-preemptively in both schedule tables LO-*table* and HI-*table*.

Assume we start the scheduling process with a HI-criticality job. In LO-*table*, we schedule this HI-criticality job for its LO-criticality WCET $C_i(\text{LO})$ and in HI-*table*, we can schedule the job its HI-criticality WCET $C_i(\text{HI})$. In this way, we do not need to make use of the split jobs and can avoid the introduced precedence constraint when we split jobs. A job has the same start time in both schedule tables, and is continuously scheduled until its completion, and thus, we can obtain the switch-through property. The problem in this case are the different completion times of the job in LO-*table* and HI-*table*. This will affect future scheduling decisions. If we schedule another HI-criticality job afterwards, the start time is determined by the completion time in HI-*table*, which is later than the completion time in LO-*table*. On the contrary, if we schedule a LO-criticality job next, we start scheduling it at the completion time in LO-*table*, because the LO-criticality job is only scheduled in LO-*table* and can be scheduled in parallel with the additionally needed execution time of the previous HI-criticality job in HI-*table*.

Now assume we start the scheduling process with a LO-criticality job. We schedule this job only in LO-*table*. If the next job to schedule is also a LO-criticality one, we can attach this job at the completion time of the previous one in LO-*table*. Further, if we schedule a HI-criticality job, we also set the start time of this job to the completion time of the previous LO-criticality job because we schedule this in both schedule tables in parallel and thus, we cannot make use of the idle time in HI-table parallel to the LO-criticality job.

By constructing the schedule tables in this way, we can minimize preemptions. The calculation of the leeway is also simplified because we need only to calculate the leeway for the completion time in LO-*table* instead of for every single slot.

The disadvantage of this procedure is the backtracking procedure when the leeway calculation results in a negative value. A negative leeway will give us the number of slots we have to swap. For instance, a leeway of minus three means that we missed the deadline by three slots. As a consequence, we have to swap the last three slots in LO-*table*. If the job has HI-criticality level, we have to re-schedule the additionally needed slots to execute in HI-criticality case. To apply swapping for the mentioned slots, we have to divide the non-preemptively scheduled job into slot-sized fractions. This has to be done for jobs until the release time of the job which caused a negative leeway because we cannot swap this job before its release time. When we swap the slots, we have to follow the rules we presented in Section 5.4 for our presented method.

In summary, we can reduce the scheduling complexity by scheduling jobs non-preemptively at cost of an increased overhead of the backtracking procedure. If we apply backtracking, we use the job set with the split jobs and change scheduling decisions with the granularity of a slot. As a consequence, the general behavior of the scheduler will change which has to be analyzed and compared to the existing approach. The results will show whether we can improve the effectiveness and in how far the runtimes, i.e., the efficiency of the method, is affected.

5.8.2 Rationale of the Order of Scheduling Decisions

In the presented methods and extensions presented so far, we always started in the LO-criticality schedule table. In the following, we justify this decision and show different starting points for the scheduling process.

In our method, the schedule table generation is based on the designers' assumptions. We chose to base the method on designers' assumptions because a proper design process should result in correct assumptions. Our methods create the LO-criticality schedule table first and adapts the HI-criticality schedule table decisions to the LO-criticality table. Hence, the leeway considers additional demand introduced by pessimism of CAs.
On the one hand, it is also possible to start with the HI-criticality schedule table and adapt LO-criticality schedule table. In this case, the leeway has to consider LO-criticality jobs. As a consequence, we adapt the normal system behavior, i.e., the LO-criticality behavior, to the special case of HI-criticality behavior which is not intuitive.
On the other hand, we can also construct both LO-criticality and HI-criticality schedule together considering all possible combinations. This will result in a very complex scheduling strategy which is similar to an exhaustive search with very high complexity.
To summarize this, we focused on the assumptions for which the designers assume that they are correct and a mode-change to HI-criticality behavior will never occur or a mode-change is extremely unlikely to occur, respectively. This also validates the assumption that we do not need to include LO-criticality jobs into the HI-criticality mode.

5.9 Discussion: How Can We Reduce the Complexity of Scheduling and Certification?

In this section, we summarize the advantages of the presented methods. The goal of our approach was to create TT schedule tables for a mode changes scheduler with a dual-criticality job set. Further, switching from LO-criticality to HI-criticality mode should be possible at every time instant (switch-through property). The approach is based on a search tree whereas classic tree search scheduling can be in complex. Jobs are characterized by two WCETs which increases the complexity of scheduling. Furthermore, creating two scheduling tables with switch-through property also increases the complexity of scheduling.

In a first step, we reduce the complexity of scheduling problem by splitting jobs. We separate the demand of determined by the designers and the additional demand introduced by the CAs. This results in jobs with only one WCET. Thus, splitting jobs reduces the complexity of the scheduling problem.

Additionally, we introduce the leeway of a slot. Instead of scheduling slot by slot until a job misses its deadline and apply backtracking, we can use the leeway for an early detection of paths in the search tree resulting in infeasible schedules. Further, the leeway is used to guide our backtracking method. Instead of checking every possible scheduling decision, scheduling decisions are changed that can result in feasible schedules. As a result, we can reduce the number of scheduling decisions and possible backtracking steps which results in a reduced scheduling complexity.

Finally, the creation of TT schedule tables represents a constructive proof. Hence, there is no need to certify all possible scheduling decisions of a scheduler. Checking the correctness of two schedules tables simplifies the certification complexity. Further, additional optimizations to improve response times, latencies, jitter, etc. can be implemented into the offline scheduler without changing the certification procedures.

Chapter 6

Mixed-Criticality Slot-Shifting without Mode Changes

In this chapter, we describe how we can increase the flexibility of time-triggered (TT) schedule tables. Based on the job parameters release times, deadlines, LO-criticality, and HI-criticality worst-case execution times (WCETs), we determine the amount and distribution of available resources. We differentiate between available resources if jobs show LO-behavior or show HI-behavior. The selection function of our scheduler uses this knowledge to schedule the dual-criticality jobs. With the presented scheduler, we can accommodate LO-criticality and HI-criticality behavior of jobs without the need for mode changes. Further, we show an acceptance test for firm aperiodic jobs, which can be integrated into the schedule without violation of already guaranteed jobs. Additionally, soft aperiodic jobs can be included while improving their response times without interfering guaranteed jobs. Our presented approach allows for an scheduling process which reacts to the actual behavior of jobs.

In Section 6.1, we explain why TT schedule tables are inflexible and mention the reason why flexibility is needed in mixed-criticality systems. After that, Section 6.2 reviews the original slot-shifting approach which forms the basis of our mixed-criticality scheduler. In the next section, we present the two phases of our approach. In Section 6.3.1, we show the offline phase and in Section 6.3.2, we present the online phase. An example in Section 6.4 illustrates the behavior and clarifies the method. A discussion in Section 6.5 concludes this chapter.

6.1 Motivation

In Chapter 5, we presented a method to construct time-triggered (TT) schedule tables with mode changes. The constructed schedule tables determine exact start and end times of job executions. The major advantage of this approach is full determinism and hence, a simplified verification and certification process. On the contrary, the resulting schedule tables are completely inflexible at runtime. As a consequence, there is the need for a more flexible method.

In this chapter, we present a method to allow for some flexibility at runtime based on slot-shifting [Foh95]. Slot-shifting uses offline constructed schedule tables but in contrast to the schedule tables presented before, these tables contain earliest start times and deadlines for each job instead of an exact job to slot assignment. As a result, the exact start and end times of jobs are not predetermined. Further, slot-shifting allows for shifting job executions within their execution window, i.e., within the time interval between release (earliest start time) and deadline of a job. By doing this, we get some flexibility in the execution of jobs. Additionally, already certified schedule tables can be used , e.g., schedule tables of TT legacy systems.

In the following, we present our slot-shifting based method to flexibly schedule mixed-criticality job sets at runtime. We assume job parameters are given based on certified schedule tables, e.g., constructed for legacy systems. We use the Vestal mixed-criticality model, shown in Chapter 3, with two criticality levels LO and HI for mixed-criticality jobs. The parameters are represented by the symbols shown in Chapter 3. We develop our method such that we can accommodate for LO-criticality and HI-criticality behavior of mixed-criticality jobs without the need to use mode changes. Without the need of changing the mode of the scheduler, we do not have to decide whether to include LO-criticality jobs into the HI-criticality table or not. As a result, we do not necessarily exclude LO-criticality jobs if a HI-criticality job exceeds its LO-criticality worst-case execution time (WCET). Based on the actual behavior, it is possible to execute LO-criticality jobs. Thus, decisions are based on actual behavior and not based on the pessimistic worst-case estimations. As a consequence, we can guarantee the worst-case behavior of HI-criticality jobs and still allow for LO-criticality job executions without interference of HI-criticality jobs.

Another advantage of our approach is the possibility to include event-triggered (ET) activities, i.e., aperiodic and sporadic jobs and tasks at runtime. Based on slot-shifting's acceptance test, we provide an acceptance test for mixed-criticality ET activities. We present detailed assumptions about ET jobs and tasks later in this chapter.

In the following, we start with a description of the original slot-shifting method and then, we continue with our mixed-criticality slot-shifting based on TT legacy system schedule tables.

6.2 Original Slot-Shifting

In the following, we describe the original slot-shifting algorithm as presented by Fohler in [Foh95]. Slot-shifting is a method to integrate ET activities into TT systems [IF99, IF00, IF09]. The method assumes a distributed system where on each node a predefined set of jobs is executed with slot-shifting as scheduling algorithm. Slot-shifting consists of two phases: an offline and an online phase. Before runtime, precedence constraints are resolved and jobs are scheduled in a schedule table containing earliest start times and deadlines for each job. In a further step, *capacity intervals* are determined based on the schedule table and *spare capacities* are calculated, i.e., unused resources within these intervals. At runtime, the capacity intervals and spare capacities are used to schedule jobs and updated according to the execution of jobs. Additionally, aperiodic jobs and jobs of sporadic tasks can be handled at runtime. For firm aperiodic jobs and jobs of sporadic tasks, an acceptance test is provided. The capacity intervals allow to control the scheduling overhead which results in lower overall overhead than, for instance, standard EDF schedulers [VDHBL+12]. Schorr and Fohler showed that slot-shifting has an applicable runtime overhead [SF13].

6.2.1 Offline Phase

Time-triggered (TT) jobs are characterized by their earliest start time (release time) r_i, deadline d_i and WCET C_i. The earliest start time of a job can be seen as release time of a job and thus, we use both terms as synonyms. Further, these jobs can have precedence constraints represented by a precedence graph. Figure 6.1 shows an example for such a precedence graph. The execution of the four jobs starts with job J_1 which is required to execute jobs J_2 and J_3. The order of execution for J_2 and J_3 can be arbitrary. Job J_4 can only start its execution after both J_2 and J_3 have finished their execution.

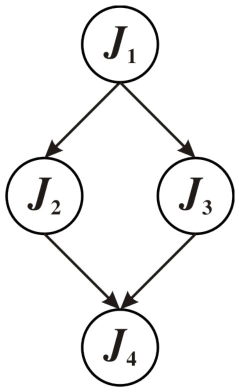


Figure 6.1: *Example precedence graph with four jobs.*

Precedence constraints are resolved by combining jobs into *scheduling blocks* which are assigned earliest start times and deadlines such that precedence constraints are obeyed. The combination of jobs into scheduling blocks reduces the number of objects to schedule which reduces the complexity of the scheduling process. In the following, we present slot-shifting by using the term job equivalent to scheduling block. These jobs are defined by their earliest start time, deadline and WCET. Slot-shifting assigns jobs to (capacity) intervals which are based on the earliest start times and deadlines of all jobs. We show the creation of intervals based on an example. Table 6.1 shows the job set used in the example.

jobs	WCET	earliest start time	deadline
J_1	1	0	3
J_2	1	5	8
J_3	2	7	12
J_4	2	8	12
J_5	3	9	14

Table 6.1: *Example job set to illustrate slot-shifting.*

Figure 6.2 shows the execution windows, i.e., the time interval between earliest start time and deadline, of jobs J_1, J_2, J_3, J_4, and J_5. Furthermore, the resulting intervals are shown below. The determination of intervals starts at the end of the schedule and ends at the start of the schedule. Each deadline marks the end of an interval, e.g., $end(I_4) = d_5$. Jobs are assigned to the interval for which the end of the interval is their deadline, e.g., job J_5 is assigned to the last interval I_4. In addition, the earliest start time (not start) of an interval is determined by the minimum earliest start time of jobs belonging to this interval, e.g., $est(I_3) = \min(r_3, r_4)$. The start of an interval is defined by the maximum of its earliest start time and the end of the previous interval, e.g., $start(I_3) = \max(est(I_3), end(I_2))$. Additionally, consecutive slots between these interval are also defined as intervals. These intervals are called *empty intervals*, in the figure: I_1. Intervals in which no execution windows overlap are called *independent intervals*, in the figure: I_0. Eventually, the length of an interval is calculated as follows:

$$|I_i| = end(I_i) - start(I_i).$$

In the next step, we calculate the spare capacities $sc(I_i)$ of all intervals I_i starting with the last interval. The spare capacities represent the available resources within an interval. The calculation of spare capacities in an empty interval is trivial: the available resources are determined by the length of the interval. For independent intervals, the difference between the length of the interval and the sum of the WCETs of the assigned jobs in that interval determines the spare capacities. Up to this, spare capacities are always greater than or equal to zero. For non-independent intervals, it is possible that the spare capacities are negative. If the spare capacities of an interval are negative, we have to *borrow* slots from an earlier interval. Keep in mind that (capacity) intervals are not equivalent to execution windows. As a consequence of borrowing, we also have to consider if an interval lent slots to the later interval. In this case, we reduce the available spare capacities by the amount of slots which are lent to the later interval. As a result, Equation (6.1) shows the spare capacity calculation of an interval I_i.

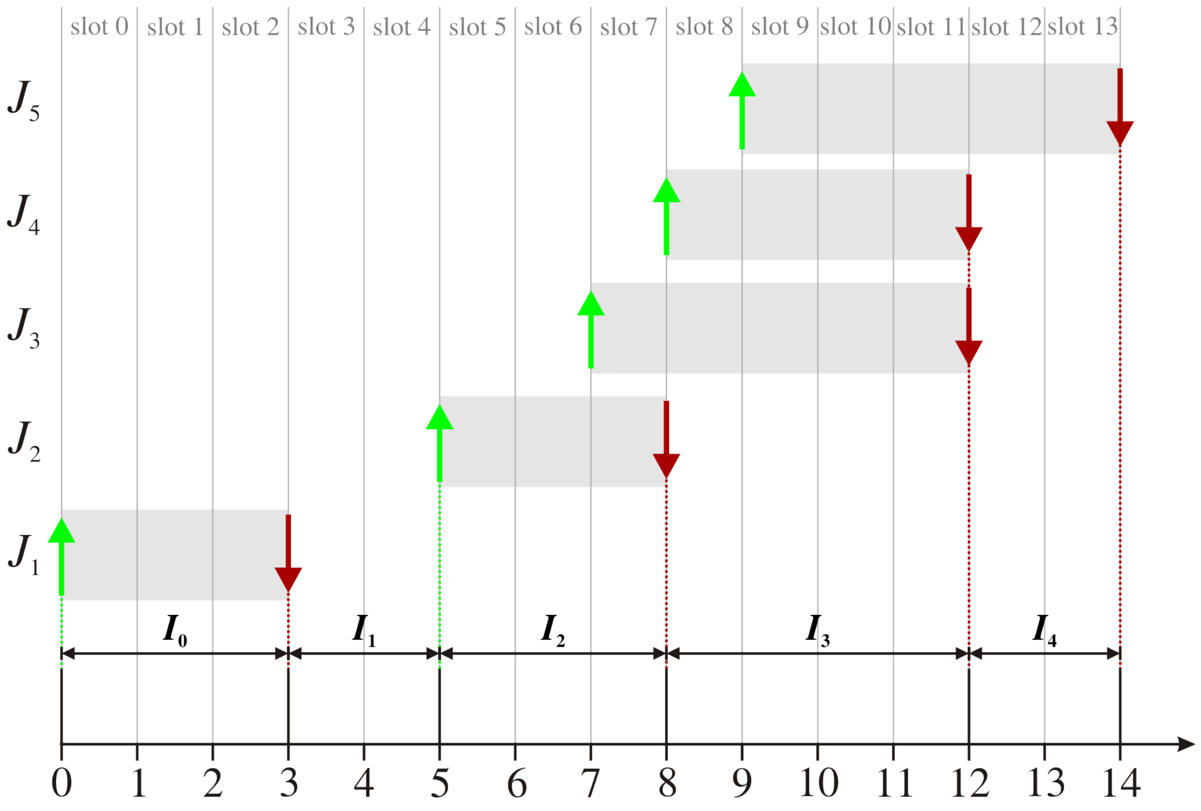


Figure 6.2: *Capacity interval derivation with five jobs.*

$$sc\left(I_i\right) = |I_i| - \sum_{J_k \in I_i} C_k + \min\left(sc\left(I_{i+1}\right), 0\right) \tag{6.1}$$

Figure 6.3 shows the resulting spare capacities for the example job set. We use as late as possible scheduling to illustrate the available spare capacities. Starting in the last interval, the spare capacities in interval I_4 are negative, i.e., this intervals borrows one slot from I_3 which is highlighted in slot 11 in the figure. Although interval I_3 is as long as the WCETs of the assigned jobs, the spare capacities are negative because I_3 lent one slot to I_4. As a consequence, I_3 also has to borrow one slot from an earlier interval. This phenomenon is called *borrowing propagation* and is highlighted in slot 7 in the figure. Borrowing and borrowing propagation will be important when we schedule jobs at runtime and update the spare capacities. Finally, the spare capacities of intervals I_0, I_1, and I_2 are positive which is illustrated in the figure by the available (idle) slots.

The calculation of the spare capacities concludes the offline phase. Based on the knowledge of intervals and their spare capacities, we schedule the jobs at runtime.

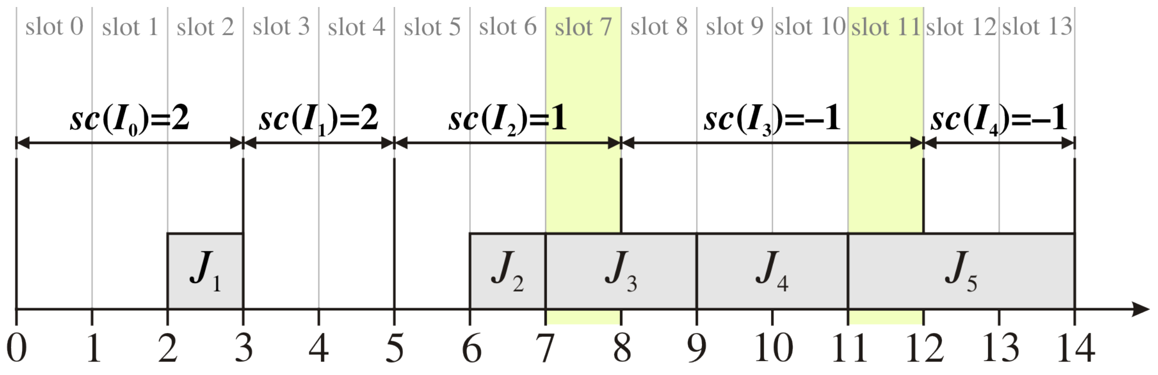


Figure 6.3: *Spare capacity calculation with five jobs.*

6.2.2 Online Phase

At runtime, we execute jobs from the schedule table by selecting the job with the earliest deadline. If there are no jobs ready to execute, we schedule an idle slot. Further, at arrival of firm aperiodic jobs or jobs of a sporadic task, i.e., event-triggered (ET) jobs, we perform an acceptance test to check whether we can integrate these jobs into the schedule without violation of already guaranteed jobs. Firm jobs which pass the test are included into the set of guaranteed jobs, i.e., TT and accepted ET jobs. Soft ET jobs do not need to perform an acceptance test because soft jobs are scheduled only if the spare capacities allow for an execution without violation of guaranteed jobs. Additionally, soft jobs do not have timing constraints. In the following, we review the selection function, spare capacity maintenance, and the acceptance test of original slot-shifting. We consider aperiodic jobs as ET activities, but all procedures are also valid for jobs of sporadic tasks.

The scheduler has a ready queue $R(t) = \{J_i | r_i \leq t\}$ with jobs that are ready at time t. The current interval is named I_c. Based on the following cases, a job to execute or an idle slot is selected:

a) $R(t) = \{\}$:
No job is ready to execute and hence, we schedule an idle slot.

b) $R(t) \neq \{\} \wedge sc(I_c) = 0$:
We schedule a guaranteed job with the earliest deadline because zero spare capacities indicate that we have to schedule a guaranteed job otherwise there will be a deadline miss.

c) $R(t) \neq \{\} \wedge sc(I_c) > 0 \wedge \nexists J_{\mathrm{A}i},\ J_{\mathrm{A}i}$ soft aperiodic job:
We schedule a guaranteed job with the earliest deadline.

d) $R(t) \neq \{\} \wedge sc(I_c) > 0 \wedge \exists J_{\mathrm{A}i},\ J_{\mathrm{A}i}$ soft aperiodic job:
We schedule an aperiodic job $J_{\mathrm{A}i}$. The positive spare capacities allow for a delay of the guaranteed jobs' execution.

In summary, we schedule soft aperiodic jobs only if the spare capacities of the current interval are positive. In contrast to background service, we know the amount of available resources and can schedule soft aperiodic jobs not only when there is no guaranteed job ready to execute. As a result, we improve the response times of soft aperiodic jobs.

Based on the scheduled job in a slot, we update the spare capacities. There are four cases to consider:

No execution: One slot of the spare capacities is unused and thus, $sc(I_c)$ is reduced by one slot.

Soft aperiodic execution: One slot of the spare capacities in the current intervals is used to execute a job which has not been considered when calculating the spare capacities and thus, $sc(I_c)$ is reduced by one slot.

Execution of a guaranteed job within its interval: There is no change in the spare capacities because the execution of this job has been considered in the spare capacity calculation.

Execution of a guaranteed job outside its interval: A job assigned to interval I_k consumes execution time in the current interval I_c. Hence, in the current interval the spare capacities are decreased by one slot and in I_k the spare capacities are increased by one slot, i.e., one slot has been swapped. If $sc(I_k) < 0$, there has been borrowing. As a consequence, one slot is now borrowed less in all intervals with borrowing and borrowing propagation which are affected by the earlier execution, and thus, we increase the spare capacities by one slot in these intervals.

The maintenance of spare capacities is performed after the execution at the end of each slot. It is also possible to include the spare capacity updates at the beginning of a slot when the scheduler selects the next executing job, but the spare capacities describe the available resources after the execution. The spare capacities of the current interval can be used as correctness criterion at the end of each interval, when the spare capacity value has to be zero, otherwise there has been an error.

So far, we considered scheduling of guaranteed jobs, i.e., TT jobs and ET jobs that passed the acceptance test. In the following, we present this acceptance test which is performed at the arrival of a firm aperiodic job. The test checks whether we can integrate and guarantee a firm aperiodic job without violation of the timing constraints of already guaranteed jobs. To do this, the test sums up all positive spare capacities until the deadline of the aperiodic job. These space capacities can be divided into 3 groups: spare capacities of the current interval; spare capacities of all full interval between the current interval and the interval including the aperiodic job deadline; and spare capacities in the interval with the job deadline until the job deadline. We consider only positive spare capacities because negative spare capacities are already considered in the previous interval when taking borrowing into account. Finally, the sum of the spare capacities has to be larger than or equal to the WCET of the aperiodic job to pass the test. Else, the aperiodic job is rejected. This condition is represented by Inequality (6.2).

$$\underbrace{sc(I_c)}_{\text{current interval}} + \underbrace{sc(I_{full})}_{\text{full intervals}} + \underbrace{sc(I_{dl})}_{\substack{\text{interval with aperiodic}\\ \text{job deadline}}} \geq C_{\mathrm{A}j} \tag{6.2}$$

$$\begin{aligned} \text{with} \quad & sc(I_{full}) = \sum_{c<i\leq l} \max\left(0, sc(I_i)\right) \\ \text{and} \quad & sc(I_{dl}) = \max\left(0, \min\left(sc(I_{l+1}), d_{\mathrm{A}j} - start(I_{l+1})\right)\right) \\ \text{and} \quad & end(I_l) < d_{\mathrm{A}j} \wedge end(I_{l+1}) \geq d_{\mathrm{A}j} \end{aligned}$$

After its acceptance, an aperiodic job is assigned to the set of guaranteed jobs. As a consequence, its WCET has to be included into the spare capacities. If the deadline of an aperiodic job is equal to an deadline, i.e., the end of an interval, which is already present in the system, then job is assigned to this interval. Else, the interval, in which the deadline is located, is split into two intervals with the aperiodic job deadline as new end of the first interval and also start of the second interval. Afterwards, the spare capacities of the interval or the two intervals after splitting have to be recalculated. This includes a possible re-calculation of earlier intervals due to borrowing and borrowing propagation.

6.3 Mixed-Criticality Slot-Shifting

In the following, we present our slot-shifting-based method to schedule mixed-criticality jobs without the need for mode changes. In this chapter, we use the Vestal model with the two criticality levels LO and HI. TT jobs are scheduled offline in a schedule table with earliest start times, deadlines, and WCETs. At runtime, these TT jobs are guaranteed to execute for their WCET within their execution window in the corresponding criticality level. Additionally, ET aperiodic jobs or jobs of sporadic tasks, i.e., jobs with unknown arrival time, are handled. In the following, we consider only aperiodic jobs, but the procedures are also valid for sporadic jobs. A benefit of this approach is that slot-shifting allows for non-work-conserving schedules based on spare capacities. Further, slot-shifting allows for scheduling of strictly periodic jobs, i.e., jobs that have to be executed directly at their periodic release. The approach is divided into an offline and an online phase which are presented in the following.

6.3.1 Offline Phase

In the offline phase of slot-shifting, we can resolve complex constraints, e.g., precedence constraints. The interested reader is referred to [Foh95]. We assume that complex constraints are resolved and focus on the scheduling of mixed-criticality jobs. In the following, we show how to determine capacity intervals and how to calculate spare capacities which form the basis for the slot-shifting runtime scheduler.

Capacity Intervals

We divide the schedule into disjoint *capacity intervals* I_i with $i \in \{0, ..., m\}$ based on the release times and deadlines of the jobs. As mentioned before capacity intervals are not identical to the execution windows, i.e., the time between release and deadline of a job. In the following, capacity intervals are briefly referred to as intervals. The intervals are not affected by the mixed-criticality nature of jobs, i.e., jobs with two WCETs. In the Vestal model, the release times and deadlines of jobs do not change with the criticality level and hence, the construction of intervals also does not change from the original slot-shifting. We briefly recapitulate the rules to construct the intervals. Each deadline of a job marks the end $end\,(I_i)$ of an interval I_i. Each job J_k with $k \in \{1, ..., n\}$ is assigned to an interval with $end\,(I_i) = d_k$. Jobs with the same deadline belong to the same interval. The earliest start time $est\,(I_i)$ of an interval I_i is determined by the minimum of all release times of jobs assigned to that interval:

$$est(I_i) = \min_{\forall J_k \in I_i} (r_k) \tag{6.3}$$

The start of an interval is determined by the maximum of its earliest start time and the end of the previous interval:

$$start(I_i) = max\,(est(I_i), end(I_{i-1})) \tag{6.4}$$

The gaps between the determined intervals above are defined as *empty intervals*, i.e., there is no job assigned to them. An interval I_i is called *independent* if there is no interval I_e with $e < i$ and $end(I_e) > est(I_i)$ and there is no interval I_l with $i < l$ and $end(I_i) > est(I_l)$. The length $|I_i|$ of an interval is calculated by Equation 6.5.

$$|I_i| = end(I_i) - start(I_i) \tag{6.5}$$

Spare Capacities

In the following, we explain how we can apply slot-shifting's concept of *spare capacities* to mixed-criticality job sets. The offline calculated spare capacities $sc(I_i)$ of an interval I_i represent the amount of available resources within this interval after guaranteeing TT jobs. The requirements of mixed-criticality job sets result in different spare capacities based on the behavior of jobs. At runtime, we perform an acceptance test for ET jobs with firm deadlines. We consider the demand of guaranteed ET jobs, i.e., jobs that passed the acceptance test, in the selection function and the maintenance of the spare capacities. In the following, we show how spare capacities are calculated for mixed-criticality job sets. We calculate spare capacities beginning with the last interval I_m until the first interval I_0.

We can test the schedulability based on the spare capacities under the condition of obeying release times and deadlines. Spare capacities of independent intervals have to be non-negative. Further, after borrowing and borrowing propagation there must be an earlier interval with non-negative spare capacity.

The calculation of mixed-criticality spare capacities is based on the calculation shown in Section 6.2 for original slot-shifting. We calculate two spare capacity values for each interval: $sc^{\text{LO}}(I_i)$ and $sc^{\text{HI}}(I_i)$. The calculation of spare capacities is done by Equations (6.6) and (6.7).

$$sc^{\text{LO}}\left(I_i\right) = |I_i| - \sum_{J_k \in I_i} C_k(\text{LO}) + min\left(sc^{\text{LO}}\left(I_{i+1}\right), 0\right) \tag{6.6}$$

$$sc^{\text{HI}}\left(I_i\right) = |I_i| - \sum_{\substack{J_k \in I_i \\ \wedge \chi_k = \text{HI}}} C_k(\text{HI}) + min\left(sc^{\text{HI}}\left(I_{i+1}\right), 0\right) \tag{6.7}$$

Spare capacities $sc^{\text{LO}}(I_i)$ represent available capacities based on designer's requirements, i.e., considering $C_k(\text{LO})$, including *all jobs*. If the LO-criticality spare capacities are equal to zero, we cannot include non-guaranteed jobs, i.e., soft ET jobs. Additionally, $sc^{\text{HI}}(I_i)$ represents the available resources based on certification authorities' (CAs) requirements, i.e., considering $C_k(\text{HI})$, considering *only* HI-*criticality jobs*. If the HI-criticality spare capacities are equal to zero, then we have to run a HI-criticality job, otherwise we cannot guarantee the HI-criticality WCETs.

6.3.2 Online Phase

At runtime, we execute mixed-criticality jobs based on the spare capacity in the current interval I_c and the deadlines of the jobs. Further, soft ET aperiodic jobs are executed if there are available resources to execute them without harming guaranteed jobs. We assume that soft aperiodic jobs always have LO-criticality. On the contrary, for firm aperiodic jobs, we perform an acceptance test to check whether we can guarantee the execution within their execution windows. If the firm aperiodic jobs have LO-criticality, we use the LO-criticality spare capacities sc^{LO} for the acceptance test. Although HI-criticality firm ET aperiodic jobs are possible, they are not reasonable because we cannot guarantee their execution. It is possible that one of these jobs does not pass the acceptance test and cannot be executed. In this case, we would reject HI-criticality jobs which are subject to certification. As a result, if we want to certify and guarantee ET activities, e.g., HI-criticality firm sporadic jobs, we need an offline reservation task, for instance, as we showed in Chapter 4, to include the worst-case demand into the spare capacity calculations. In case the designer wants to include these jobs despite the mentioned problems, we would have to perform two acceptance tests: one test with LO-criticality spare capacities under designer assumptions, i.e., using $C_{\text{A}k}(\text{LO})$, and one test with HI-criticality spare capacities under CAs' assumptions, i.e., using $C_{\text{A}k}(\text{HI})$. The HI-criticality aperiodic jobs have to pass both tests to get accepted. In the following, we assume only LO-criticality ET aperiodic jobs.

In the following, we present how to select the next job for execution. Further, we show the update procedures for spare capacities depending on the job execution. Finally, we present an acceptance test for LO-criticality ET jobs.

Decision Mode and Selection Function

The decision mode of the slot-shifting runtime scheduler is preemptive at slot borders. We use three ready queues: $R^{\text{LO}}(t)$, $R^{\text{HI}}(t)$, and $R^{\text{soft}}(t)$. Whereas,

$$R^{\text{LO}}(t) = \{J_i | r_i \leq t \wedge \chi_i = \text{LO}\} \cup \{J_{\text{A}i} | r_{\text{A}i} \leq t \wedge \chi_{\text{A}i} = \text{LO}\}$$

contains both TT and firm ET LO-criticality jobs. Further,

$$R^{\text{HI}}(t) = \{J_i | r_i \leq t \wedge \chi_i = \text{HI}\}$$

is used for TT HI-criticality jobs. Finally, the FIFO-queue

$$R^{\text{soft}}(t) = \{J_i | r_i \leq t\}$$

contains all soft aperiodic (LO-criticality) jobs. We define

$$R^{\text{firm}}(t) = R^{\text{LO}}(t) \cup R^{\text{HI}}(t)$$

which represents all guaranteed (firm) jobs to simplify our explanations. The scheduler selects the next executing jobs based on the ready queues, the intervals and the spare capacities. In contrast to original slot-shifting, LO-criticality spare capacities can be negative in the current interval I_c. This can occur when a HI-criticality job J_k executes for more than its LO-criticality WCET and we have to continue its execution. As long as the LO-criticality spare capacity in the current interval is negative, i.e., $sc^{\text{LO}}(I_c) < 0$, we can only execute HI-criticality jobs. Based on these observations, we distinguish the following possible decision cases at time t.

a) $R^{\text{firm}}(t) = \{\}$:

- **1)** $R^{\text{soft}}(t) = \{\}$:
  We schedule an idle slot because there are no ready jobs.
- **2)** $R^{\text{soft}}(t) \neq \{\}$:
  $sc^{\text{LO}}(I_c)$ and $sc^{\text{HI}}(I_c)$ are always greater than zero and hence, we can schedule a soft aperiodic job.

b) $R^{\text{firm}}(t) \neq \{\} \wedge sc^{\text{HI}}(I_c) > 0$:

- **1)** $sc^{\text{LO}}(I_c) > 0$:
  If $R^{\text{soft}}(t) \neq \{\}$ then we schedule the first soft aperiodic job in the FIFO-queue $R^{\text{soft}}(t)$. Else, we select the ready job with the earliest deadline in $R^{\text{firm}}(t)$. This is the only situation when soft aperiodic jobs are scheduled if guaranteed jobs are ready to execute. Figure 6.4 illustrates this situation with an example.
- **2)** $sc^{\text{LO}}(I_c) = 0$:
  We select the job with the earliest deadline in $R^{\text{firm}}(t)$ because there are enough available resources for both LO- and HI-criticality jobs and thus, there is no need to prioritize HI-criticality jobs. On the contrary, there are no spare capacities left for soft aperiodic demand and thus, we have to select a guaranteed job. Figure 6.5 illustrates this situation with an example.

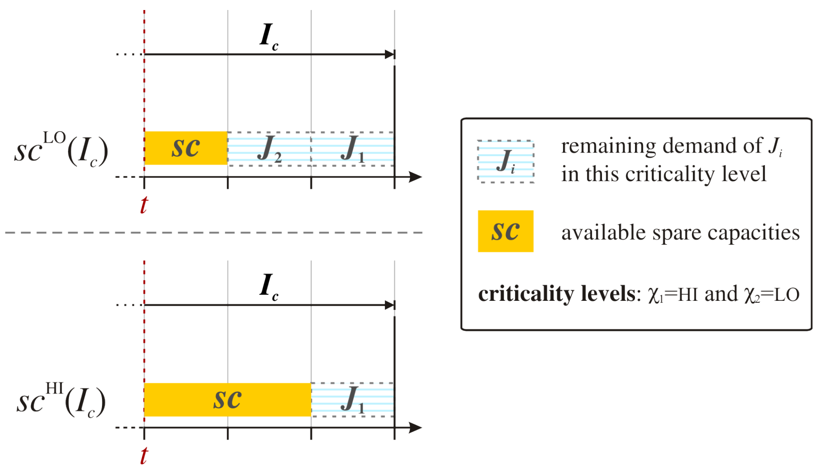

I_c
$sc^{LO}(I_c)$
sc
J_2
J_1
t
I_c
$sc^{HI}(I_c)$
sc
J_1
t
J_i
remaining demand of J_i in this criticality level
sc
available spare capacities
criticality levels: χ_1=HI and χ_2=LO

Figure 6.4: *Example situation when selecting the next job with* $R^{\text{LO}}(t) = \{J_2\}$, $R^{\text{HI}}(t) = \{J_1\}$, $sc^{\text{LO}}(I_c) > 0$, *and* $sc^{\text{HI}}(I_c) > 0$.

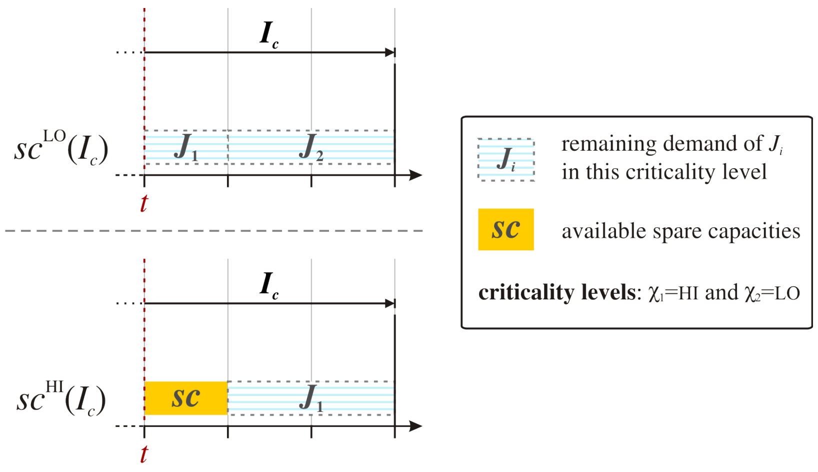

I_c
$sc^{LO}(I_c)$
J_1
J_2
t
I_c
$sc^{HI}(I_c)$
sc
J_1
t
J_i
remaining demand of J_i in this criticality level
sc
available spare capacities
criticality levels: χ_1=HI and χ_2=LO

Figure 6.5: *Example situation when selecting the next job with* $R^{\text{LO}}(t) = \{J_2\}$, $R^{\text{HI}}(t) = \{J_1\}$, $sc^{\text{LO}}(I_c) = 0$, *and* $sc^{\text{HI}}(I_c) > 0$.

3) $sc^{\text{LO}}(I_c) < 0$:
We select the job with the earliest deadline in $R^{\text{HI}}(t)$. There are not enough available LO-criticality spare capacities in the current interval to complete the remaining LO-criticality jobs and hence, they are skipped until enough resources are available to schedule both LO- and HI-criticality jobs again. Figure 6.6 illustrates this situation with an example.

c) $R^{\text{firm}}(t) \neq \{\} \wedge sc^{\text{HI}}(I_c) = 0$:

1) $sc^{\text{LO}}(I_c) > 0$:
The job with the earliest deadline in $R^{\text{HI}}(t)$ is selected because we have to execute a HI-criticality job to guarantee its completion with CAs' assumptions before its deadline. Figure 6.7 illustrates this situation with an example.

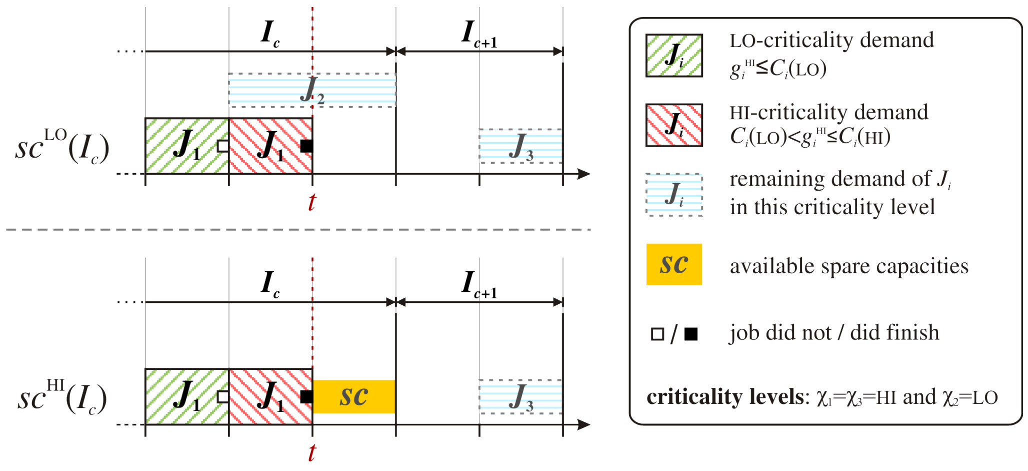

I_c
I_{c+1}
J_2
$sc^{LO}(I_c)$
J_1
J_1
J_3
t
$sc^{HI}(I_c)$
sc
LO-criticality demand $g_i^{HI} \leq C_i(LO)$
HI-criticality demand $C_i(LO) < g_i^{HI} \leq C_i(HI)$
remaining demand of J_i in this criticality level
available spare capacities
job did not / did finish
criticality levels: $\chi_1=\chi_3=$HI and $\chi_2=$LO

Figure 6.6: *Example situation when selecting the next job with* $R^{\text{LO}}(t) = \{J_2\}$, $R^{\text{HI}}(t) = \{J_3\}$, $sc^{\text{LO}}(I_c) < 0$, *and* $sc^{\text{HI}}(I_c) > 0$.

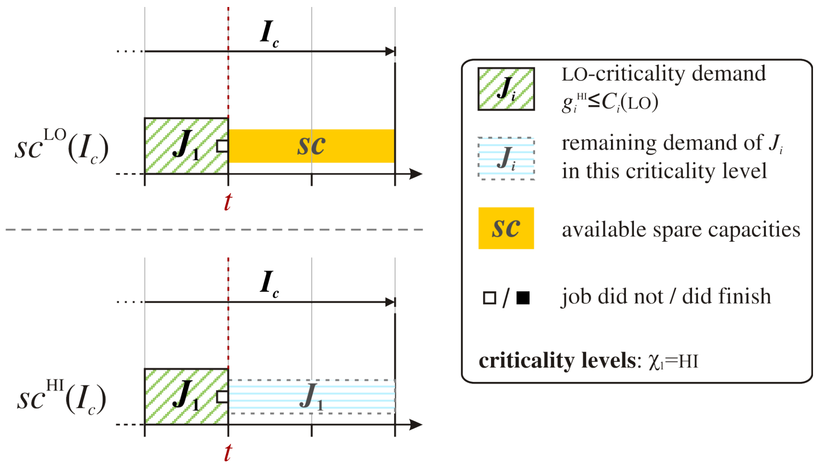

I_c
$sc^{LO}(I_c)$
J_1
sc
t
$sc^{HI}(I_c)$
J_1
LO-criticality demand $g_i^{HI} \leq C_i(LO)$
remaining demand of J_i in this criticality level
available spare capacities
job did not / did finish
criticality levels: $\chi_1=$HI

Figure 6.7: *Example situation when selecting the next job with* $R^{\text{LO}}(t) = \{\}$, $R^{\text{HI}}(t) = \{J_1\}$, $sc^{\text{LO}}(I_c) > 0$, *and* $sc^{\text{HI}}(I_c) = 0$.

2) $sc^{\text{LO}}(I_c) \leq 0$:
We select the job with the earliest deadline in $R^{\text{HI}}(t)$ because of the reasons mentioned in case c1. A consequence of this situation is that LO-criticality jobs are skipped. Figure 6.8 illustrates this situation with an example.

d) $R^{\text{firm}}(t) \neq \{\} \wedge sc^{\text{HI}}(I_c) < 0$:
The HI-criticality spare capacities cannot be less than zero. This could only happen if we execute a HI-criticality job J_k for more than $C_k(\text{HI})$ which we assume is prevented by the system.

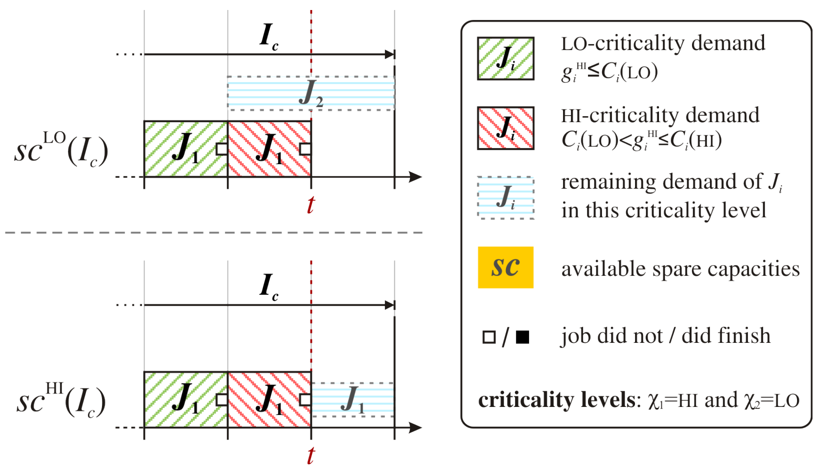

I_c
J_2
$sc^{LO}(I_c)$
J_1
J_1
t
I_c
$sc^{HI}(I_c)$
J_1
J_1
J_1
t
J_i LO-criticality demand $g_i^{HI} \leq C_i(LO)$
J_i HI-criticality demand $C_i(LO) < g_i^{HI} \leq C_i(HI)$
J_i remaining demand of J_i in this criticality level
sc available spare capacities
□ / ■ job did not / did finish
criticality levels: χ_1=HI and χ_2=LO

Figure 6.8: *Example situation when selecting the next job with* $R^{LO}(t) = \{J_2\}$, $R^{HI}(t) = \{J_1\}$, $sc^{LO}(I_c) \leq 0$, *and* $sc^{HI}(I_c) = 0$.

Spare Capacity Maintenance

As a consequence of the process to select the next executing job, we have to update the spare capacities depending on the criticality level and type of the job.
No execution: If an idle slot has been scheduled, we decrease both $sc^{LO}(I_c)$ and $sc^{HI}(I_c)$ by one slot.
Soft aperiodic execution: Soft aperiodic jobs are not considered in the spare capacity calculations. As a result, we have to decrease the LO- and HI-criticality spare capacities in the current interval by one slot.
Guaranteed job execution: If a guaranteed job J_i, either TT or firm ET (in this case: J_{Ai}), has been scheduled, then we have to differentiate whether J_i is assigned to the current interval I_c or to a later interval I_k. Further, the fact whether a HI-criticality job exceeded its LO-criticality WCET, i.e., exhibited HI-behavior, influences the maintenance of spare capacities.

A) $J_i \in I_c$

1) J_i did not exceed $C_i(\text{LO})$ and $\chi_i = \text{LO}$:
In the LO-criticality spare capacities $sc^{LO}(I_c)$, J_i has been considered and thus, $sc^{LO}(I_c)$ is not changed. On the contrary, in the HI-criticality spare capacities $sc^{HI}(I_c)$, J_i has not been included such that we decrease $sc^{HI}(I_c)$ by one slot.

2) J_i did not exceed $C_i(\text{LO})$ and $\chi_i = \text{HI}$:
In both spare capacity calculations, $sc^{LO}(I_c)$ and $sc^{HI}(I_c)$, the scheduled demand has already been considered and hence, both spare capacities in the current interval remain unchanged.

3) J_i exceeded $C_i(\text{LO})$:
As the job executes for more than the considered amount of LO-criticality execution time, which is only possible for HI-criticality jobs, we have to decrease the LO-criticality spare capacity in the current interval by one slot. On the contrary,

for the HI-criticality spare capacity $sc^{\text{HI}}(I_c)$, this scheduled execution has already been considered and thus, $sc^{\text{HI}}(I_c)$ is unchanged.

B) $J_i \in I_k$ with $I_k \neq I_c$ and $k > c$

1. J_i did not exceed $C_i(\text{LO})$ and $\chi_i = \text{LO}$:
The scheduled demand has not been considered in $sc^{\text{LO}}(I_c)$. As a consequence, we decrease $sc^{\text{LO}}(I_c)$ by one slot. Additionally, we increase $sc^{\text{LO}}(I_k)$ where the demand has been originally considered in the spare capacity calculations. In other words, one slot of execution is swapped between the current interval I_c and the assigned job interval I_k.

Furthermore, there is the aspect of borrowing which has to be considered: if $sc^{\text{LO}}(I_k)$ was less than zero before increasing by one in the current step, then I_k was borrowing capacity from at least one earlier interval. Thus, we have to increase the spare capacities by one if there was borrowing or borrowing propagation in one or several of the intervals from I_c to I_{k-1}.

In all HI-criticality spare capacities, job J_i has not been considered. As a result, we decrease $sc^{\text{HI}}(I_c)$ by one slot because we used one slot to execute a job which has not been considered in the current interval. Further, $sc^{\text{HI}}(I_k)$ is not affected by this and thus, not changed.

2. J_i did not exceed $C_i(\text{LO})$ and $\chi_i = \text{HI}$:
The LO-criticality spare capacities $sc^{\text{LO}}(I_c)$ and $sc^{\text{LO}}(I_k)$ are updated as shown before in step B1.

Additionally, we have to update the HI-criticality spare capacities. The scheduled demand has not been considered in $sc^{\text{HI}}(I_c)$. As a consequence, we decrease $sc^{\text{HI}}(I_c)$ by one slot and increase $sc^{\text{LO}}(I_k)$ where the demand has been originally included. Thus, we swap one slot of execution between the current interval I_c and the assigned job interval I_k.

Furthermore, the aspect of borrowing has to be considered also: if $sc^{\text{HI}}(I_k)$ was less than zero before increasing by one slot in the current step, then I_k was borrowing capacity from at least one earlier interval. Thus, we have to increase the spare capacities by one slot in all intervals from I_c to I_{k-1} if they were affected by borrowing or borrowing propagation.

3. J_i exceeded $C_i(\text{LO})$:
The demand has not been considered in the LO-criticality spare capacities, neither in $sc^{\text{LO}}(I_c)$ nor in $sc^{\text{LO}}(I_k)$. Thus, only the spare capacity in the current interval $sc^{\text{LO}}(I_c)$ is decreased by one.

For the HI-criticality spare capacities, we have to apply the same procedures as in step B2; which are:
The scheduled demand has not been considered in $sc^{\text{HI}}(I_c)$ and thus, we have to decrease $sc^{\text{HI}}(I_c)$ by one slot. Further, $sc^{\text{HI}}(I_k)$, where the demand has been originally considered, is increased by one.

Still, there may be the aspect of borrowing which has to be considered: if $sc^{\text{HI}}(I_k)$ was less than zero before increasing by one in the current step, then I_k was borrowing capacity from at least one earlier interval. Thus, we have to increase the spare capacities by one if there was borrowing or borrowing propagation in one or several of the intervals from I_c to I_{k-1}.

After execution of a job in the current slot and before the scheduling process continues in the next slot, we have to compare the actual execution time of the job with the WCET for the LO- and the HI-criticality case if the job finished in the current slot. If guaranteed jobs complete earlier than their criticality level specific WCET ($C_k(\text{LO})$ for LO- and $C_k(\text{HI})$ for HI-criticality spare capacities), the difference between actual execution time and specified WCET can be added to the spare capacities of the corresponding intervals, as shown in [IF09].

In conclusion, we can make efficient use of the available resources by the update mechanism presented above. Further, the LO-criticality and HI-criticality spare capacities allow for flexibility to react to the actual job behavior.

Acceptance Test for Aperiodic Jobs

So far, we showed how we schedule TT jobs by introducing (capacity) intervals and spare capacities for LO- and HI-critical behavior of jobs. In the following, we present a method which uses these intervals and spare capacities to integrate ET jobs into the system. To integrate them, we show an acceptance test for firm aperiodic jobs. As we showed before, soft aperiodic jobs are only scheduled if they do not interfere with guaranteed jobs and hence, there is no need for an acceptance test for them.

We present the acceptance test for LO-criticality firm aperiodic jobs. Changes for an acceptance test for HI-criticality jobs are straight forward but having aperiodic jobs which cannot be guaranteed before runtime contradicts the rationale of HI-criticality jobs. A certified HI-criticality job must meet its deadline otherwise catastrophic consequences can happen for the system. Hence, a HI-criticality aperiodic job which may be rejected by an acceptance test is a contradiction.

When an ET job is released in the system, the scheduler checks for ET activities at the beginning of a slot. The acceptance test summarizes the available LO-criticality spare capacities until the deadline of the aperiodic job. (For a HI-criticality acceptance test, we would have to perform an acceptance test based on the LO-criticality and one test based on the HI-criticality spare capacities.) We can group the available spare capacities into three groups: the spare capacities $sc^{\text{LO}}(I_c)$ in the current interval; the spare capacities $sc^{\text{LO}}(I_{full})$ of all full intervals between the current interval and the interval in which the aperiodic job deadline is located; and finally, the spare capacities $sc^{\text{LO}}(I_{dl})$ available between the start of the interval in which the aperiodic job deadline is located and the absolute deadline of the job. The sum of these spare capacities must be greater than or equal to the LO-criticality WCET of the aperiodic job. Inequality (6.8) shows this condition including the calculations of the three spare capacity groups.

$$\underbrace{\max\left(0, sc^{\text{LO}}(I_c)\right)}_{\text{current interval}} + \underbrace{sc^{\text{LO}}(I_{full})}_{\text{full intervals}} + \underbrace{sc^{\text{LO}}(I_{dl})}_{\substack{\text{interval with aperiodic}\\ \text{job deadline}}} \geq C_{\text{A}j}(\text{LO}) \tag{6.8}$$

$$\begin{aligned} \text{with} \quad & sc^{\text{LO}}(I_{full}) = \sum_{c<i\leq l} \max\left(0, sc^{\text{LO}}(I_i)\right) \\ \text{and} \quad & sc^{\text{LO}}(I_{dl}) = \max\left(0, \min\left(sc^{\text{LO}}(I_{l+1}), d_{\text{A}j} - start(I_{l+1})\right)\right) \\ \text{and} \quad & end(I_l) < d_{\text{A}j} \wedge end(I_{l+1}) \geq d_{\text{A}j} \end{aligned}$$

In contrast to original slot-shifting, the spare capacities in the current interval can be negative which is a result of the execution of HI-criticality jobs longer than their specified LO-criticality WCET. As a consequence, the lower bound for the considered spare capacities in the current interval is zero. In the full intervals, negative spare capacities are already included in the spare capacities of the previous interval as consequence of borrowing. Thus, only non-negative spare capacities in these intervals are included. In the interval with the aperiodic job deadline, we can only use the spare capacities until the deadline. As a result, if the spare capacities are larger than time between start of the interval and the job deadline, we can only use this time until the deadline. Further, if the spare capacities in this interval are negative, zero is used as lower bound because the negative spare capacities have been considered when we included borrowing in the calculations.

If the acceptance test fails, then the aperiodic job is rejected. Else, the (LO-criticality) aperiodic job is included into the queue of guaranteed jobs and (LO-criticality) spare capacities are updated. The job is included into the interval whose end is equal to the deadline of the job. If the there is no such interval, the deadline marks the end of an interval and hence, one interval is split into two intervals. The WCET of the job has to be considered in the spare capacities and if the spare is negative, we have to recalculate the spare capacities of intervals affected by borrowing and borrowing propagation.

6.4 Mixed-Criticality Slot-Shifting Example

In this section, we show an example showing that with our presented approach we do not need mode changes for handling HI-criticality behavior of jobs. Further, we illustrate the advantage that we only skip LO-criticality jobs if the actual behavior of HI-criticality jobs requires this. Finally, the acceptance test for ET activities is shown. Table 6.2 depicts the job set and the aperiodic jobs used in this example. The jobs are characterized by the parameters used before. Additionally, we include the actual execution time γ_i which is only known when the job signals its completion at runtime. The release time of the aperiodic job is also unknown until the job enters the system at this time.

First, we determine the (capacity) intervals which we use for the calculation of the spare capacities. We set all job deadlines as an end of an interval. Then, starting from

jobs	criticality level	LO-criticality WCET	HI-criticality WCET	release time	deadline	actual execution time
J_1	LO	4	4	0	7	4
J_2	LO	2	2	1	9	1
J_3	HI	4	8	2	12	5
J_4	HI	1	2	10	15	2
J_{A1}	LO	2	2	9	12	2
J_{A2}	LO	2	2	9	—	2

Table 6.2: *Example job set for mixed-criticality slot-shifting.*

the last interval end, we determine the earliest start times. All intervals have only one job assigned, such that the earliest start time of an interval is equal to the release time of the job assigned to it. For the intervals with jobs J_2, J_3, and J_4, the maximum between earliest start time of the interval and end of the previous interval is always the end of the previous interval. As a result, for these intervals, the start times are the end times of their previous interval. For the interval with J_1, there is no previous interval and hence, the start time is set to the release time of J_1. Figure 6.9 shows the release times and deadlines of the jobs, and the resulting intervals whereas the release times of the jobs are the earliest start times of the intervals. Table 6.3 summarizes the earliest start times, start times, and end times of all intervals.

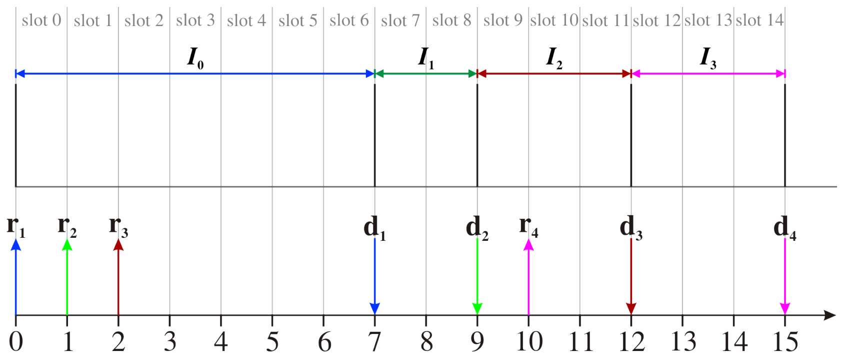


Figure 6.9: *Resulting intervals of the mixed-criticality job set.*

In the next step, we calculate the LO- and HI-criticality spare capacities using Equations (6.6) and (6.7). Figure 6.10 illustrates the calculations by showing the WCETs of all jobs depending on the criticality level of the spare capacities. Scheduling the jobs as late as possible shows the resulting spare capacities with the previously determined intervals. We can see that we cannot execute J_3 for its HI-criticality WCET before its deadline while including J_1 and J_2 at the same time. The selection function of our scheduler schedules the job such that we can always guarantee the execution of HI-criticality

interval	earliest start time	start time	end time
I_0	0	0	7
I_0	1	7	9
I_0	2	9	12
I_0	10	12	15

Table 6.3: *Intervals with their earliest start times, start times, and end times.*

jobs under CAs' pessimistic assumptions. Further, the scheduler can react dynamically on the actual behavior, i.e., the actual execution times, of all jobs.

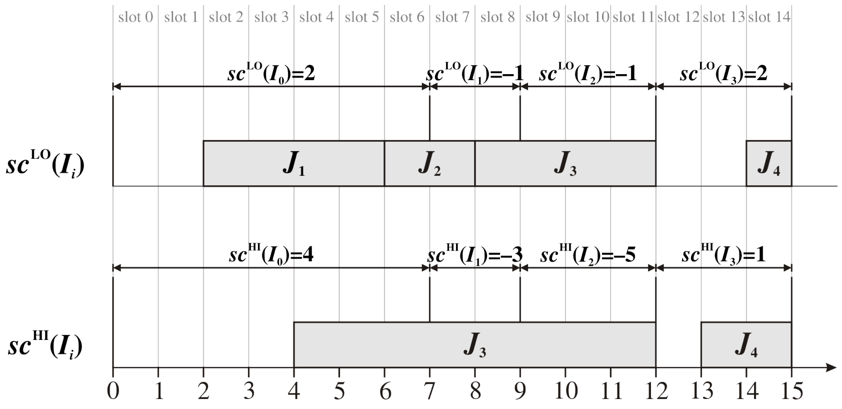


Figure 6.10: LO- *and* HI-*criticalilty spare capacities of the determined intervals.*

After completing the offline calculation of spare capacities, we now start the system and execute the jobs with their actual execution time and react to the ET activities. At the start of the system, there are no soft aperiodic jobs and hence, we select a guaranteed job with earliest deadline, which is J_1. J_1 is assigned to the current interval I_0 and thus, the LO-criticality spare capacities in I_0 are unchanged. Due to the criticality level LO of J_1, the job has not been considered in the HI-criticality spare capacity calculation of $sc^{HI}(I_0)$. As a consequence, $sc^{HI}(I_0)$ is reduced by one slot. This scheduling decision and the spare capacity updates are repeated until slot 3. In slot 3, J_1 signals completion and as a consequence of the spare capacity updates, the HI-criticality spare capacities $sc^{HI}(I_0)$ become zero.$sc^{HI}(I_0)$. In Figure 6.11, we show the execution of J_1 in 4 slots and the resulting spare capacities after slot 3.

The HI-criticality spare capacities in the current interval indicate that we must schedule a HI-criticality job otherwise we cannot guarantee its HI-criticality WCET before its deadline. We select a job with earliest deadline from $R^{HI}(t)$ which is J_3. In slot 4, we schedule J_3 which is assigned to interval I_2. J_2 is a HI-criticality job and executes outside its interval. As a result, we have to update the spare capacities in the following way: in $sc^{LO}(I_0)$, this execution has not been considered and hence, we reduce the spare capacities here by one slot. In the assigned interval of J_3, we increase the spare

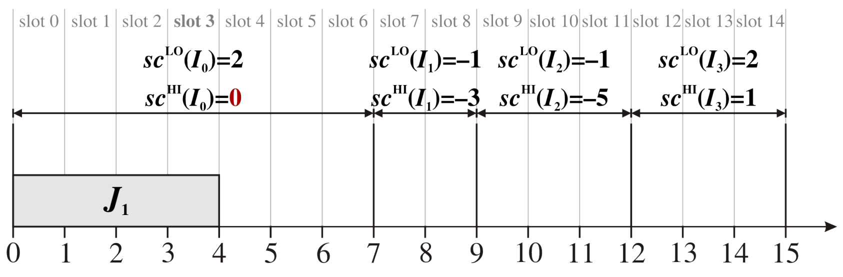


Figure 6.11: *Scheduled jobs and spare capacities after slot 3.*

capacities by one slot because this slot has now been executed earlier already in interval I_0. Due to the borrowing of I_2, indicated by the negative spare capacities before we increased them by one slot, interval I_1 needs to lend one slot less to I_2 and thus, we increase $sc^{\text{LO}}(I_1)$ by one slot. The HI-criticality spare capacities are updated in the same way, but with additionally increasing $sc^{\text{HI}}(I_0)$ by one slot because I_0 is affected by borrowing propagation in the HI-criticality case. In other words, due to the swapping of one slot between I_0 and I_2 by the earlier execution of J_3, I_2 borrows one slot less in I_1 and consequently, I_1 borrows one slot less in I_0. Increasing by one and decreasing by one slot results in unchanged $sc^{\text{HI}}(I_0)$ which is still zero. The resulting spare capacities are shown in Figure 6.12.

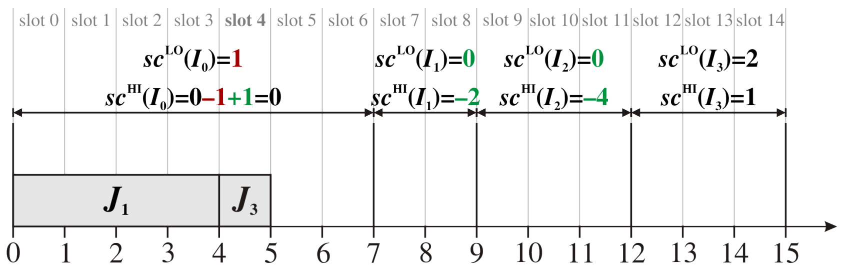


Figure 6.12: *Scheduled jobs and spare capacities after slot 4.*

The HI-criticality spare capacity value $sc^{\text{HI}}(I_0) = 0$ requires that we schedule a HI-criticality job in the next slot. The spare capacity updates are done as in slot 4 except that $sc^{\text{LO}}(I_1)$ is not affected by borrowing anymore which is indicated by the non-negative spare capacity value. As a result, these spare capacities are unchanged. Zero HI-criticality spare capacities require that we schedule a HI-criticality job also in slot 6. As a consequence, in slot 5 and 6, we schedule J_3. The resulting spare capacities are shown in Figure 6.13. The negative LO-criticality spare capacities at the end of the current interval indicate that we executed more slots with workload which has not been assigned to this interval than we have had spare capacities. This is a first indicator that we may have to skip a LO-criticality job if all jobs need to run for their WCET.

At $t = 7$, the HI-criticality spare capacities in the new current interval I_1 are zero. As a consequence, we select a HI-criticality job with earliest deadline again. The earlier execution of one slot of J_3 which has been assigned to interval I_2 swaps one slot between

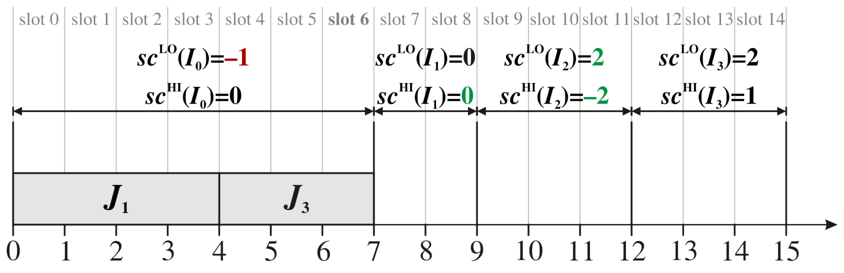


Figure 6.13: *Scheduled jobs and spare capacities after slot 6.*

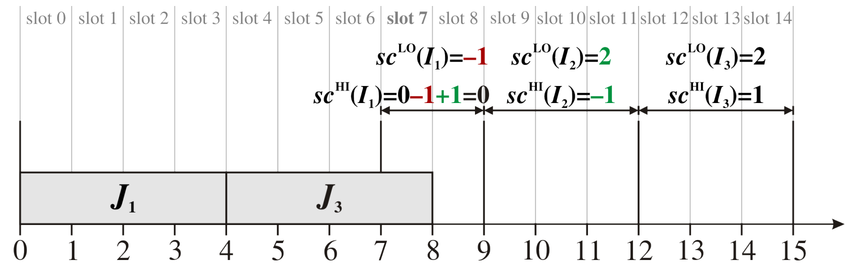
![htb]


Figure 6.14: *Scheduled jobs and spare capacities after slot 7.*

$sc^{\text{LO}}(I_1)$ and $sc^{\text{LO}}(I_2)$. Further, one slot is swapped between $sc^{\text{HI}}(I_1)$ and $sc^{\text{HI}}(I_2)$ and additionally, there is one slot borrowed less in I_1 which results in increasing $sc^{\text{HI}}(I_1)$ by one slot. We show the resulting spare capacities in Figure 6.14.

In slot 8, we have to schedule a HI-criticality job again. The spare capacity updates are as before. In this slot, J_3 completes its execution within 5 slots. In the LO-criticality spare capacities only 4 slots have been considered and thus, we need to update them. On the contrary, in the HI-criticality spare capacities we considered an execution of 8 slots. As a consequence, the spare capacities in the assigned interval I_2 can be increased by the difference of 3 slots. In Figure 6.15, we depict the resulting spare capacities at the end of slot 8.

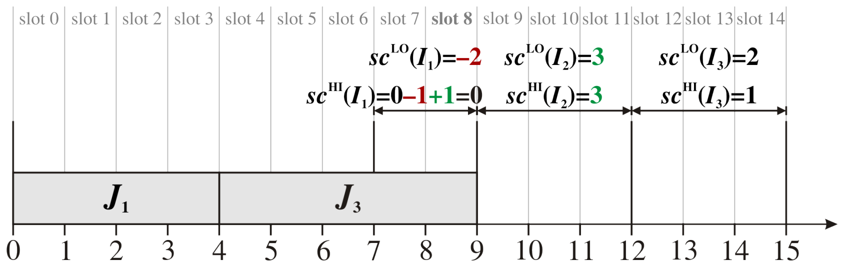


Figure 6.15: *Scheduled jobs and spare capacities after slot 8.*

At $t = 9$, the ET jobs J_{A1} and J_{A2} are released. J_{A2} is a soft aperiodic job and thus, can be directly added to the queue of soft jobs $R^{\text{soft}}(t)$. Firm aperiodic job J_{A1} has LO-criticality and hence, we perform an acceptance test with LO-criticality spare capacities and WCETs. The deadline of this job is equal to the start of the next interval I_2. Based on Inequality (6.8), we test whether the firm aperiodic job can be accepted. In this case, the test is simple: the deadline of the aperiodic job is equal to the start of the next interval. As a consequence, the acceptance test is simplified to: $\max(0, sc^{\text{LO}}(I_c)) \geq C_{\text{A}j}(\text{LO})$. Here, the LO-criticality spare capacities in the current interval are three and the computational demand of the aperiodic job is two, and hence, we can accept the job. By accepting this job, we have to update the LO-criticality spare capacities. The aperiodic job is assigned to the current interval which reduces $sc^{\text{LO}}(I_2)$ by $C_{\text{A}1}(\text{LO})$. The resulting spare capacities are positive, i.e., there is no borrowing. Figure 6.16 depicts the spare capacities after the acceptance of the aperiodic job.

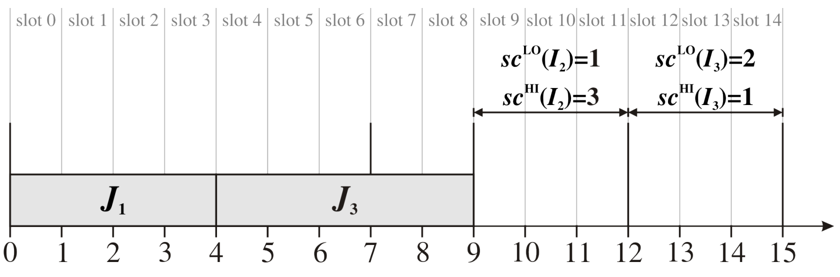


Figure 6.16: *Spare capacities after accepting firm aperiodic job J_{A1}.*

In the current interval I_2, the ready queue for soft aperiodic jobs is not empty anymore. The LO- and HI-criticality spare capacities in the current interval are both greater than zero such that we can schedule a soft aperiodic job without violation of guaranteed firm jobs. As a result, we schedule the soft aperiodic job $J_{\text{A}2}$ in slot 9. Scheduling a soft aperiodic job reduces $sc^{\text{LO}}(I_c)$ and $sc^{\text{HI}}(I_c)$ by one slot. We show the resulting spare capacities and the execution of the jobs so far in Figure 6.17.

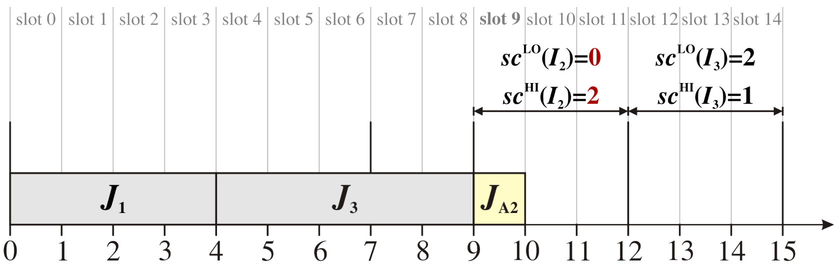


Figure 6.17: *Scheduled jobs and spare capacities after slot 9.*

The LO-criticality spare capacities in the current interval are equal to zero and hence, we have to schedule a guaranteed job. $sc^{\text{HI}}(I_c)$ is greater than zero, i.e., it is possible to select a LO-criticality job. The scheduler selects the job with the earliest deadline in $R^{\text{firm}}(t)$ which is $J_{\text{A}1}$. $J_{\text{A}1}$ has LO-criticality and executes inside its assigned interval such that $sc^{\text{LO}}(I_c)$ is unchanged and $sc^{\text{HI}}(I_c)$ is reduced by one slot. The spare capacities in the current interval still require the selection of a guaranteed job. As a consequence, $J_{\text{A}1}$

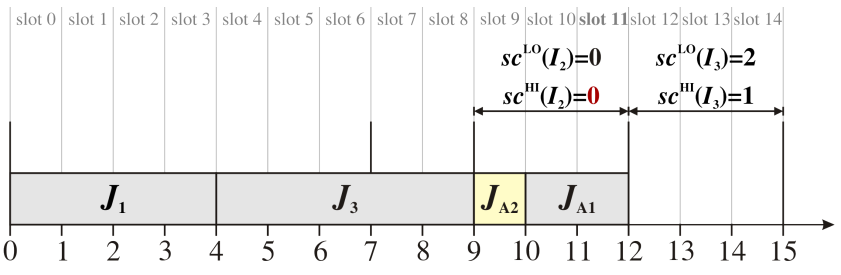


Figure 6.18: *Scheduled jobs and spare capacities after slot 11.*

is scheduled for another slot and $sc^{\text{HI}}(I_c)$ is reduced by one slot again. In Figure 6.18, we show the resulting spare capacities after slot 11.

At $t = 12$, the current interval is now interval I_3. Spare capacities $sc^{\text{LO}}(I_c)$ and $sc^{\text{HI}}(I_c)$ are greater than zero which allows for the execution of a soft aperiodic job. We schedule J_{A2} which reduces the LO- and HI-criticality spare capacities in the current interval by one slot. We show the resulting spare capacities in Figure 6.19.

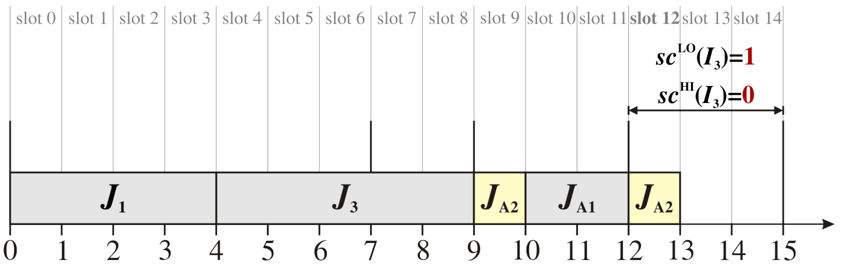


Figure 6.19: *Scheduled jobs and spare capacities after slot 12.*

As a consequence of the soft aperiodic job execution, the HI-criticality spare capacities in the current interval indicate that we have to schedule a HI-criticality job to guarantee the HI-criticality WCET. This results in scheduling J_4 in slot 13 which is assigned to the current interval and hence, the execution has already been considered in both $sc^{\text{LO}}(I_c)$ and $sc^{\text{HI}}(I_c)$. Thus, $sc^{\text{LO}}(I_c)$ and $sc^{\text{HI}}(I_c)$ remain unchanged which is shown in Figure 6.20.

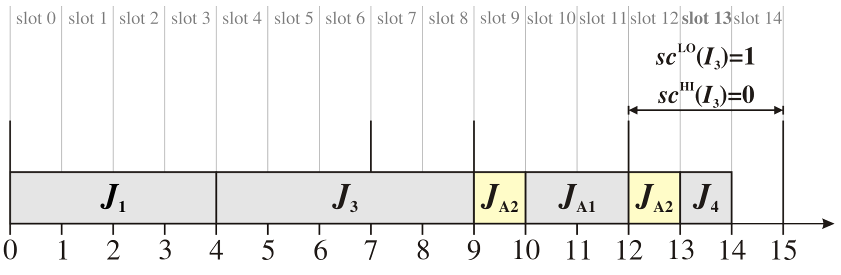


Figure 6.20: *Scheduled jobs and spare capacities after slot 13.*

Job J_4 executed for its LO-criticality WCET but did not signal completion. Further, the HI-criticality spare capacities indicate that we have to schedule a HI-criticality job.

We schedule J_4 in slot 14 which signals completion after its execution in this slot. The execution for more than $C_4(\text{LO})$ has not been considered in the LO-criticality spare capacities and thus, we reduce $sc^{\text{LO}}(I_c)$ by one slot. In the HI-criticality spare capacities, the execution has been considered such that they are unchanged. This completes the execution of the given job set and Figure 6.21 depicts the resulting schedule with the spare capacities after the last slot.

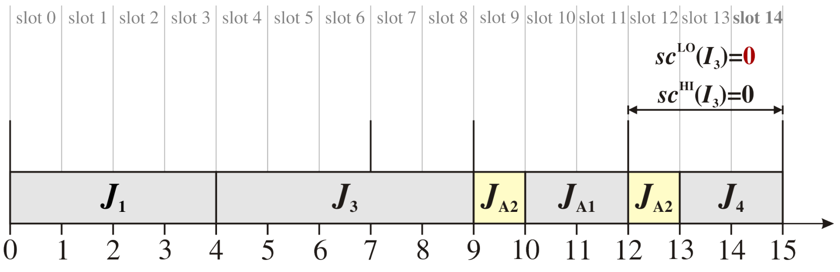


Figure 6.21: *Scheduled jobs and spare capacities after slot 14.*

In the presented example, we showed that with our method we can react to the actual behavior of mixed-criticality jobs. Due to the HI-criticality behavior of job J_3, we had to skip LO-criticality job J_2. On the contrary, J_3 completed earlier than its HI-criticality WCET such that we could make use of the available capacities to guarantee the firm aperiodic job J_{A1} and also execute the soft aperiodic job J_{A2}.

6.5 Discussion: How Can We Dynamically Use Resources at Runtime?

In the following, we answer the question on how we can make use of the knowledge about the available resources at runtime. Our approach is based on an schedule table with earliest start times (release times), deadlines, and WCETs of jobs. To answer the question, we can consider two versions of schedule tables: certified and non-certified schedule tables.

First, we assume that the starting point is a certified schedule table of a legacy system. As a result, the pessimistic assumptions of the CAs are already included in the schedule table. Using our presented approach with such tables can flexibly make use of the available resources. We include ET activities into the scheduling process at runtime without interference of certified system behavior. Further, the knowledge of actually available resources at runtime can be used to improve quality of result. As an example, we consider applications that improve their results when executing for longer times, e.g., quality of result is improved when more iterations of a calculation are performed. In order to achieve this goal, we can implement this desired behavior as follows: a job provides the required quality of result within its specified WCET. If the job completes its execution before its deadline, we can use the available resources to create an aperiodic job with deadline equal to the original job deadline to improve the quality of result by executing the job for some additional time. As a consequence, we can guarantee a "minimum" quality of the results while making use of the available resources to further improve the quality of results.

In contrast to the certified schedule tables, we can begin with a designer based schedule table excluding the pessimistic assumptions of the CAs. The schedule table is constructed based on the LO-criticality WCETs and hence, it is not certified. Release times and deadlines of HI-criticality jobs are not changed by CAs. As a result, capacity intervals are not influenced by CAs' pessimistic assumptions. On the contrary, WCETs change for HI-criticality jobs. This results in changing spare capacities for HI-criticality system state. We can use the HI-criticality spare capacities to prove correct system behavior even in case all HI-criticality jobs show HI-behavior because LO-criticality jobs do not interfere them. The available resources, i.e., spare capacities, can be used for ET activities based on the actual behavior of jobs instead of worst-case estimations.

Finally, the knowledge of actually available resources can be used for LO-criticality jobs at runtime. When HI-criticality jobs exceeds their LO-criticality WCET, it is possible that LO-criticality jobs cannot execute before their deadline and thus, are skipped. If results of such a LO-criticality job are useless after their deadline, then the system performance will degrade. On the contrary, if the affected LO-criticality jobs allow for some tardiness, we can use the knowledge of available resources to try to avoid this performance degradation. If a delayed execution of a LO-criticality job results in worse result but results which can still be used to improve the performance of the system, then we can create a new (LO-criticality) aperiodic job to execute this job. The skipped LO-criticality job is re-released as an aperiodic job with an extended deadline. Although the execution of the job is delayed, its execution can still contribute to the system performance. As a result, a temporary higher workload by HI-criticality jobs can lead to skipping LO-criticality jobs. Despite this, jobs allowing some tardiness can execute with available resources when the high workload is decreasing again. The presented acceptance test guarantees that by re-releasing this job, the timing constraints of other guaranteed jobs are not violated.

Chapter 7

Slot-Shifting with Generic Mode Changes

In this chapter, we present two approaches to use mode changes with slot-shifting. On the one hand, we show slot-shifting with generic mode changes. This method schedules jobs with varying parameters in different modes. Furthermore, we can determine the feasibility of mode changes and perform them only if the active jobs in the destination mode can meet their deadlines. On the other hand, we adapt slot-shifting with generic mode changes to accommodate the requirements of mixed-criticality systems. In contrast to the generic mode change case, we need to change modes to guarantee timing constraints of jobs with high criticality levels. As a result, the method does not check the feasibility of mode changes but triggers mode changes such that a timing behavior of high criticality jobs can be guaranteed.

In Section 7.1, we show the limitations of the Vestal mixed-criticality model and how we can refine the term "criticality". Next, Section 7.2 presents important aspects of mode changes with mixed-criticality and non-mixed-criticality jobs. The used job model for our algorithms is shown in Section 7.3. In Section 7.4, we explain slot-shifting with generic mode changes and illustrate its behavior with an example in Section 7.5. The application of slot-shifting with generic mode changes to mixed-criticality systems is shown in Section 7.6. Finally, a discussion about the mixed-criticality model in Section 7.7 concludes the chapter.

7.1 Motivation

In the previous chapter, we showed a slot-shifting-based approach to handle dual-criticality jobs with two worst-case execution times (WCETs). In this chapter, we generalize this approach to increase the flexibility in the mixed-criticality job parameters. We assume a system defined by several modes with different jobs and variable job parameters. As a result, the job set is defined by jobs which are executed only in specific modes. Furthermore, job parameters can change depending on the mode. The exact assumptions about job parameters are shown in Section 7.3.

As Graydon and Bate suggested, the term criticality of a job or task should be considered more specifically [GB13]. The "criticality" of systems and their jobs should be defined by the terms *confidence*, *importance*, and *mode*. In the Vestal model [Ves07], the criticality of a job is reflected by the confidence in the WCET bounds. As a consequence, larger WCET values correspond to a higher confidence in this bound. On the contrary, the importance is not considered in this model. An often used assumption is that LO-criticality jobs do not need to execute when the system is in HI-criticality state. This assumption refers to the mode of the scheduler. Burns and Baruah generalized the Vestal model by extending the constant periods and deadlines of periodic tasks to periodic tasks which allow for shortened periods and shortened deadlines of tasks in a higher criticality state [BB11].

In this chapter, we consider two "versions" of mode changes. On the one hand, we use mode changes as consequence of a HI-criticality job exceeding its LO-criticality WCET, as shown in Chapter 5. In this case, we must perform the mode change to guarantee a specific behavior at the cost of the LO-criticality jobs. On the other hand, a mode change can be performed to switch into a different operational mode. Here, the mode change cannot be performed at every instant. An example to illustrate different operational modes is switching from take-off mode to flying mode of an airplane. For instance, during take-off phase, an airplane needs its landing gear on the runway. At take-off completion, the landing gear is retracted and the airplane can switch to flying mode. As a consequence, switching from starting to flying is only possible if the job "retract landing gear" is completed. Thus, a mode change request can be declined until specific conditions allow for the mode change.

In our method, we show the conditions that allow for a mode change to a specific destination mode. Afterwards, we present how we can adapt the method to accommodate for the requirements of mixed-criticality systems. In other words, the needed adaptations when a mode has to be changed to guarantee a specified behavior, i.e., to guarantee HI-criticality jobs with the HI-criticality WCETs of the certification authorities (CAs). Finally, we can include event-triggered (ET) activities into the scheduling process with our method.

7.2 Generic and Mixed-Criticality Mode Changes

In Chapter 5, LO-criticality jobs have been aborted when we switched from LO-criticality to HI-criticality mode. In Chapter 6, an HI-criticality job exceeding its LO-criticality

WCET, resulted in skipped LO-criticality jobs as long as there are not enough resources to schedule LO-criticality jobs and guarantee CAs' requirements of HI-criticality jobs.

7.2.1 General Job Behavior during Mode Changes

In this chapter, we show an approach which allows for different WCETs, different execution windows, and different jobs in each mode. In the following, we present the consequences for ready jobs when a mode change request (MCR) occurs.

The original Vestal model is based on periodic tasks. Burns and Baruah showed an extension of this mixed-criticality task model [BB11]. In their extended model, the periodic mixed-criticality tasks are characterized by criticality level, periods, relative deadlines, and WCETs. As in Vestal's model, there is one WCET for each criticality level. In addition, for criticality levels higher than the task's criticality level, the WCET is not defined or defined as the same value as for the highest used criticality level. Furthermore, relative deadlines can also change: for a higher criticality level, the task's deadline is allowed to be shorter than the original deadline at its lowest criticality level. Finally, the periods can also be shortened if the task is executed in a higher criticality level.

Although our method works on job-base, we show the different behaviors based on periodic tasks during the mode changes for reasons of comparability. This does not influence the applicability of our method.

In the following, mode changes are performed from *source* mode to *destination* mode. We show different types of periodic tasks before and after a mode change.

The current instance of a periodic task can be canceled at mode change when this task is not active in the destination mode. As a result, the task is removed from ready queue and not scheduled anymore.

On the contrary, there are tasks which are only active in the source mode and must complete the execution of the currently active instance. Our approach works without a transition mode and hence, this task is not included in the destination mode and thus, will not be scheduled. We perform mode changes based on the information in the destination mode. In other words, when we switch to the destination mode, we guarantee tasks in that mode and assume that tasks in the source mode are not needed anymore. Despite this, we can complete the execution of ready source mode jobs by re-releasing them as an aperiodic job such that this job can complete its execution. In this case, the job can continue its execution with additional execution time granted for the re-released aperiodic job.

Besides tasks that are only active in the source mode, tasks that are active in both modes are also possible. These tasks can continue with their release pattern or change it. In both cases, only the task instance which is active during the mode change is important for a job-based scheduler. Due to being active in both source and destination mode, the currently active instance should not be aborted. In the destination mode, the currently active job is also considered and if there is enough time available, we can switch to the destination mode and complete its execution; otherwise, the MCR is denied.

Finally, there are tasks which are only active in the destination mode. Thus, their execution is considered in the destination mode. If the MCR is accepted, these tasks can be executed.

Our approach is based on slot-shifting. We have shown the original slot-shifting approach in Section 6.2. For each mode, we determine capacity intervals and spare capacities based on the jobs active in this mode with the job parameters for this mode. While we schedule jobs in the current mode, we update the spare capacities of all modes. When a mode change is requested, we perform the mode change only if we can guarantee the execution of all jobs of the destination mode with deadline later than the current time instance. If a job is active in both source and destination mode, the already granted execution time is accounted for the execution in the destination mode. It is possible that a job has a different deadline and/or WCET in the destination mode and we perform the mode change only if the new parameters can be guaranteed.

7.2.2 Mixed-Criticality Mode Changes

In the following, we show the impact of the requirements of mixed-criticality systems on job parameters and modes. The event which triggers a mode change is the overrun of the LO-criticality WCET of a guaranteed HI-criticality job. Hence, mixed-criticality systems focus on the guarantee of the HI-criticality jobs. In this chapter, we also show how we can adapt our approach with generic mode changes to guarantee the requirements of HI-criticality jobs. As a consequence, an MCR is triggered by the need for a guarantee of the job parameters in the HI-criticality mode and hence, the MCR is never denied.

In contrast to the generic mode change scheduler, there is an order of modes which is based on the criticality levels. In a dual-criticality system, we begin in criticality level LO and we can switch to the higher criticality level HI. In a system with more than two criticality levels, we sort the mode according to their ascending criticality level. The goal is to guarantee jobs with the highest criticality level at cost of jobs with lower criticality level.

As a consequence of the previous considerations, the question “How many criticality levels are reasonable?” arises. The rational for two criticality levels is intuitive: one criticality level for certified jobs and one criticality level for non-certified jobs. Certified jobs have to be guaranteed under all circumstances under CAs’ pessimistic parameters. For more criticality levels, the reason for different parameters is not that intuitive and the term criticality is too general. As a consequence, we discuss the question of how many criticality levels we need in more detail using the suggested terms of Graydon and Bate [GB13].

First, the *confidence* in the WCET bound does not justify having many criticality levels. There are not that many completely different WCET analysis methods to justify many different WCETs.

Second, having many different *modes* in a system is reasonable but this does not necessarily result in having different parameters for the same jobs. Using different modes during different operational phases of a system result in different jobs that have to be performed depending on the situation. We can perform these mode changes with our generic mode change scheduler based on slot-shifting. Jobs in a source mode can trigger an MCR and the mode change is performed if the requirements, i.e., the timing constraints, of the destination mode can be fulfilled. In other words, we stay in the feasible source mode until it is safe (feasible) to switch to the destination mode.

Finally, the *importance* of jobs can justify having several or many criticality levels. It is possible to group jobs into many classes with different importances. Despite this, different importances do not result in different job parameters. As a consequence, a refined viewpoint is needed. The system developers have to define the consequences of a higher importance of a job. Possible optimization goals are, e.g., response times, latency, jitter, and many other characteristics of the jobs. Thus, depending on the target applications, the scheduling problem for these cases has to be refined.

In the following, we show a method to handle generic mode changes. For each mode, a specified set of jobs is scheduled and jobs can be scheduled in several modes. The feasibility of an MCR is determined by the guarantee to successfully schedule the jobs in the destination mode. Further, we show how we can adapt the approach to schedule mixed-criticality jobs where jobs with the highest criticality level are guaranteed with the WCET of their highest criticality level. By doing this, we can handle mixed-criticality jobs with several WCETs and changing release times and deadlines, at the cost of very pessimistic scheduling decisions.

7.3 Job Model

In Sections 7.4 and 7.5, we will schedule jobs only in modes if the job is active in that mode, i.e., the job has a defined WCET for that mode. For reasons of simplicity, we assume that a job has the same release time and deadline in all modes which does not restrict the applicability and the behavior of our methods. A job with a different deadline in another mode, only influences the assigned interval and in which interval the spare capacities are updated. The way in which spare capacities are updated is not influenced by this different deadlines.

The jobs are characterized by the tuple $J_i = \langle C_i(1), ..., C_i(k), r_i, d_i \rangle$ with:

- $C_i(j)$: WCET in mode j;
- r_i: release time in modes where the job is active;
- d_i: deadline in modes where the job is active;

If a job is inactive in a mode, the WCET for this mode is not defined. For instance, job $J_i = \langle 3, -, 2, 0, 10 \rangle$ is only active in mode 1 and 3 with release time 0 and deadline 10.

In Section 7.6, we use the Vestal job model with k criticality levels whereas 1 is the lowest criticality level. The WCET of job J_i for criticality level j is represented by $C_i(j)$ whereas the WCET for a higher criticality level is not smaller than the WCET of the next lower criticality level, i.e., the WCETs are monotonically increasing.

7.4 Slot-Shifting with Generic Mode Changes

We showed the original slot-shifting algorithm in Section 6.2. In Section 6.3, we showed the mixed-criticality slot-shifting algorithm without mode changes. In the following, we show a scheduler based on slot-shifting to handle generic mode changes. The scheduler consists of an offline and an online phase.

7.4.1 Offline Phase

For each mode, we determine intervals and spare capacities based on the jobs active in that mode. The process of interval determination and spare capacity calculation is shown in Section 6.2 in general and in Section 6.3 in the dual-criticality case. In the following, we briefly recapitulate the procedures and show the symbols for the generic mode change case.

We divide the schedule of each mode m into disjoint (capacity) intervals I_i^m. In each mode m, each deadline of a job, which is active in this mode, marks the end $end(I_i^j)$ of an interval. Jobs active in mode m are assigned to the intervals with the corresponding deadline. The earliest start time $est\left(I_i^m\right)$ of an interval is determined by the minimum of all release times of jobs assigned to this interval in this mode:

$$est(I_i^m) = \underset{\forall J_k \in I_i^m}{min} \left(r_k\right) \tag{7.1}$$

The start of an interval in a mode m is determined by the maximum of its earliest start time and the end of the previous interval in that mode:

$$start(I_i^m) = max\left(est(I_i^m), end(I_{i-1}^m)\right) \tag{7.2}$$

Furthermore, consecutive slots between these intervals are defined as empty intervals. Finally, the length $|I_i^m|$ of an interval I_i^m is defined by the difference in the number of slots between start and end of the interval.

Spare capacities reflect the number of empty slots within an interval. We can calculate the spare capacities by the length of the interval, the assigned WCETs of jobs in this interval, and the slots lent to later intervals because of borrowing[1]. Equation (7.3) shows the calculation of the spare capacities for an interval I_i^m in mode m.

$$sc\left(I_i^m\right) = |I_i^m| - \sum_{J_k \in I_i^m} C_k(m) + min\left(sc\left(I_{i+1}^m\right), 0\right) \tag{7.3}$$

In summary, the offline phase is quite similar to the original slot-shifting algorithm. At runtime, intervals and spare capacities are used to schedule jobs and check the feasibility of MCRs.

7.4.2 Online Phase

In the following, we show how we schedule jobs and update the spare capacities at runtime. Additionally to slot-shifting without mode changes, as shown in Chapter 6, we have to check the feasibility of MCRs.

[1]The effect of borrowing is described in Section 6.2.1 on page 108.

Correctness of Mode Changes

Based on a system configuration, a specific sequence of possible mode changes is given. This sequence is based on the behavior of the system. As an example, a mode change from *landing* to *flying* should not be possible without the mode *take-off* in between. As an example, Figure 7.1 shows a deterministic finite automaton defining the possible sequences of mode changes. For instance, in mode 4, the system can switch to modes 5 and 7 based on the specified events. In this automaton, it is also possible to switch to the current mode, i.e., we do not change the mode. For reasons of clarity, we neglect to show the corresponding transitions from one mode to itself. In contrast to the system view,

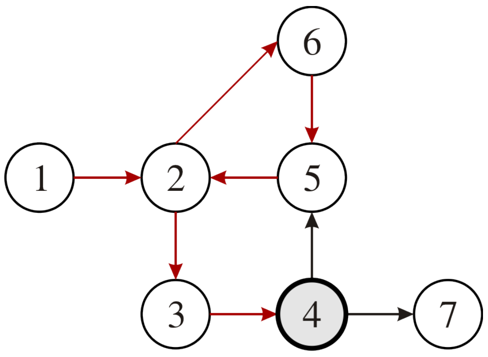


Figure 7.1: *System view of mode changes.*

our scheduler checks the feasibility of a mode change considering timing guarantees of the executing jobs of the destination mode. Figure 7.2 shows the same modes as Figure 7.1 from a scheduling point of view. Considering only the feasibility of scheduling, i.e., ignoring the automaton showing the system view of mode changes, we can switch to a mode if all jobs with deadline later than the current time instant can meet their deadline. In the shown example, we can switch from mode 4 to modes 2, 3, and 5. We assume that jobs request only mode changes to modes which result in a valid behavior of the system, i.e., a behavior which is compliant with the system view of the mode changes.

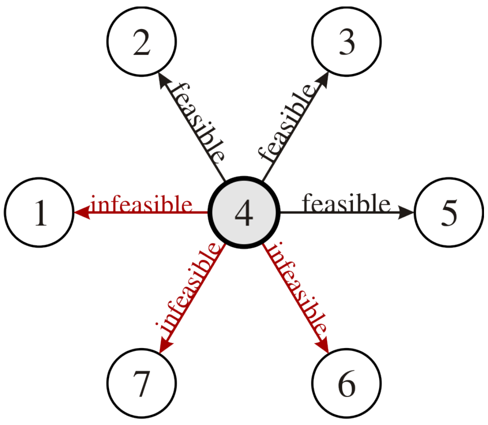


Figure 7.2: *Scheduling view of mode changes.*

Selection Function

Based on the spare capacities and the deadlines of jobs, we select the next executing job. We use two ready queues for each mode m: $R^{m|\text{firm}}(t)$ contains all guaranteed ready jobs in the current mode and $R^{m|\text{soft}}(t)$ contains all soft aperiodic jobs which are active in the current mode. The possible decision cases at time t are shown in the following:

a) $R^{m|\text{firm}}(t) = \{\}$:

- **1)** $R^{m|\text{soft}}(t) = \{\}$:
  We schedule an idle slot because there are no ready jobs.
- **2)** $R^{m|\text{soft}}(t) \neq \{\}$:
  The spare capacities $sc(I_c^m)$ are always greater than zero and hence, we can schedule a soft aperiodic job.

b) $R^{m|\text{firm}}(t) \neq \{\}$:

- **1)** $sc(I_c^m) > 0$:
  If $R^{m|\text{soft}}(t) \neq \{\}$ then we schedule the first soft aperiodic job in the FIFO-queue $R^{m|\text{soft}}(t)$. Else, we select the ready job with the earliest deadline in $R^{m|\text{firm}}(t)$. Only in this situation, we can schedule soft aperiodic jobs if guaranteed jobs are ready to execute.
- **2)** $sc(I_c^m) = 0$:
  We select the job with the earliest deadline in $R^{m|\text{firm}}(t)$ because there are no available resources for soft aperiodic jobs.
- **3)** $sc(I_c^m) < 0$:
  In the current interval, the spare capacities of the current mode cannot be less than zero.

In summary, the selection of the next execution job is simple. Here, we only have to consider jobs in the current mode. If we have to fulfill job timing requirements of other modes, the selection function becomes more complex, which we will show in Section 7.6.

Acceptance of Mode Change Requests

The decision about the acceptance of an MCR is taken at the beginning of a slot when the scheduler is active. The scheduler checks whether the MCR can be accepted or is denied, and then selects the next executing job and updates the spare capacities. Whether an MCR from current (source) mode *src* to the destination mode *dest* is feasible depends on the spare capacities in the destination mode. It is important to highlight that intervals can be different in the destination mode. We repeat that the interval border, i.e., the point in time when an interval ends and the next interval starts, belongs to the interval which is starting at that time instant. The scheduler accepts an MCR if and only if the spare capacities in the current interval in the destination mode are non-negative, i.e., $sc\left(I_c^{dest}\right) \geq 0$. If these spare capacities are negative, at least one job of the current or of a later interval cannot meet its deadline. Spare capacities in the current interval in another mode can be negative because we can execute jobs which are not active in that mode and hence, are not considered in the spare capacity calculations. It is sufficient to check the spare capacities of the current interval because WCETs in later intervals exceeding the length of the interval, are considered by borrowing. As long as the spare capacities in the current interval in the destination mode are negative, we cannot change the mode to *dest*. Negative spare capacities in the current interval of a mode can become positive if a job finishes earlier than its WCET in the current interval of the source mode, i.e., the current mode. If a denied MCR is performed as soon as it is feasible or if the denied mode change is not performed at all depends on the system configuration. Both approaches do not influence our scheduler and the only difference is whether the scheduler checks the feasibility of a denied MCR in the next slot(s) again or not.

Spare Capacity Maintenance

Based on the executed job, we update the spare capacities in all modes. We assume that the performed execution of a job in the current mode is also valid in another mode where the job is active. For instance, a job calculating the current position of a plane can start its execution in *take-off* mode and complete its execution in *flying* mode and the results are valid. In this case, there is no reason to restart the job execution in case of a mode change. On the contrary, if it should be necessary to restart a job with different parameters in a new mode, then this can be solved by simply using a copy of the job with a different identifier in the destination mode and thus, performing the same calculations but with different input parameters again. The spare capacities are updated according to the assigned interval and the modes in which the executed job is active.
No execution: If an idle slot has been scheduled, we decrease the spare capacities of the current interval in all modes by one slot.
Soft aperiodic execution: As the workload of soft aperiodic jobs is not included in the spare capacity calculations, we decrease the spare capacities of the current interval in all modes by one slot.
Guaranteed job execution: If a guaranteed job J_i, either time-triggered (TT) or firm ET (in this case: $J_{\mathrm{A}i}$), has been scheduled, then we have to differentiate to which interval the job is assigned and in which modes the job is active. The spare capacities have to be updated in all modes m. Due to different WCETs in the modes, it is possible that a job executes for longer than the WCET of another mode. We highlight that a job which is assigned to the current interval of the current mode, is not necessarily assigned to the current interval in another mode. Additional jobs in other modes can create additional intervals in the corresponding modes.

A) J_i is **not active** in mode m:

The execution of J_i does not contribute to any job active in this mode and thus, is not considered in the spare capacities of this mode. As a result, the spare capacities in the current interval are decreased by one slot.

B) J_i is **active** in mode m and $J_i \in I_c^m$:

1. J_i executed for **less** than $C_i(m)$:
   The scheduled job has been considered in the current interval of this mode and hence, the spare capacities are not changed.
2. J_i executed for **more** than $C_i(m)$:
   The job executed for more than the mode specific WCET. Hence, the execution has not been considered in the spare capacity calculations and we decrease $sc(I_c^m)$ by one slot.

C) J_i is **active** in mode m and $J_i \in I_k^m$ with $I_k^m \neq I_c^m$ and $k > c$:

1. J_i executed for **less** than $C_i(m)$:
   The scheduled demand has not been included in the spare capacity calculations of the current interval I_c^m. As a consequence, we decrease (I_c^m) by one slot. Additionally, we increase $sc(I_k^m)$ where the demand has been originally considered in the spare capacity calculations. In other words, one slot of execution is swapped between the current interval I_c^m and the assigned job interval I_k^m.

   Furthermore, we have to consider borrowing: if $sc(I_k^m)$ was less than zero before increasing by one slot in the current step, then I_k^m was borrowing capacity from at least one earlier interval. As a consequence, we have to increase the spare capacities by one slot if there was borrowing or borrowing propagation in one or several of the intervals from I_c^m to I_{k-1}^m.
2. J_i executed for **more** than $C_i(m)$:
   The job executed for more than the WCET specified in this mode. Hence, the execution has not been considered in the spare capacity calculations and we decrease $sc(I_c^m)$ by one slot.

If guaranteed jobs complete earlier than their specified WCET in a mode, then the difference between actual execution time and specified WCET can be added to the spare capacities in the assigned interval of the corresponding modes. Further intervals in that mode which are affected by borrowing also have to be updated.

Acceptance Test

The acceptance test in slot-shifting with generic mode changes is similar to the test of the original slot-shifting algorithm. The test checks whether firm aperiodic jobs can complete their execution within their execution window without violating timing constraints of already guaranteed TT and ET jobs. It has to summarize the available spare capacities until the aperiodic job's deadline in a mode m where the job is active. As in original slot-shifting, spare capacities can be grouped into three groups: the spare capacities $sc(I_c^m)$ in the current interval; the spare capacities $sc(I_{full}^m)$ of all full intervals in between the current interval and the interval in which the aperiodic job deadline is located; and the spare capacities $sc(I_{dl}^m)$ available between the start of the interval in which the aperiodic job deadline is located and the absolute deadline of the job. Inequality (7.4) shows the condition that the sum of these three spare capacity groups have to be at least as large as the WCET of the aperiodic job.

$$\underbrace{\max(0, sc(I_c^m))}_{\text{current interval}} + \underbrace{sc(I_{full}^m)}_{\text{full intervals}} + \underbrace{sc(I_{dl}^m)}_{\substack{\text{interval with aperiodic}\\ \text{job deadline}}} \geq C_{\mathrm{A}j}(m) \tag{7.4}$$

$$\begin{aligned} \text{with} \quad & sc(I_{full}^m) = \sum_{c<i\leq l} \max(0, sc(I_i^m)) \\ \text{and} \quad & sc(I_{dl}^m) = \max\left(0, \min\left(sc(I_{l+1}^m), d_{\mathrm{A}j} - start(I_{l+1}^m)\right)\right) \\ \text{and} \quad & end(I_l^m) < d_{\mathrm{A}j} \wedge end(I_{l+1}^m) \geq d_{\mathrm{A}j} \end{aligned}$$

If the acceptance test fails, i.e., in at least one mode where the job is active the spare capacities are less than the corresponding WCET, the aperiodic job is rejected. Else, the aperiodic job is included into the queues of guaranteed jobs in the affected modes and spare capacities are updated in all modes where the job is active.

7.5 Generic Mode Changes Example

In this section, we present an example to illustrate how we schedule jobs and update the spare capacities in each mode. Further, we show how the spare capacities are used to determine safe time instants to switch to another mode, i.e., time instants at which an MCR is accepted. Table 7.1 depicts the used job set. The job set consists of one job active in all mode with different WCETs and three jobs active only in a subset of the modes. If there is no WCET defined for a job in a mode, then this job is not active in the corresponding mode. In this example, we focus on the feasibility of possible mode changes and thus, we do not include ET activities. We assume that all jobs execute for the entire WCET defined in the currently active mode.

jobs	WCET in mode 1	WCET in mode 2	WCET in mode 3	release time	deadline
J_1	3	—	1	0	4
J_2	—	2	—	0	8
J_3	2	3	5	0	8
J_4	—	3	2	0	10

Table 7.1: *Example job set for slot-shifting with generic mode changes.*

We present the sequence of scheduling decisions for the execution of mode 1. The resulting schedules and the feasibility of mode changes executing mode 2 or mode 3 are shown in the appendix in Section A.3. The spare capacity values of a slot refer to the available resources after at the end of this slot.

As discussed before, we determine the intervals and spare capacities in each mode. Figure 7.3 shows the resulting intervals and spare capacities for mode 1, 2, and 3. In modes 1 and 3, the job parameters create three intervals each. In mode 2, only two intervals are created. The current interval I_c^m in mode m is the interval I_i^m with $start(I_i^m) \leq t_c < end(I_i^m)$ with t_c as the current time instance. In other words, the time instance when one intervals ends and the next interval start belongs to the next starting interval. In all three modes, the spare capacities of the first interval are positive. As a result, it is feasible to start the system in each of the modes. Here, we assume that we schedule mode 1 without an MCR.

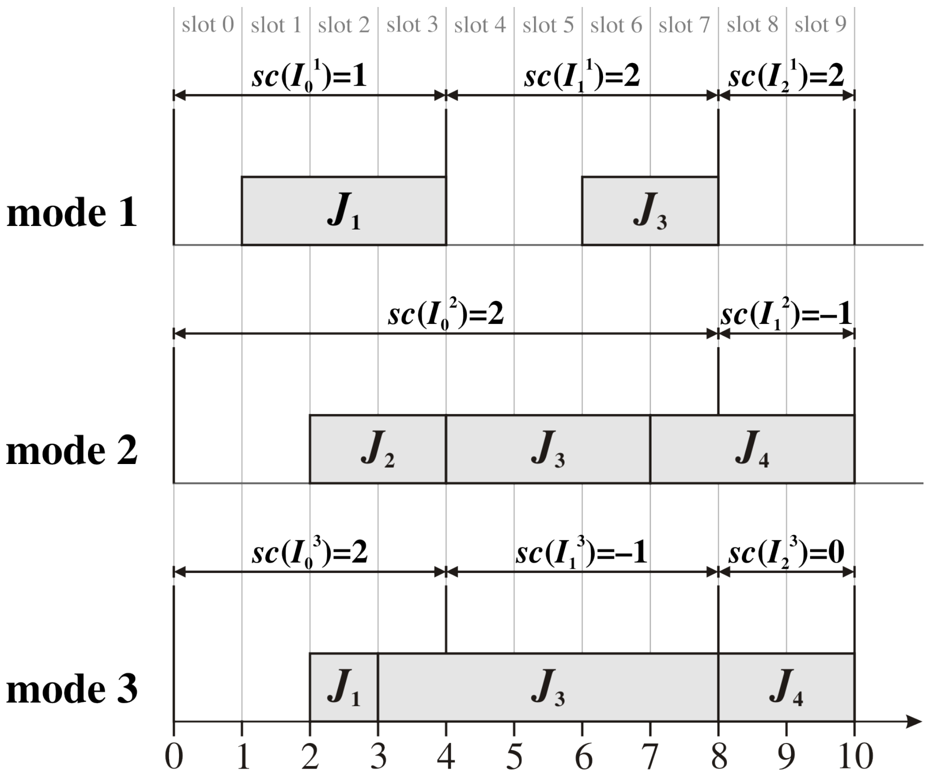


Figure 7.3: *Spare capacities in all three modes.*

In the following, we show the scheduling decisions for each slot. Below the schedule, we depict the spare capacity values for all three modes. Spare capacities which are not affected by the last scheduling decision are grayed out.

In slot 0, we schedule J_1 which has the earliest deadline of jobs active in mode 1. The job is active in modes 1 and 3, and is assigned to the current interval in both modes. As a result, the spare capacities $sc(I_0^1)$ and $sc(I_0^3)$ are unchanged. In mode 2, the job is not active and hence, its execution is not considered in any spare capacity value of this mode. As a consequence, we reduce the spare capacities in the current interval in

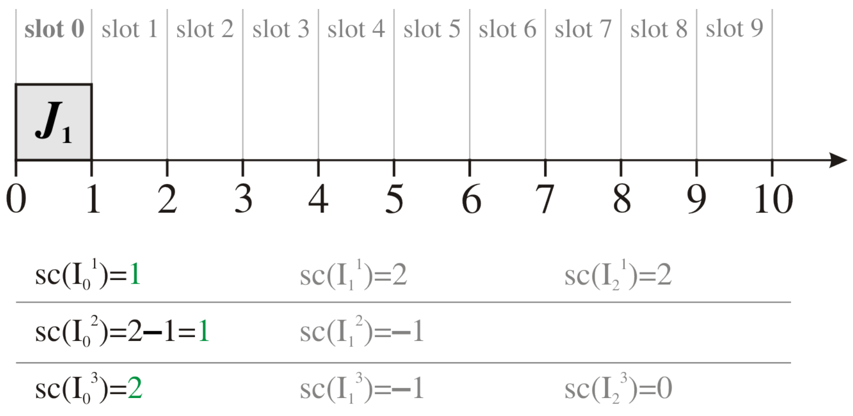


Figure 7.4: *Spare capacities after slot 0.*

this mode, $sc(I_0^2)$, by one slot. Figure 7.4, shows the scheduled job in slot 0 and the resulting spare capacities.

In the next step, J_1 still is the job with the earliest deadline and hence, is scheduled in slot 1. In the current mode, the spare capacities do not changed for the same reason as mention before. In mode 2, J_1 is not active and thus, we decrease $sc(I_0^2)$ by one slot again. Although, J_1 is active in mode 3, we must decrease $sc(I_0^3)$ because the WCET defined in mode 3 is only 1. As a result, the job executes for longer than the specified WCET of that mode and this has not been considered in the spare capacity calaculations. We show the schedule after slot 1 and the corresponding spare capacity values in Figure 7.5.

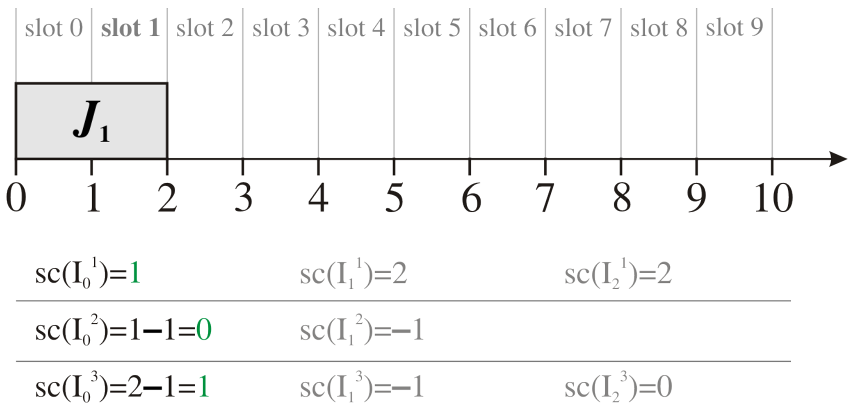
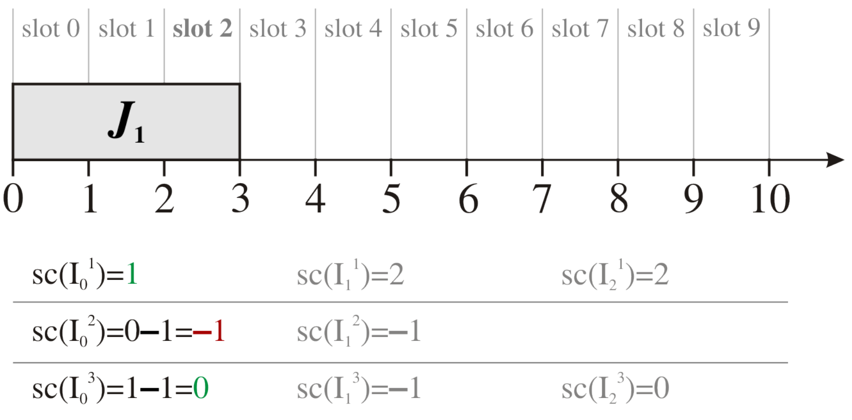


Figure 7.5: *Spare capacities after slot 1.*

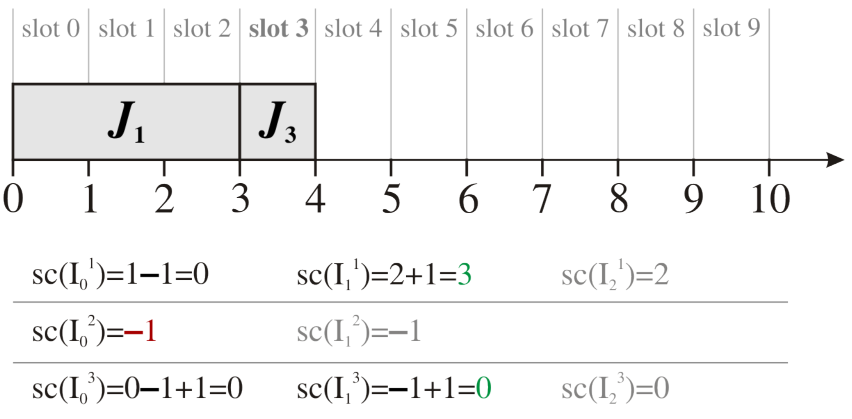
In slot 2, we schedule job J_1 as before. As mentioned in the beginning of the example, all jobs execute for their defined WCET in the current mode, and thus, we assume that J_1 signaled completion in this slot. The spare capacities are updated as in the slot before. A consequence of these updates is that the spare capacities in the current interval of mode 2 become negative. Hence, time instant $t = 3$ is not a feasible time instant to switch the mode change. The spare capacities $sc(I_0^3)$ are still non-negative, and thus, we could switch mode 3. The resulting schedule and spare capacity values are shown in Figure 7.6.



$sc(I_0^1)=1$	$sc(I_1^1)=2$	$sc(I_2^1)=2$
$sc(I_0^2)=0-1=-1$	$sc(I_1^2)=-1$	
$sc(I_0^3)=1-1=0$	$sc(I_1^3)=-1$	$sc(I_2^3)=0$

Figure 7.6: *Spare capacities after slot 2.*

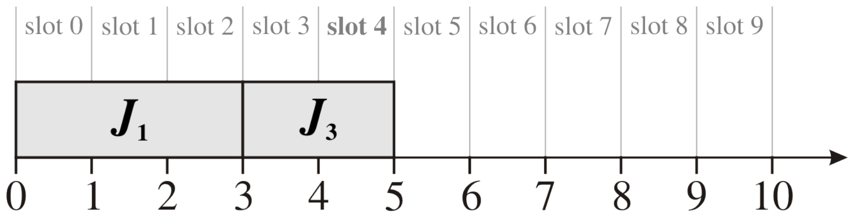
In the current mode, the next job with earliest deadline is J_3 and thus, we schedule it in slot 3. This job is assigned to interval I_1^1 and executed for one slot in interval I_0^1. The consequence of this execution is that one slot is swapped between these two intervals. In other words, $sc(I_0^1)$ is reduced by one slot and $sc(I_1^1)$ is increased by one slot. In mode 2, J_3 is assigned to the current interval and hence, the spare capacities $sc(I_0^2)$ are not changed, but $sc(I_0^2)$ is still negative such that we could not switch to this mode in case of an MCR. In mode 3, J_3 is also assigned to a later interval and thus, one slot is swapped between I_0^3 and I_1^3. Further, interval I_1^3 borrowed one slot from interval I_0^3 and as a consequence, we increase $sc(I_0^3)$ by one slot again. In modes 1 and 3, after the execution in slot 3, the first interval ends and a new interval starts. The spare capacities of these new intervals determine the feasibility of an MCR at $t = 4$. Both $sc(I_1^1)$ and $sc(I_1^3)$ are non-negative and thus, indicate time instances for feasible mode changes. In mode 2, the spare capacities of the current interval are still negative such that a mode change to mode 2 is not feasible. Figure 7.7, shows the schedule and the spare capacities of slot 3.



$sc(I_0^1)=1-1=0$	$sc(I_1^1)=2+1=3$	$sc(I_2^1)=2$
$sc(I_0^2)=-1$	$sc(I_1^2)=-1$	
$sc(I_0^3)=0-1+1=0$	$sc(I_1^3)=-1+1=0$	$sc(I_2^3)=0$

Figure 7.7: *Spare capacities after slot 3.*

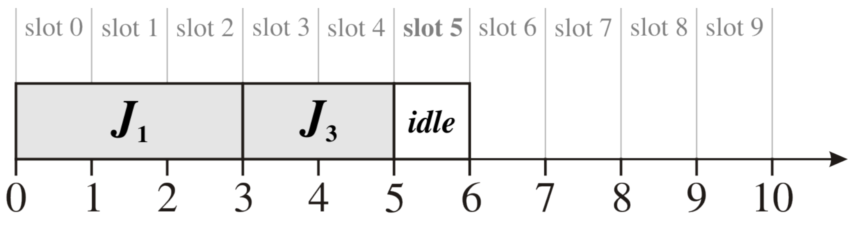
Job J_3 is also scheduled in slot 4 due to its earliest deadline. In this slot, it signals completion and thus, executed in total for two slots. As a consequence, we increase $sc(I_0^2)$ by one slot where a WCET of 3 slots has been considered, and increase $sc(I_1^3)$ by three slots where a WCET of 5 slots has been considered. In all three modes, J_3 has been assigned to the current interval and hence, no further changes to the spare capacities are necessary. In Figure 7.8, the schedule and spare capacity updates are shown.



$sc(I_1^1)=3$	$sc(I_2^1)=2$
$sc(I_0^2)=-1+1=0$	$sc(I_1^2)=-1$
$sc(I_1^3)=0+3=3$	$sc(I_2^3)=0$

Figure 7.8: *Spare capacities after slot 4.*

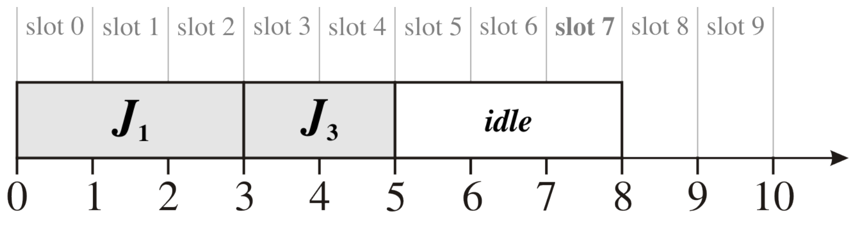
The jobs active in mode 1 are completely executed and hence, we schedule now idle slots until the end of the schedule. An idle slot decreases the spare capacities in the current interval in all modes. The spare capacity value in interval I_0^2 becomes negative, which results in an infeasible mode change to this mode. The resulting schedule and spare capacities are shown in Figure 7.9.



$sc(I_1^1)=3-1=2$	$sc(I_2^1)=2$
$sc(I_0^2)=0-1=-1$	$sc(I_1^2)=-1$
$sc(I_1^3)=3-1=2$	$sc(I_2^3)=0$

Figure 7.9: *Spare capacities after slot 5.*

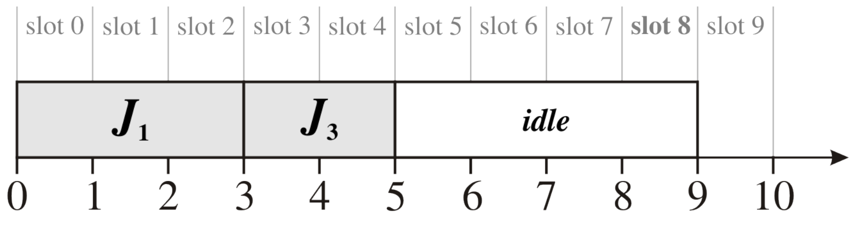
After scheduling idle slots in slots 6 and 7, in all three modes a new interval starts. As a consequence, the spare capacities of the last interval in each mode now determine the feasibility of a mode change. A mode change to mode 1 and 3 is feasible. Changing to mode 1 is trivial because it is the currently active mode, and a mode change to mode 2 is infeasible due to the negative spare capacities. Figure 7.10 shows the schedule and spare capacities after scheduling slot 7.



$sc(I_1^1)=1-1=0$	$sc(I_2^1)=2$
$sc(I_0^2)=-2-1=-3$	$sc(I_1^2)=-1$
$sc(I_1^3)=1-1=0$	$sc(I_2^3)=0$

Figure 7.10: *Spare capacities after slot 7.*

After scheduling an idle slot in slot 8, the spare capacities are decreased by one slot again and the spare capacities in mode 3 become negative, too. Hence, a mode change to this mode is not feasible. In Figure 7.11, we show the resulting schedule and spare capacities.



$sc(I_2^1)=2-1=1$
$sc(I_1^2)=-1-1=-2$
$sc(I_2^3)=0-1=-1$

Figure 7.11: *Spare capacities after slot 8.*

Scheduling slot 9 completes the schedule. The spare capacities of the last intervals in each schedule do not determine feasibility anymore. For a periodic system, we continue with scheduling the same jobs again. As a consequence, time instance $t = 10$ is equivalent to the start of the schedule when all three modes are feasible starting modes. Figure 7.12 shows the complete schedule and the spare capacities of the last intervals in each mode.

Figure 7.13 summarizes the feasible and infeasible mode change instances based on the execution in mode 1. A circle refers to a feasible mode change into this mode at the

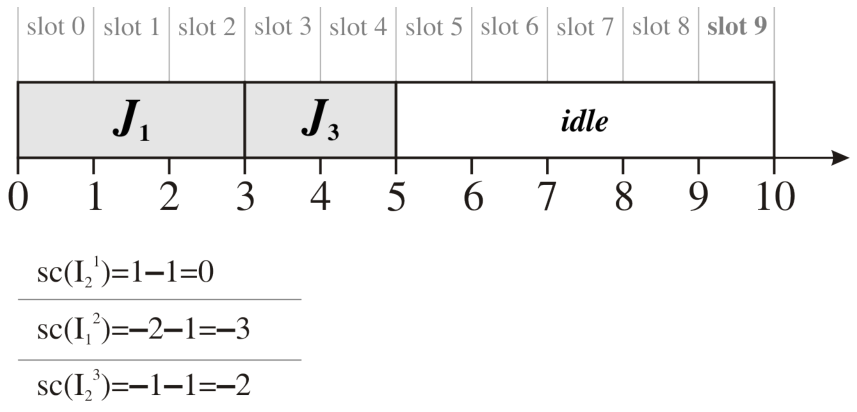


Figure 7.12: *Spare capacities after slot 9.*

beginning of the corresponding slot, whereas a cross corresponds to an infeasible time instant for a mode change.

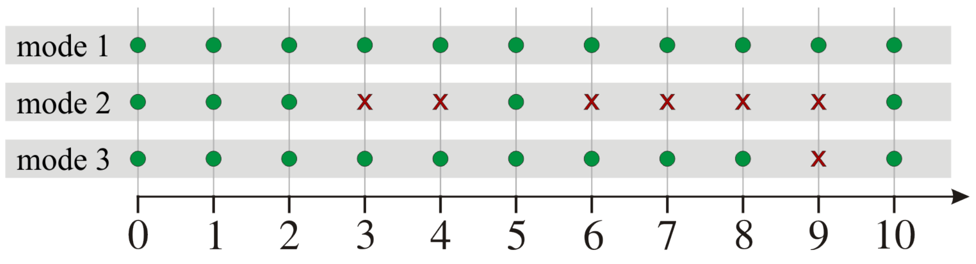


Figure 7.13: *Feasibility of mode changes in mode 1.*

The example showed that we can use the spare capacities to determine the feasibility of mode changes. In contrast to static schedule tables with mode changes, we can react to the actual behavior of jobs. In the static schedule tables, mode change instants can be rare but by the possibility to react on the actual job behavior, i.e., react on the actual execution times, it is possible to react to requests faster and change modes earlier.

7.6 Application of Generic Mode Changes to Mixed-Criticality Systems

So far, we focused on the execution of jobs in one mode and check whether a mode change can be accepted. In the following, we aim at executing jobs such that modes, and consequently jobs, with higher criticality level can be guaranteed.

In Chapter 6, we showed how we fulfill the requirements of mixed-criticality job sets with two criticality levels LO and HI. We assumed that all jobs are characterized by two WCETs but with same release time and deadline for both criticality levels. Burns and Baruah presented an extension of the Vestal mixed-criticality model to allow shorter periods and shorter relative deadline for higher criticality levels of periodic tasks [BB11].

Based on our job model presented before, see Section 7.3, we now allow for independent parameters for all jobs in the different modes.

7.6.1 Assumptions and Requirements

We use the job model presented in Section 7.3 and extend it by the criticality of the job. Jobs are active in all modes with a criticality level up to the criticality level of the job. This requires that the WCETs are specified up to the criticality level of the job. Based on the assumption that different WCETs originate from the confidence in the execution time bound, it can logically be assumed that WCETs of higher criticality levels (modes) are not shorter than the WCETs of lower criticality levels (modes). As a consequence, the modes are ordered according to their criticality level. For instance, in mode 1, i.e., the mode with the lowest criticality level, all jobs are active. Further, in mode 2, only jobs with criticality level greater than or equal to 2 are active, and so on... Eventually, in the highest mode k, only jobs with criticality k are active. Our goal is to always guarantee the highest criticality levels. In other words, we switch to a higher mode when we can guarantee jobs with higher criticality level only if we drop jobs with lower criticality level. For instance, if a job with criticality 5 can only be guaranteed with its highest WCET if all jobs with criticality 4 and lower are dropped, then we have to switch to mode 5 where only jobs with criticality level greater than or equal to 5 are scheduled.

7.6.2 Mixed-Criticality Slot-Shifting with Generic Mode Changes

The offline phase is the same as presented before in Section 7.4 for the non-mixed-criticality case. At runtime, the update of spare capacities and the acceptance test are also unchanged. The differences to non-mixed-criticality slot-shifting with generic mode changes are the requests to change a mode. In contrast to the mode changes triggered by internal or external events, in the mixed-criticality case mode changes are triggered by the requirement to guarantee jobs with higher criticality, i.e., guaranteeing their WCET at their highest criticality level. As a consequence, the selection of the next executing job has to be adapted to this requirement.

Next, we present the selection function that triggers mode changes if a mode change is needed to guarantee jobs with a criticality level higher than the criticality level of the current mode. We use two ready queues for each mode m: $R^{m|\text{firm}}(t)$ contains all guaranteed ready jobs in the current mode and $R^{m|\text{soft}}(t)$ contains all soft aperiodic jobs which are active in the current mode. The possible decision cases in the current mode m_c at time t are shown in the following:

a) $\exists m$ with $sc(I_c^m) = 0$ and $m > m_c$:
Switch to mode m with $sc(I_c^m) = 0$ and highest criticality level. The spare capacities in the current interval in the destination mode are equal to zero and thus, there must be at least on firm job ready to execute. As a result, set destination mode as new current mode m_c and schedule job according to decision case c2.

b) $R^{m_c|\text{firm}}(t) = \{\} \wedge \nexists m$ with $sc(I_c^m) = 0$ and $m > m_c$:

- **1)** $R^{m_c|\text{soft}}(t) = \{\}$:
  We schedule an idle slot because there are no ready jobs.
- **2)** $R^{m_c|\text{soft}}(t) \neq \{\}$:
  The spare capacities $sc(I_c^m)$ are always greater than zero and hence, we can schedule a soft aperiodic job.

c) $R^{m_c|\text{firm}}(t) \neq \{\} \wedge \nexists m$ with $sc(I_c^m) = 0$ and $m > m_c$:

- **1)** $sc(I_c^{m_c}) > 0$:
  If $R^{m|\text{soft}}(t) \neq \{\}$ then we schedule the first soft aperiodic job in the FIFO-queue $R^{m|\text{soft}}(t)$. Else, we select the ready job with the earliest deadline in $R^{m|\text{firm}}(t)$. Only in this situation, we can schedule soft aperiodic jobs if guaranteed jobs are ready to execute.
- **2)** $sc(I_c^{m_c}) = 0$:
  We select the job with the earliest deadline in $R^{m|\text{firm}}(t)$ because there are no available resources for soft aperiodic jobs.
- **3)** $sc(I_c^{m_c}) < 0$:
  In the current interval, the spare capacities of the current mode cannot be less than zero.

In general, we select the job with the earliest deadline in the current mode and if there are available resources and ready soft aperiodic jobs, we can execute them. Mode changes are triggered by the spare capacities of other modes in the current interval which are equal to zero. If spare capacities of a current interval in a mode, which has a higher criticality level than the current mode, are equal to zero, then we switch to the highest mode with spare capacities equal to zero. We illustrate this with an example.
Example. For simplicity, we show an example with only one interval, named current interval I_c^m in mode $m \in \{1, 2, 3, 4\}$, and jobs with the same release time and deadline. Table 7.2 shows the used job set. For higher criticality levels, jobs have monotonically increasing WCETs. If a WCET is not defined, the job is not active in this mode because of a lower criticality level.

jobs	criticality level	WCET in mode 1	WCET in mode 2	WCET in mode 3	WCET in mode 4	release time	deadline
J_1	1	1	—	—	—	0	5
J_2	2	1	1	—	—	0	5
J_3	3	1	2	2	—	0	5
J_4	4	1	2	3	4	0	5

Table 7.2: *Example job set for mixed-criticality slot-shifting with generic mode changes.*

We determine the intervals and calculate the spare capacities. Figure 7.14 shows the spare capacities of the current intervals in all modes before start of the system.

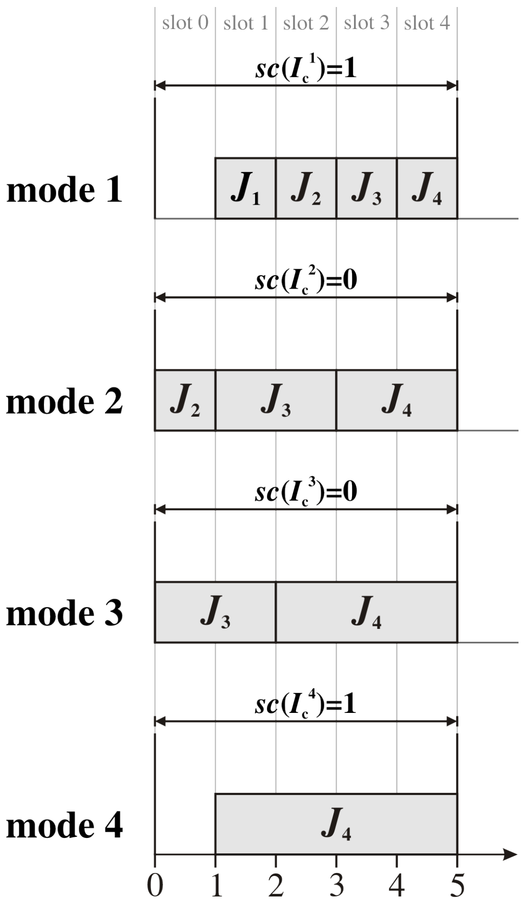


Figure 7.14: *Offline calculated spare capacities.*

Assume we want to start the system in mode 1 and the actual execution times of all jobs are equal to their WCET. At the beginning of slot 0, two spare capacity values in other modes are equal to zero. As a consequence, to guarantee the jobs with higher criticality, we must switch to the highest mode with spare capacities equal to zero in the current interval. In this case, this is mode 3. As a result, jobs J_1 and J_2 will never be executed because of guaranteeing the WCETs of J_3 and J_4 in mode 3. Furthermore, if the tie breaking[2] results in executing J_3 first, we have to switch to mode 4 to guarantee the highest WCET of job J_4 after one slot of execution. The example shows that in case of more than two criticality levels, the mode with the highest workload determines whether other modes with lower criticality level can be executed or not. Hence, the number of criticality levels is not the dominating factor but the more pessimistic WCETs of modes

[2]Tie breaking rules are applied if two or several jobs have the same priority. Tie breaking rules must not change to deterministically schedule the job set. In this case, both jobs have the same deadline and thus, the same priority. A tie breaking that would favor J_3 is tie breaking by index, i.e., the job with the smaller index is selected first.

with higher criticality level. As a result, it is possible that many intermediate criticality are skipped and directly, a mode with relatively high criticality is switched to. ◊

In summary, the mode with the highest workload and the highest criticality determines whether we can schedule jobs in lower modes or not. This can result in extremely pessimistic scheduling decisions. As we mentioned in Section 7.2.2, having too many criticality levels, and hence, many different WCETs, is not reasonable. Do we get a benefit by adding many intermediate WCETs between a "normal" WCET and a very pessimistic WCET? Eventually, the difference between the shortest and the longest WCET of one job determines the degree of pessimism for the scheduling decisions. In the next section, we discuss the problems of many different WCETs and the use of many different modes to guarantee them.

7.6.3 Applicability to Mixed-Criticality Systems with Shared Resources

So far, we considered mixed-criticality jobs which do not share resources. The challenge of systems with shared resources is to prevent a simultaneous access of several jobs to the same resource. For instance, two jobs writing into the same memory block results in inconsistent and erroneous data. Thus, we need to prevent this situation, which is called *mutual exclusion*. Common solutions to solve this problem increase the priority of a job accessing the shared resource such that a competing job cannot preempt the job already accessing the resource. Examples are the Priority Inheritance Protocol (PIP) or the Priority Ceiling Protocol (PCP) [SRL90]. In both examples, the priority of a job is temporarily increased such that mutual exclusion is ensured.

In the following, we present the rationale how we can achieve mutual exclusion based on the slot-shifting approaches shown in this chapter. To achieve mutual exclusion, we have to ensure that once a job accesses a resources, no other job can access the same resource. We can implement this by a temporary mode which consists of the same jobs with the same parameters as in the current mode, but we need to remove all jobs which can also access this resource from the ready queues. The temporary mode is dynamically created at runtime and is used to block jobs competing for the resource. After release of the resource, we switch back to the original mode and this temporary mode is removed. By doing these steps, we also prevent deadlocks. A deadlock can occur when two jobs need to access the same two resources. One jobs accesses the first resources and the other job the second resource. In this situation both jobs wait for the other job to release the other resource. As a result, both jobs are stuck and cannot continue with their execution.

For reason of clarity of explanation, we restrict description of the basic idea of mixed-criticality jobs with shared resources to dual-criticality systems. As long as LO-criticality jobs do not share resources with HI-criticality jobs, we can guarantee mutual exclusion and the execution of HI-criticality jobs. If LO-criticality and HI-criticality jobs share resources another problem arises: Assume a LO-criticality job accesses a resource which is shared with a HI-criticality job. During the execution of this LO-criticality job, we have to execute the HI-criticality job to guarantee its HI-criticality requirements. For instance, the correctness of data – the shared resource is represented by shared memory –

competes with the guarantee of the certified HI-criticality behavior. In both cases, the safe operation of the system cannot be ensured. As a consequence, we need to determine the maximum time HI-criticality jobs can be blocked by LO-criticality jobs accessing a shared resource before we can grant access to the resource. Only if all HI-criticality jobs can meet their timing requirements even when they experience this maximum blocking time, LO-criticality jobs can access a shared resource.

The above mentioned problem and basic idea for solutions are not necessarily complete. The complete solution to the mixed-criticality shared resources problem using the presented slot-shifting approach with generic mode changes remains future work.

7.7 Discussion: How Much Flexibility in the Mixed-Criticality Model Is Reasonable?

In the following, we focus on the criticality levels and modes. In this chapter, we showed an approach based on slot-shifting to handle mode changes in general and mode changes in mixed-criticality systems. With the presented methods, we can schedule jobs which are active only in a subset of the modes and can be characterized by different parameters in the different modes.

First, we consider the origin of different criticality levels. Vestal introduced the term “criticality” to use several WCETs per task [Ves07], whereas WCETs of higher criticality levels are greater than or equal to WCETs of lower criticality levels, i.e., monotonically increasing WCETs. This definition corresponds to the *confidence* in the bounds on the WCETs. Burns and Baruah extended this definition by allowing shorter periods and shorter relative deadlines with increasing criticality level [BB11]. For instance, sampling tasks can provide a better quality of service with shorter sampling periods. By doing this, the approach can increase the *importance* of tasks when they are scheduled with period- or deadline-driven schedulers, e.g., Rate Monotonic, Deadline Monotonic, or Earliest Deadline First. A usual assumption in mixed-criticality schedulers is the dropping of lower criticality jobs and tasks if the system changes to a higher criticality state. Using this assumption, we can influence the flexibility in scheduling which refers to the *mode* of the scheduler. As a result, we obtain job sets and tasks sets with varying WCETs, periods, and deadlines and different scheduling requirements depending on the system state.

Second, we review the results of the presented approach in this chapter. Using slot-shifting with generic mode changes, we can schedule jobs with different parameters in different modes. Further, we can determine the feasibility of mode changes to a destination mode such that active jobs can execute for their WCET until their deadline. Additionally, we also presented how we can use this approach to guarantee the requirements of mixed-criticality job sets. Guaranteeing high criticality jobs with possible long WCETs can result in very pessimistic scheduling decisions. In conclusion, the flexibility of scheduling and guaranteeing mixed-criticality job with varying WCETs and deadlines can result in dropping many jobs with lower criticality levels.

With the aforementioned information, we now try to answer the question on how many criticality levels and thus, modes, we need. Our scheduler works on job basis and as a consequence, handling periodic tasks with our scheduler is straight forward. Tasks with varying periods and relative deadlines, thus, resulting in jobs with changing release times and absolute deadlines, are reasonable. The example of sampling tasks illustrates the benefits of shorter periods and relative deadlines in higher criticality levels. We can fulfill these requirements with the presented slot-shifting scheduler with generic mode changes. The more difficult aspects are the different WCETs and criticality levels. The confidence in the execution time bounds justifies dual-criticality jobs with two WCETs. On the contrary, mixed-criticality systems with many criticality levels and consequently, many different WCETs per job cannot be justified by the confidence in the execution time bounds. Especially, the assumption that WCETs of higher criticality levels are greater than WCETs of lower criticality levels is not reasonable. If designers want to increase the importance of jobs and tasks, then shorter periods and shorter relative deadlines are more reasonable than increasing WCETs. Furthermore, additional methods as fault tolerance algorithms can improve the quality of service of mixed-criticality systems. In summary, jobs with many WCETs, which always have to be guaranteed at highest criticality level, result in pessimistic scheduling and bad overall performance of the system. Further, the pessimistic WCETs of high criticality levels can result in skipped many modes with lower criticality at once such that the intermediate criticality levels do not provide any benefits.

Chapter 8

Conclusions

In this thesis, we focused on the problem of scheduling mixed-criticality systems which are subject to certification. Increasing processing power and decreasing structure sizes made it possible to integrate many functionalities onto a single chip. In mixed-criticality systems, safety-critical and non-safety-critical functionalities are implemented on these system-on-chips. For instance, the software standards AUTOSAR [AUT] in the automotive industry and ARINC [ARI] in the avionics domain address such mixed-criticality systems. The mixed-criticality nature of tasks and jobs is usually represented by their parameters worst-case execution time (WCET), periods, and deadlines. These safety-critical systems are often subject to certification, i.e., the correctness of safety-critical functionalities has to be proven under conservative assumptions. This difference between certified and non-certified tasks and jobs can be expressed by two criticality levels LO and HI which results in dual-criticality systems. As a consequence, we need effective and efficient scheduling policies which are cognizant of the mixed-criticality nature of tasks and jobs. The certification is also a major aspect of these systems such that reduction of certification costs is also a major goal of scheduling. Finally, besides the unknown actual behavior of tasks and jobs at runtime, we aimed at handling event-triggered (ET) activities.

8.1 Main Contributions

In this thesis, we presented different methods to fulfill the requirements of mixed-critical systems which are subject to certification. In the following, we review the fundamental steps of the methods, the results, and their consequences.

Using the presented sporadic task transformation, we created periodic reservation tasks. We can include them in the offline scheduling process such that we can guarantee the execution of all sporadic task instances independent of the actual arrival time at runtime. The method provides reservation tasks with different parameter combinations for the corresponding sporadic task. The different combinations allow for a reduction of the utilization of the reservation tasks and a reduction of the worst-case response time (WCRT) of the sporadic tasks. A major advantage of the method is its independence of a scheduling algorithm, i.e., as long as the scheduler can feasibly schedule periodic tasks, the resulting reservation tasks can be used to guarantee the runtime execution of sporadic tasks. As a result, we can include sporadic tasks already in the offline scheduling process. The disadvantage of this approach is that we have to reserve enough execution time for the worst-case scenario of sporadic task arrivals which is pessimistic. Besides the application on sporadic tasks, we can use the method also to transform periodic tasks. In this case, we obtain periodic tasks with different WCETs and periods. Performing this transformation increases the utilization of the periodic tasks. On the contrary, we get additional feasible periods which influence the length of the hyper-period. The hyper-period determines the length of a schedule table. As a consequence, long schedule tables increase the certification costs. Hence, a slight increase in the utilization of the periodic tasks can be justified by an extremely shorter hyper-periods.

The aforementioned task transformation aims at the inclusion of ET (sporadic) activities into schedule tables. This transformation is applicable to our schedule table generation algorithm for dual-criticality time-triggered (TT) systems. In this scheduler, we use mode changes to fulfill the requirements of dual-criticality job sets. On the one hand, we implement one mode to construct a feasible schedule with all jobs, i.e., LO- and HI-criticality jobs, based on the LO-criticality WCETs of the designers. On the other hand, in the second mode, we guarantee the conservative requirements of the Certification Authorities (CAs) of the HI-criticality jobs. The resulting schedule tables are executed at runtime resulting in a low execution overhead. Furthermore, they represent a constructive proof such that certification costs can be reduced. Besides the general advantages of TT schedulers with mode changes, the method can efficiently generate the schedule tables. In our algorithm, we reduce the complexity of scheduling by splitting the mixed-criticality jobs with two WCETs into jobs with only one WCET. The scheduling process is based on a search tree. We guide the search for a feasible schedule using the HI-criticality demand of HI-criticality jobs. Further, we use this demand to reduce the complexity of the implemented backtracking procedure. As a result, we reduce the complexity of scheduling the mixed-criticality job set. We evaluated the efficiency and effectiveness of our scheduler by comparing it with Baruah's and Fohler's TT mode change scheduler [BF11] which is based on Audsley's optimal fixed priority assignment [Aud91, Aud93]. The evaluation results showed that our presented scheduler is less time consuming in the schedule table construction process and can create a feasible schedule table for mixed-criticality test cases.

So far, the mentioned methods completely work at design time. We also presented a slot-shifting [Foh95] based method without mode changes that can make use of an already existing certified schedule table. The dual-criticality algorithm makes use of the schedule table to determine intervals and spare capacities, i.e., amount of available resources within these intervals. At runtime, the scheduler can handle ET activities while guaranteeing the timing constraints of TT jobs. The major advantage of this method is the ability to react to the actual behavior of jobs. If a job finishes its execution earlier than its WCET, the scheduler can use the gained computation time to execute TT jobs earlier and integrate more ET activities. Additionally, if jobs show HI-criticality behavior at runtime, then we do not necessarily skip LO-criticality jobs. LO-criticality jobs are only skipped when the execution of them jeopardizes the timing constraints of HI-criticality jobs. As soon as computational resources are available for

LO-criticality jobs, the scheduler schedules LO-criticality jobs again. As a consequence, the selection function allows for criticality based scheduling decisions. The provided acceptance test allows for an inclusion of ET jobs without interference of already guaranteed jobs. The possibility to re-use already certified schedule tables allows for flexibility in the scheduling process without adding certification costs. Schorr and Fohler showed that slot-shifting has an applicable runtime overhead [SF13], which results in efficient scheduling at runtime.

The presented slot-shifting method without mode changes is restricted to dual-criticality systems. We extended this approach to schedule job sets with more than two criticality levels using different modes for the different criticality levels. Further, the extension allows for varying job parameters, and adding and removing jobs in different modes. On the one hand, the method can schedule jobs in different modes and check the feasibility of mode change requests (MCRs) and perform mode changes when an event triggers an MCR and the MCR is feasible. On the other hand, the scheduler can trigger mode changes to guarantee the requirements of a mode with a higher criticality level. As a result, the algorithm can schedule mixed-criticality job sets with an arbitrary number of criticality levels. Further, the method allows for varying job WCETs and deadlines in different modes. This approach improves the flexibility in scheduling and variation of job parameters compared to the approaches shown before. On the contrary, updating many modes during scheduling increases the runtime overhead such that this method is applicable for a reasonable number of modes. Finally, the method provides also an acceptance test for ET jobs such that the feasibility of including them into modes in which the jobs are active can be tested.

8.2 Final Remarks

In this thesis, we showed that we can effectively reduce the complexity of certification by the implementation of TT schedule tables. The efficient construction of these schedule tables is a challenging problem. We showed that a separation of LO-criticality and HI-criticality WCET requirements can reduce the complexity of scheduling dual-criticality jobs. As a result, we can simplify the mixed-criticality jobs to jobs characterized by only one WCET. This result has been used by Jan et al. to implement a mode change scheduler based on linear programming [JZLP14]. Further, the construction of schedule tables for different modes can be optimized by including knowledge about the LO- and HI-criticality requirements in both modes. We evaluated these results by comparing our method to generate schedule tables with an approach based on an optimal priority assignment to construct schedule tables for dual-criticality TT systems.

Using TT schedule tables provides a very low runtime overhead. On the contrary, this does not provide any flexibility in the execution and thus, can be pessimistic. We showed that slot-shifting makes use of the advantages of TT schedule tables. Further, we can react to the actual behavior of jobs at runtime such that we can effectively use of the available resources. Due to the possibility to use already certified schedule tables, the method simplifies the certification process.

Additionally, we discussed the mixed-criticality job model. We concluded that the criticality of a job should represent more than just different WCET values. Different WCETs are justified by different levels of confidence in the upper bound of the execution times. The often implemented assumption that we can skip jobs with lower criticality level to guarantee high criticality jobs is too pessimistic. In contrast to this, all jobs should be allowed to make some progress and high criticality jobs are only prioritized as long as it is necessary to guarantee their timing constraints. We showed that different job parameters, as longer relative deadlines and shorter execution budgets, are suitable approaches to achieve this goal. As a consequence, scheduling focuses on the guarantee of different job parameters depending on the criticality level, but additionally, methods as fault tolerant algorithms need to be applied to obtain reliable safety-critical systems

As future work, the presented slot-shifting approach with generic mode changes can be applied to mixed-criticality job sets with shared resources. To develop a reliable solution, we have to define the assumptions about resources which are shared between LO-criticality and HI-criticality jobs. Based on these assumptions, we have to elaborate a solution for the competing aspects of ensuring mutual exclusion and the guarantee of the certified behavior of HI-criticality jobs. The absolute deadlines of the jobs are one of the properties the presented method uses to schedule jobs. Using these deadlines as priorities of the jobs, we can implement resources sharing protocols as Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP) to reduce blocking times, etc. For instance, a possible solution is to shorten the deadlines while accessing a shared resource which corresponds to an increased priority. In future work, we aim at including shared resources into our methods and improve the scheduling of jobs with shared resources, e.g., by the implementation of resource sharing protocols as PIP and PCP.

Appendix A

Extended Results

A.1 Further Evaluation Results of the Effectiveness and Efficiency of the Period Transformation

In the following, we show the skipped evaluation results of our transformation method. The method generates parameters for reservation tasks to guarantee sporadic and periodic tasks. Using this new parameters, we get further feasible periods for periodic tasks, which can be used to minimize the hyper-period and thus, reduce the memory consumption of time-triggered (TT) schedule tables, for instance. In Chapter 4, we presented the method and the major results for the evaluation of the method.

In our evaluation, we have run experiments with $\kappa \in \{7, 8, 9, 10\}$ additional periods per periodic task. We have shown the results for $\kappa = 8$ as a representative example. In the following, we complete the results by showing the evaluation for the missing values of κ which are $\kappa \in \{7, 9, 10\}$. As before, we performed our experiments on an Intel XEON processor E5-2670 running at 2.60 GHz. The experiments are comprised of $n \in \{2, ..., 60\}$ periods tasks with randomly distributed periods between 50 and 1000 time units. For each parameter combination, we performed 100 experiments to measure the runtimes of the Fast Hyper-period Search (FHS) algorithm. The shown graphs in this appendix confirm the results shown in Section 4.7.

Figure A.1 depicts a box plot for $\kappa = 7$ confirming the strongly increasing runtimes with increasing number of tasks n. There is a wide range of runtimes for each n. The scatter plot in Figure A.2 shows all measured runtimes for this case. Due to the fact that we performed 100 experiments per parameter combination, a clear distinction between the resulting runtimes is difficult. The major result of this graph is the wide range of resulting runtimes which confirm the high data dependency of the hyper-period calculations.

Figures A.3, A.4, A.5 and A.6 confirm this behavior of FHS for $\kappa = 9$ and $\kappa = 10$.

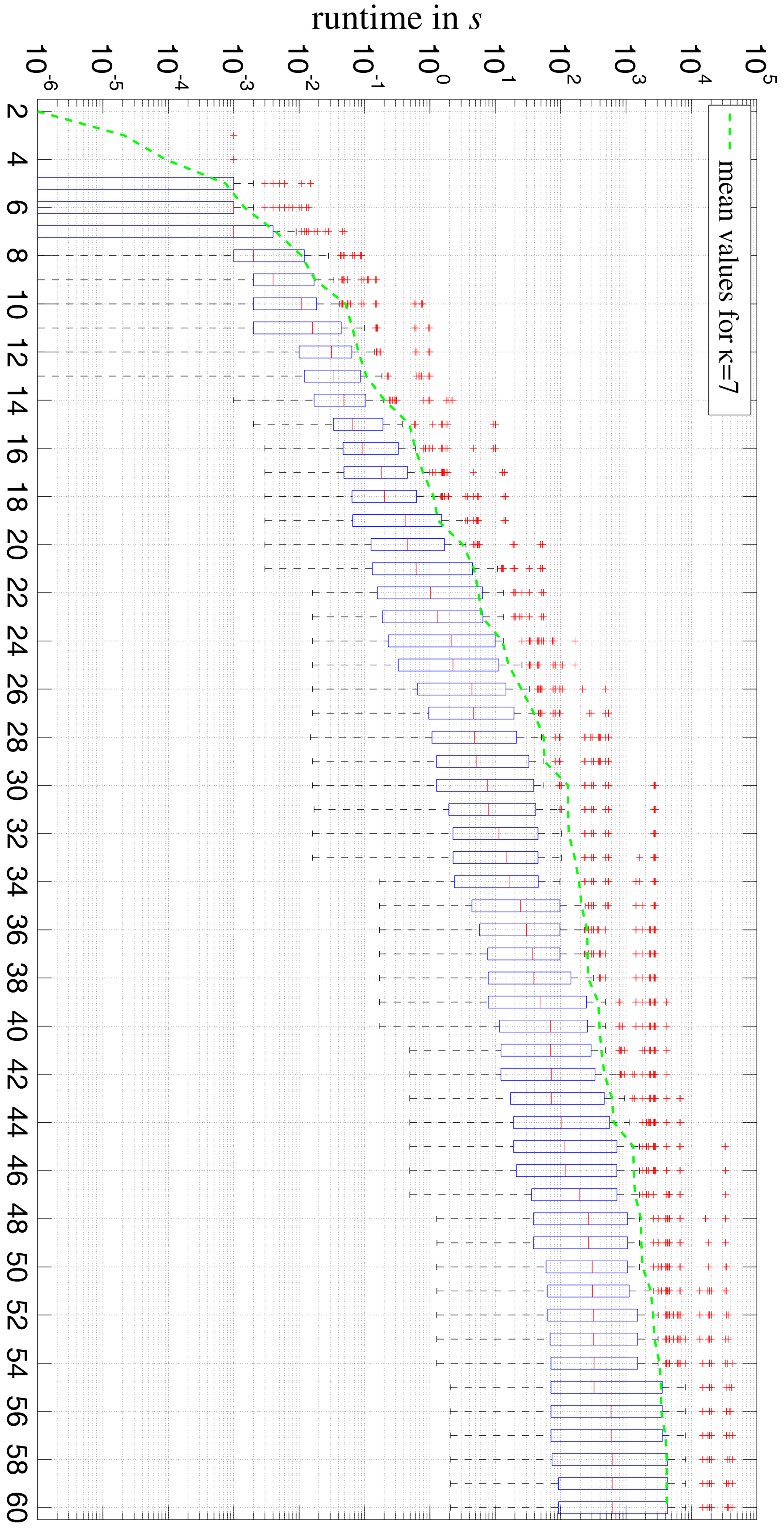


Figure A.1: *Box plot with extended results for up to 60 tasks with 7 additional periods.*

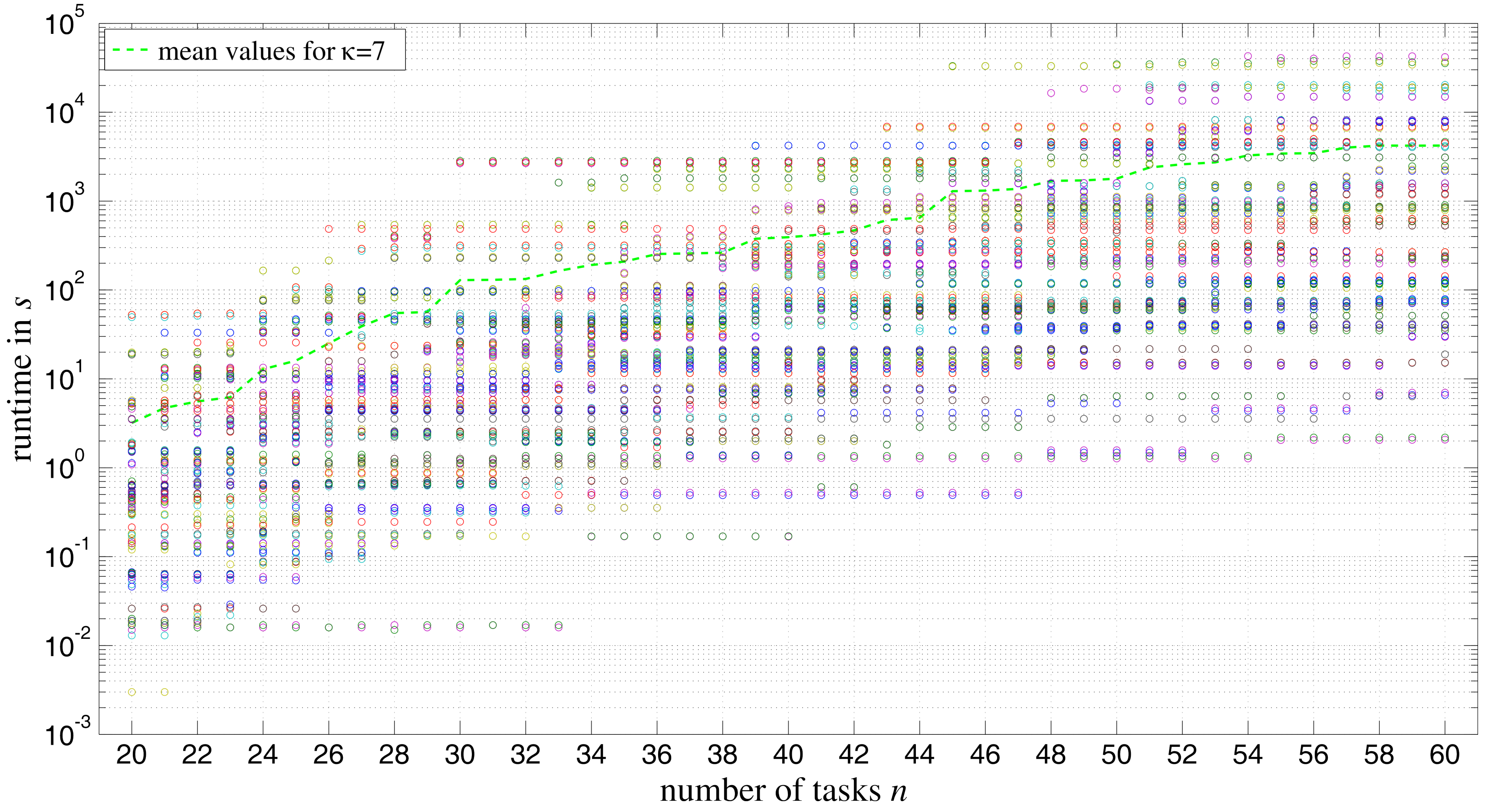


Figure A.2: *Scatter plot with extended results for up to 60 tasks with 7 additional periods.*

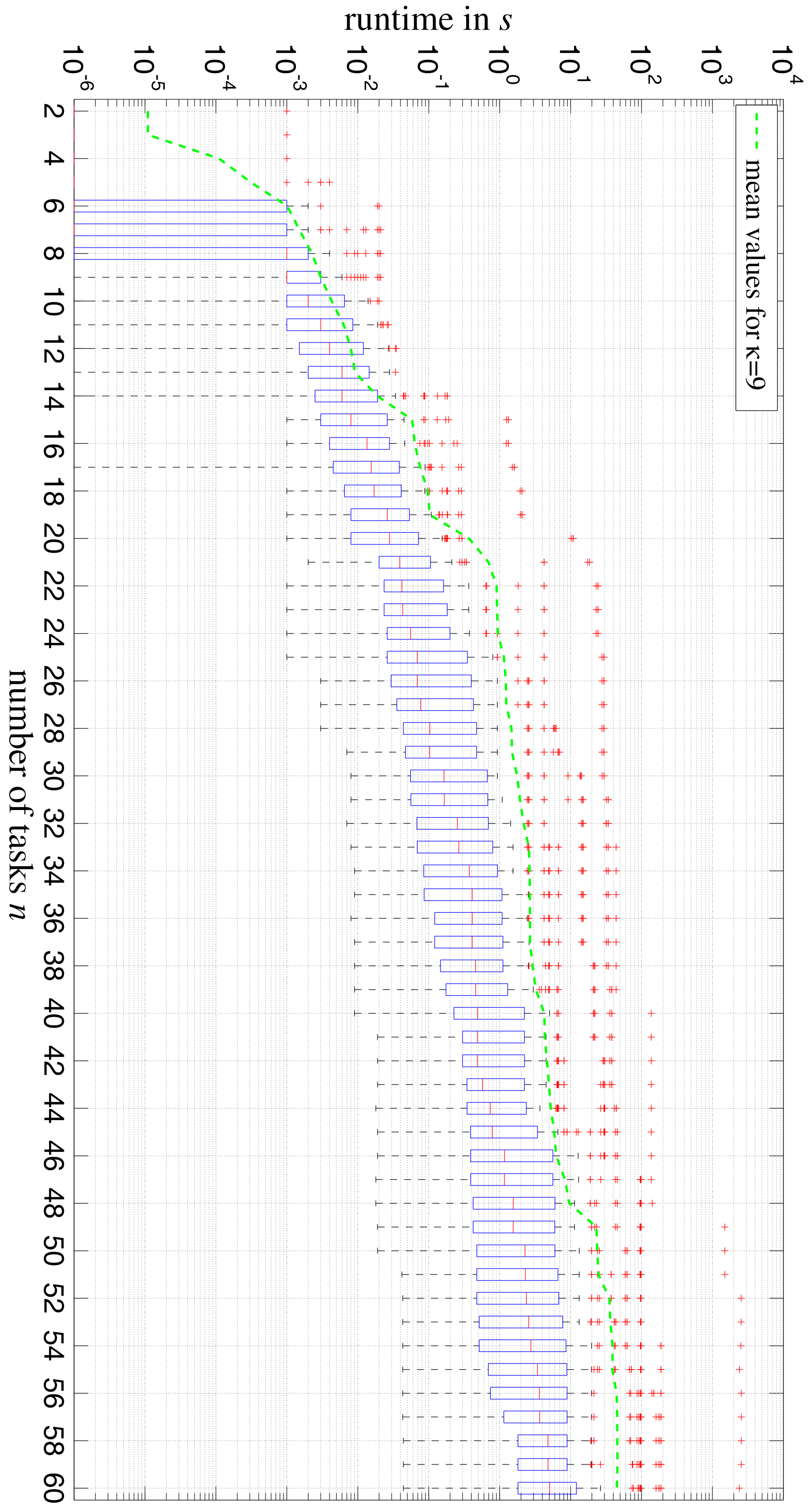


Figure A.3: *Box plot with extended results for up to 60 tasks with 9 additional periods.*

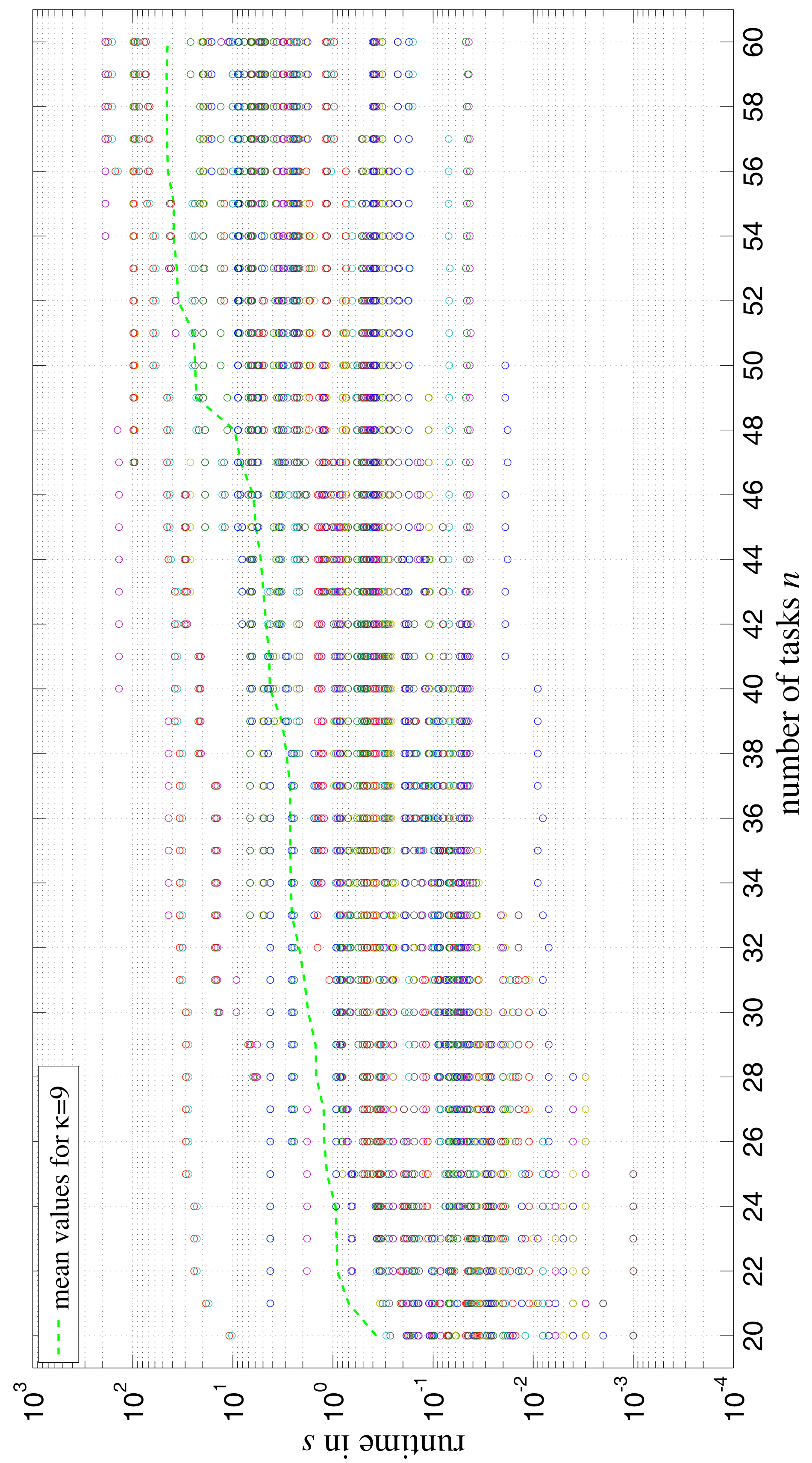


Figure A.4: *Scatter plot with extended results for up to 60 tasks with 9 additional periods.*

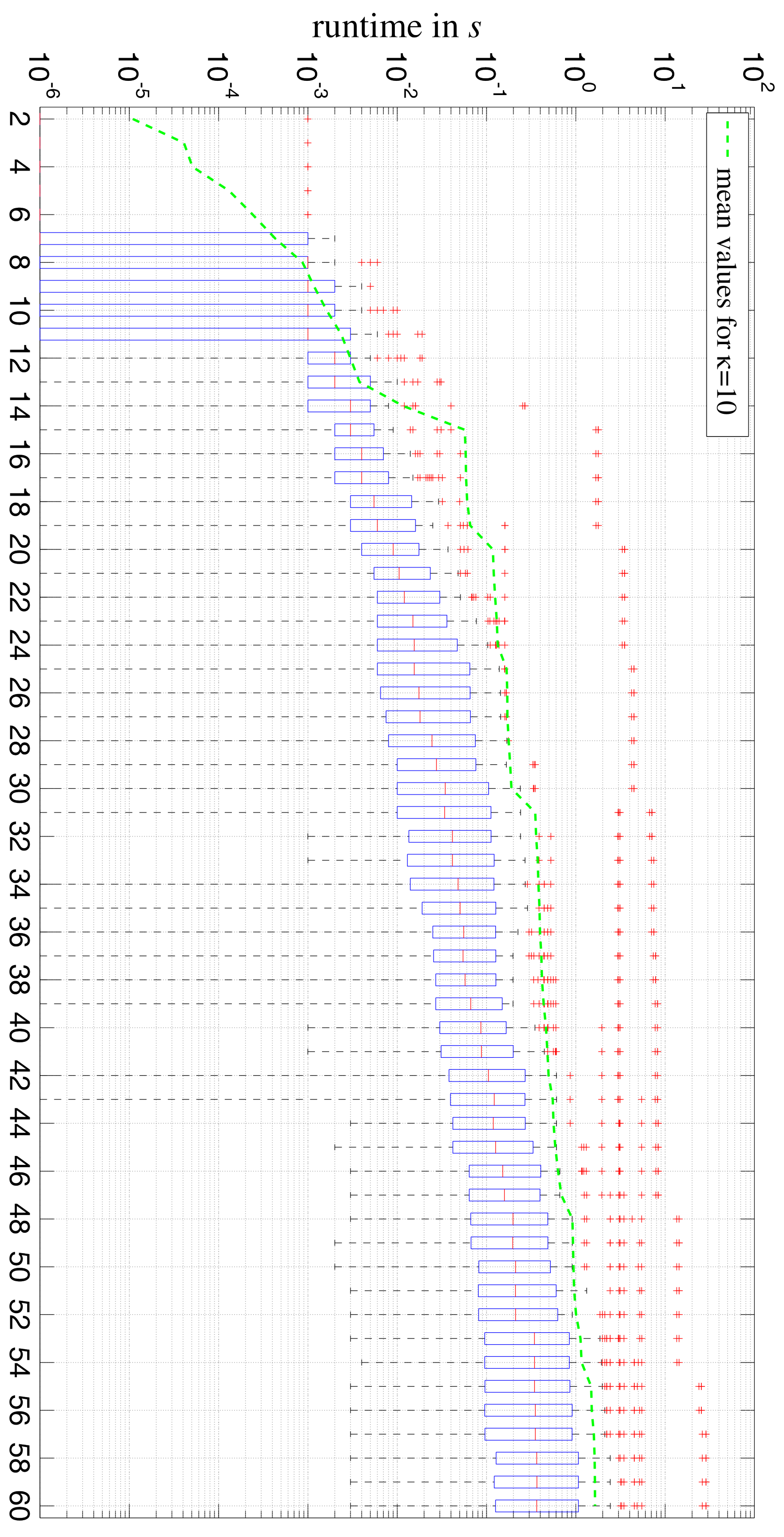


Figure A.5: *Box plot with extended results for up to 60 tasks with 10 additional periods.*

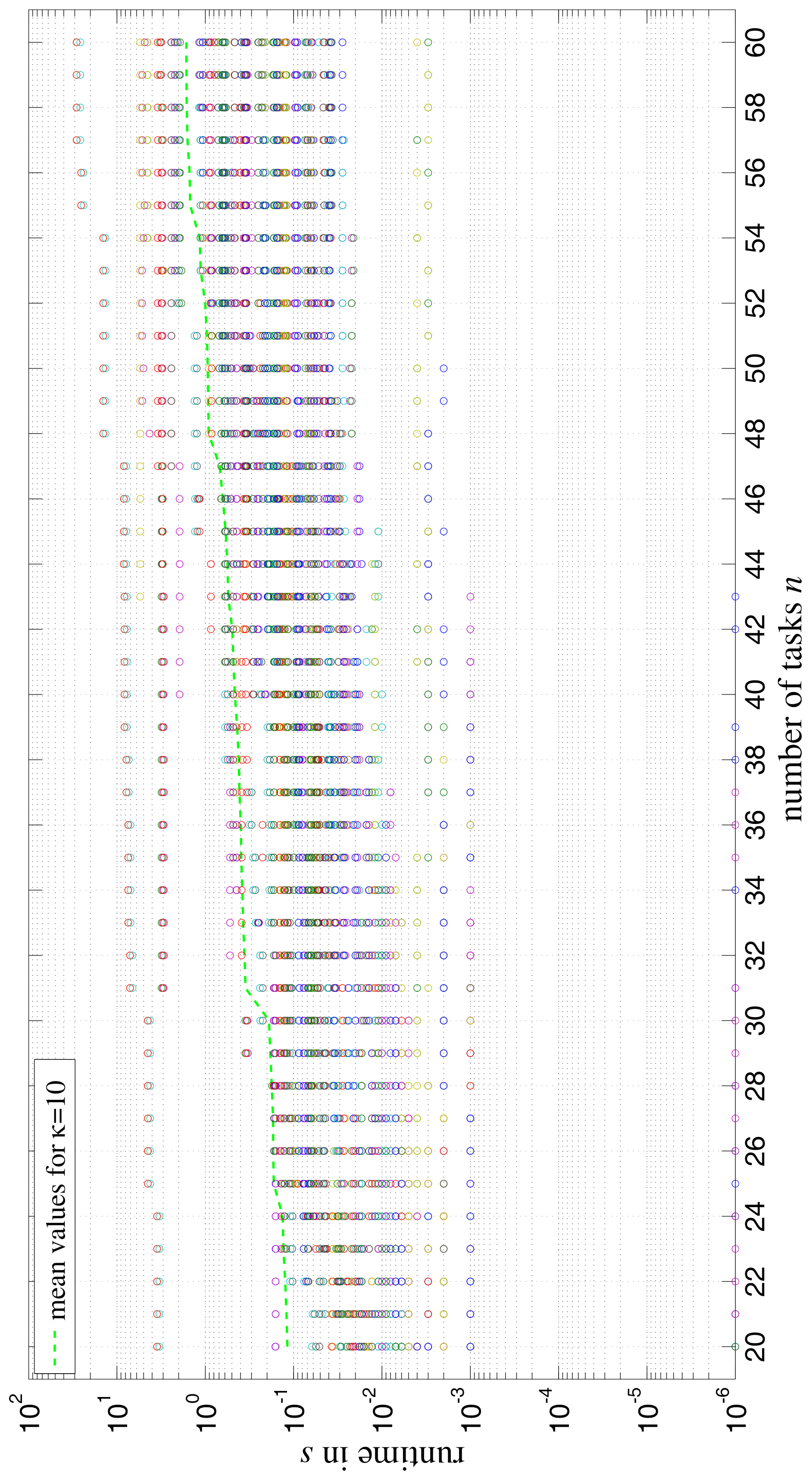


Figure A.6: *Scatter plot with extended results for up to 60 tasks with 10 additional periods.*

A.2 Extended Evaluation of the Algorithms SWAP and FPS

In the following, we present more experimental results on the success ratio for our presented algorithms SWAP and Baruah's and Fohler's fixed priority scheduling (FPS) approach called FPS. In these experiments, we generated the input job sets for the scheduler with jobs sets comprised of a higher number of jobs with smaller worst-case execution times (WCETs) and shorter execution windows. Both WCETs and execution windows are scaled down proportionally, but in this way, the high scale factor hsf has a stronger impact on the success ratio. By doing this, we show that the introduced pessimism by the Certification Authorities (CAs) has a strong impact on the results of both algorithms.

In the following results, we see that the drop of the success ratio, which we explained in our main experiments in Section 5.6, begins already for lower LO-criticality utilizations and the success ratios drop faster. The shown experiments strengthen the fact that FPS works better when there is little idle time or few LO-criticality jobs in the schedule while SWAP is better in more balanced job sets. Further, the scheduling of job sets with very high workloads is for both algorithms very difficult. As a result, the problem is not which algorithm to choose but how can we avoid too high HI-criticality utilizations in the system because this factor has the strongest impact in these experiments. The following tables and figures show the success ratio for both SWAP and FPS in three groups of experiments with high scale factors $hsf \in \{2, 3, 5\}$.

		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$															
success		10%		20%		30%		40%		50%		60%		70%		80%	
ratio [%]		SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS
HI-ratio	25%	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.6	98.5	93.7	76.7	52.9	44.0	4.5	16.2	0.0
	50%	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.0	92.9	90.6	32.8	34.3	4.2	0.6	0.4	0.0
	75%	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.5	87.8	90.5	5.8	25.7	0.0	0.0	0.0	0.0

Table A.1: *Extended results: SWAP and FPS success ratios for $hsf = 2$.*

		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$															
success		10%		20%		30%		40%		50%		60%		70%		80%	
ratio [%]		SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS
HI-ratio	25%	100.0	100.0	100.0	100.0	100.0	99.4	89.8	87.3	52.9	37.1	25.2	5.4	10.0	0.4	2.9	0.0
	50%	100.0	100.0	100.0	100.0	100.0	99.0	64.2	77.0	11.2	17.4	1.4	1.8	0.0	0.0	0.0	0.0
	75%	100.0	100.0	100.0	100.0	100.0	98.9	35.2	71.1	11.0	11.0	0.0	0.1	0.0	0.0	0.0	0.0

Table A.2: *Extended results: SWAP and FPS success ratios for $hsf = 3$.*

		LO-criticality utilization $\sum_{i=1}^{n} U_i(\text{LO})$															
	success	10%		20%		30%		40%		50%		60%		70%		80%	
	ratio [%]	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS	SWAP	FPS
HI-ratio	25%	100.0	100.0	100.0	100.0	71.7	71.3	30.4	9.4	10.8	0.6	*	*	*	*	*	*
	50%	100.0	100.0	99.9	100.0	24.5	31.0	2.7	0.7	0.1	0.0	*	*	*	*	*	*
	75%	100.0	100.0	99.8	99.9	7.2	13.5	0.3	0.0	0.6	0.2	*	*	*	*	*	*

**** no job sets generated for this case***

Table A.3: *Extended results: SWAP and FPS success ratios for $hsf = 5$.*

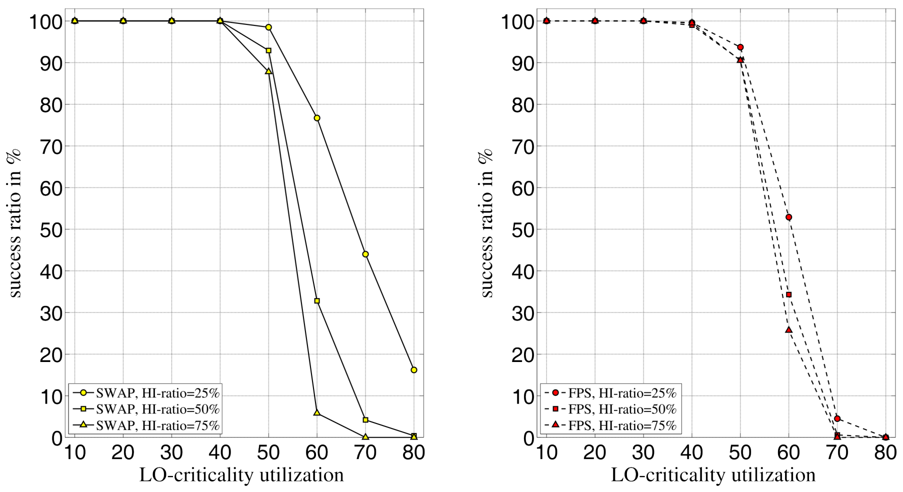


Figure A.7: *Success ratio for high scale factor $hsf = 2$.*

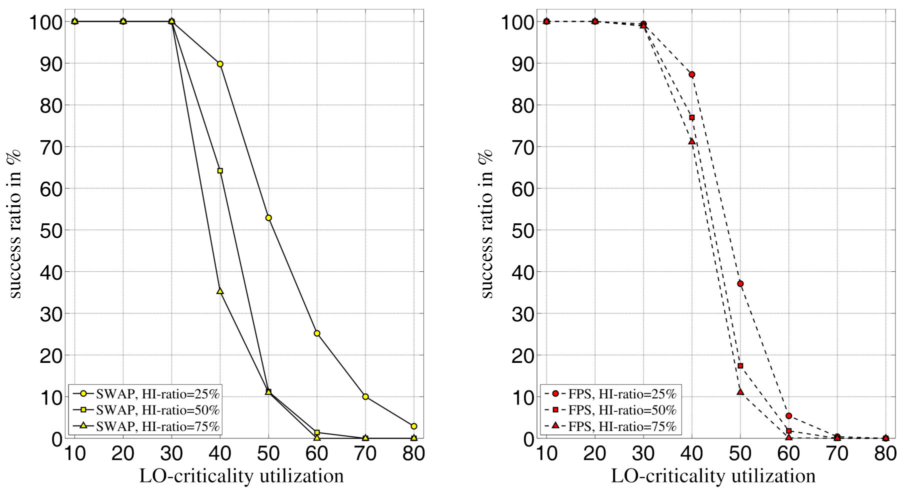


Figure A.8: *Success ratio for high scale factor $hsf = 3$.*

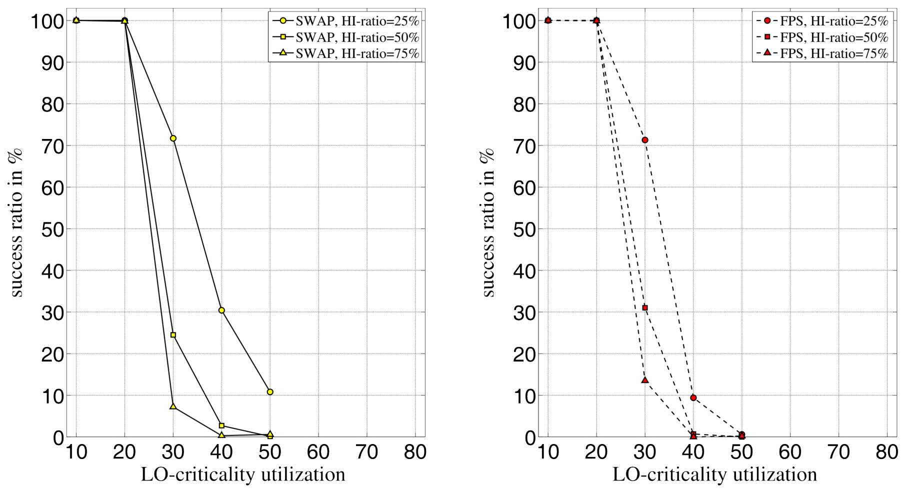


Figure A.9: *Success ratio for high scale factor $hsf = 5$.*

A.3 Resulting Example Schedules of Slot-Shifting with Generic Mode Changes

In the following, we present the resulting schedules and spare capacities after each steps of the scheduling process of the example presented in Section 7.5. Table A.4 shows the job set used.

jobs	WCET in mode 1	WCET in mode 2	WCET in mode 3	release time	deadline
J_1	3	—	1	0	4
J_2	—	2	—	0	8
J_3	2	3	5	0	8
J_4	—	3	2	0	10

Table A.4: *Example job set for slot-shifting with generic mode changes.*

First, we assume that the system operates only in mode 2. Figure A.10 depicts the resulting schedule and Table A.5 shows the spare capacities at each time instant. The feasibility of mode changes is summarized in Figure A.11.

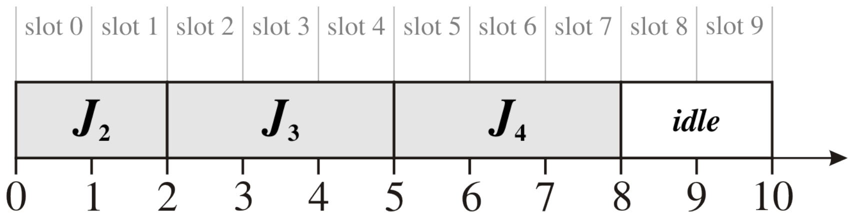


Figure A.10: *Resulting schedule of mode 2.*

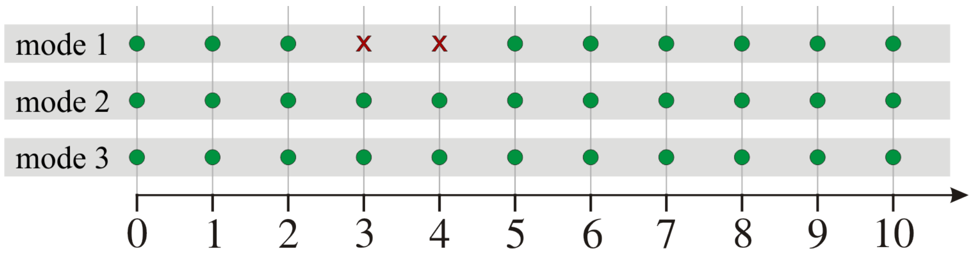


Figure A.11: *Feasibility of mode changes in mode 2.*

	$sc(I_0^1)$	$sc(I_1^1)$	$sc(I_2^1)$	$sc(I_0^2)$	$sc(I_1^2)$	$sc(I_0^3)$	$sc(I_1^3)$	$sc(I_2^3)$
$t = 0$	**1**	2	2	**2**	-1	**2**	-1	0
$t = 1$	**0**	2	2	**2**	-1	**1**	-1	0
$t = 2$	**-1**	2	2	**2**	-1	**0**	-1	0
$t = 3$	**-2**	3	2	**2**	-1	**0**	0	0
$t = 4$	-3	**4**	2	**2**	-1	-1	**1**	0
$t = 5$		**3**	2	**2**	-1		**3**	0
$t = 6$		**2**	2	**2**	0		**2**	1
$t = 7$		**1**	2	**1**	1		**1**	2
$t = 8$		0	**2**	0	**2**		0	**2**
$t = 9$			**1**		**1**			**1**
$t = 10$			0		0			0

Table A.5: *Spare capacities in mode 2 (highlighted values are indicate feasibility of mode change; horizontal lines in the table indicate interval borders).*

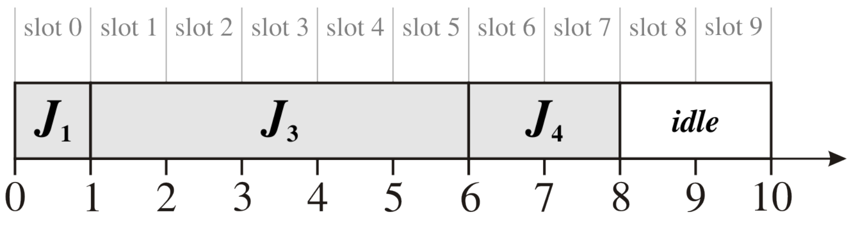


Figure A.12: *Resulting schedule of mode 3.*

Now, we assume that the system operates only in mode 3. Figure A.12 depicts the resulting schedule and Table A.6 shows the spare capacities at each time instant. The feasibility of mode changes is summarized in Figure A.13.

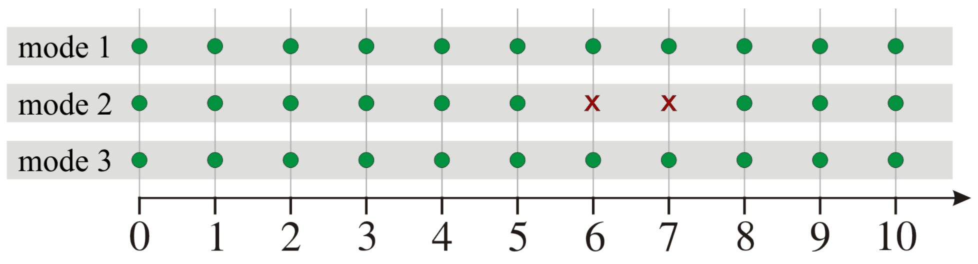


Figure A.13: *Feasibility of mode changes in mode 3.*

	$sc(I_0^1)$	$sc(I_1^1)$	$sc(I_2^1)$	$sc(I_0^2)$	$sc(I_1^2)$	$sc(I_0^3)$	$sc(I_1^3)$	$sc(I_2^3)$
$t=0$	**1**	2	2	**2**	-1	**2**	-1	0
$t=1$	**3**	2	2	**1**	-1	**2**	-1	0
$t=2$	**2**	3	2	**1**	-1	**2**	0	0
$t=3$	**1**	4	2	**1**	-1	**1**	1	0
$t=4$	0	**4**	2	**1**	-1	0	**2**	0
$t=5$		**3**	2	**0**	-1		**2**	0
$t=6$		**2**	2	**-1**	-1		**2**	0
$t=7$		**1**	2	**-1**	0		**1**	1
$t=8$		0	**2**	-2	**2**		0	**2**
$t=9$			**1**		**1**			**1**
$t=10$			0		0			0

Table A.6: *Spare capacities in mode 3 (highlighted values are indicate feasibility of mode change; horizontal lines in the table indicate interval borders).*

Bibliography

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[ABRW92] N.C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: the deadline monotonic approach. *Proceedings of the IFAC/IFIP Workshop*, 1992.

[ARI] ARINC. http://www.arinc.com.

[ARI03] *ARINC 653-1 Avionics application software standard interface.* 2003.

[Aud91] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, The University of York, England, 1991.

[Aud93] N. C. Audsley. *Flexible Scheduling in Hard-Real-Time Systems.* PhD thesis, Department of Computer Science, University of York, 1993.

[AUT] AUtomotive Open System ARchitecture AUTOSAR. http://www.autosar.org.

[BB05] Enrico Bini and GiorgioC. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 2005.

[BB11] Alan Burns and Sanjoy Baruah. Timing faults and mixed criticality systems. In *Dependable and Historic Computing.* Springer Berlin Heidelberg, 2011.

[BB13] A. Burns and S. Baruah. Towards a more practical model for mixed criticality systems. In *Proceedings of the 1st International Workshop on Mixed Criticality Systems*, 2013.

[BBBR11] V. Brocal, P. Balbastre, R. Ballester, and I. Ripoll. Task period selection to minimize hyperperiod. In *16th Conference on Emerging Technologies Factory Automation (ETFA)*, 2011.

[BBD11] S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *32nd Real-Time Systems Symposium (RTSS)*, 2011.

[BBD+12a] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[BBD+12b] S. Baruah, V. Bonifaci, G. D'Angelo, Haohan Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 2012.

[BBD13] S Baruah, A Burns, and RI Davis. An extended fixed priority scheme for mixed criticality systems. *Proceedings of the 1st workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)*, 2013.

[BC06] Sanjoy K. Baruah and Samarjit Chakraborty. Schedulability analysis of non-preemtpive recurring real-time tasks. *20th International Parallel and Distributed Processing Symposium*, 2006.

[BD96] Alan Burns and Robert Davis. Choosing task periods to minimise system utilisation in time triggered systems. *Information Processing Letters*, (5), 1996.

[BF11] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *32nd Real-Time Systems Symposium (RTSS)*, 2011.

[BLS10] S. Baruah, Haohan Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.

[BNDPS02] A. Bar-Noy, V. Dreizin, and B. Patt-Shamir. Efficient periodic scheduling by trees. In *Proceedings of Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[BRH90] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems Journal*, 1990.

[BS97] Giorgio C. Buttazzo and Fabrizio Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *Proceedings. Third IEEE International Conference on Engineering of Complex Computer Systems*, 1997.

[But05] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer Science+Business Media, 2nd edition, 2005.

[BV08] Sanjoy Baruah and Steve Vestal. Schedulability analysis of sporadic tasks with mutliple criticality specifications. *Euromicro Conference on Real-Time Systems*, 2008.

[CB96] F. Cottet and J.-P. Babau. An iterative method of task temporal parameter adjustment in hard real-time systems. In *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems*, 1996.

[CL90] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 1990.

[CLTX13] Yao Chen, Qiao Li, Xiaojie Tu, and Huagang Xiong. Certification-cognizant real-time scheduling for mixed-criticality tasks in avionics system. In *32nd Digital Avionics Systems Conference (DASC)*, 2013.

[CSB90] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 1990.

[DB05] R.I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *26th IEEE International Real-Time Systems Symposium (RTSS)*, 2005.

[dNLR09] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[EY12] P. Ekberg and Wang Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[FK90] G. Fohler and Koza. Heuristic scheduling for distributed hard real-time systems. Technical report, Intitut für Technische Informatik, Technische Universität Wien, 1990.

[FLB01] Gerhard Fohler, Tomas Lennvall, and Giorgio Buttazzo. Improved handling of soft aperiodic tasks in offline scheduled real-time systems using total bandwidth server. *Proceedings of 8th IEEE International Conference on Emerging Technologies and Factory Automation*, 2001.

[Foh93] Gerhard Fohler. Changing operational modes in the context of pre run-time scheduling. *IEICE transactions on Information and Systems*, 1993.

[Foh94] Gerhard Fohler. *Flexibillity in Statically Scheduled Hard Real-Time Systems.* PhD thesis, Technische Universität Wien, 1994.

[Foh95] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. *In Proceedings of IEEE Real-Time Systems Symposium*, 1995.

[Foh97] Gerhard Fohler. Dynamic timing constraints – relaxing overconstraining specifications of real-time systems. In *Work-in-Progress Session, Real-Time Systems Symposium*, 1997.

[GB13] Patrick Graydon and Iain Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *Proceedings of the 1st International Workshop on Mixed Criticality Systems*, 2013.

[GESY11] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *32nd Real-Time Systems Symposium (RTSS)*, 2011.

[GM01] Joel Goossens and Christophe Macq. Limitation of the hyper-period in real-time periodic task set generation. In *Proceedings of the RTS Embedded Systems*, 2001.

[IF99] Damir Isovic and Gerhard Fohler. Handling sporadic tasks in off-line scheduled distributed real-time systems. *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, 1999.

[IF00] Damir Isovic and Gerhard Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. *In Proceedings of IEEE Real-Time Systems Symposium*, 2000.

[IF09] Damir Isovic and Gerhard Fohler. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Systems*, 2009.

[Joh92] L. A. Johnson. Do-178b: Software considerations in airborne systems and equipment certification. In *Radio Technical Commission for Aeronautics*, 1992.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 1986.

[JSM91] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemtpive scheduling of perdiodic and sporadic tasks. *Proceedings of Twelfth Real-Time Systems Symposium*, 1991.

[JZLP14] Mathieu Jan, Lilia Zaourar, Vincent Legout, and Laurent Pautet. Handling criticality mode change in time-triggered systems through linear programming. In *Ada User Journal, Proceedings of Workshop on Mixed Criticality for Industrial Systems (WMCIS 2014)*, 2014.

[KB03] Hermann Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 2003.

[KO87] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 1987.

[Kop11] Hermann Kopetz. *Real-Time Systems - Design principles for Distributed Embedded Applications.* Springer, 2nd edition, 2011.

[Kor85] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 1985.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

[LB03] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proceedings of 15th Euromicro Conference on Real-Time Systems*, 2003.

[LB10] Haohan Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *31st Real-Time Systems Symposium (RTSS)*, 2010.

[LBA98] Guiseppe Lipari, Giorgio Buttazzo, and Luca Abeni. A bandwidth reservation algorithm for multi-application systems. *Proceedings of Fifth International Conference on Real-Time Computing Systems and Applications*, 1998.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

[LSS87] J. Lehoczky, L. Sha, and J. Strosnider. Enhancing aperiodic responsiveness in a hard real-time environment. In *Real-Time Systems Symposium*, 1987.

[MEA$^+$10] M.S. Mollison, J.P. Erickson, J.H. Anderson, S.K. Baruah, and J.A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *10th International Conference on Computer and Information Technology (CIT)*, 2010.

[Mok83] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD thesis, Massachusetts Institute of Technology, 1983.

[NFK14] Mitra Nasri, G. Fohler, and Mehdi Kargahi. A framework to construct customized harmonic periods for real-time systems. In *26th Euromicro Conference on Real-time Systems*, 2014.

[PB98] P. Pedro and A. Burns. Schedulability analysis for mode changes in flexible real-time systems. In *Proceedings of 10th Euromicro Workshop on Real-Time Systems*, 1998.

[PK11] Taeju Park and Soontae Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2011.

[RBR12] I. Ripoll and R. Ballester-Ripoll. Period selection for minimal hyperperiod in periodic task systems. *IEEE Transactions on Computers*, 2012.

[SB94] Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. *Proceedings of Real-Time Systems Symposium*, 1994.

[SE04] Jan Staschulat and Rolf Ernst. Multiple process execution in cache related preemption delay analysis. In *Proceedings of the 4th ACM international conference on Embedded software*, 2004.

[SF13] Stefan Schorr and Gerhard Fohler. Integrated time- and event-triggered scheduling - an overhead analysis on the arm architecture. In *The 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Taipei, Taiwan, August 2013.

[SPB13] D. Socci, P. Poplavko, and S. Bensalem. Time-triggered mixed-critical scheduler. In *Proceedings of the 1st International Workshop on Mixed Criticality Systems*, 2013.

[SPBB13] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *25th Euromicro Conference on Real-Time Systems*, 2013.

[Spr90] Brinkley Sprunt. *Aperiodic Task Scheduling for Hard-Rreal-Time Systems*. PhD thesis, Carnegie Mellon University, 1990.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, (9), 1990.

[SRLK02] S. Saewong, R.R. Rajkumar, J.P. Lehoczky, and M.H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of 14th Euromicro Conference on Real-Time Systems*, 2002.

[SSL89] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1989.

[Sys] Mixed Critical Systems. http://www.nitrd.gov/about/blog/white_papers/20-mixed_criticality_systems.pdf.

[TG] TTA-Group. http://www.ttagroup.org/technology/tta.htm.

[Thu93] Sandra Ramos Thuel. *Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy.* PhD thesis, Carnegie Mellon University, 1993.

[TL94] S.R. Thuel and J.P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of Real-Time Systems Symposium*, 1994.

[VDHBL+12] M.M.H.P. Van Den Heuvel, R.J. Bril, J.J. Lukkien, D. Isovic, and G.S. Ramachandran. Rtos support for mixed time-triggered and event-triggered task sets. In *15th International Conference on Computational Science and Engineering (CSE)*, 2012.

[Ves07] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. *28th IEEE International Real-Time Systems Symposium*, 2007.

[Xu10] Jia Xu. A method for adjusting the periods of periodic processes to reduce the least common multiple of the period lengths in real-time embedded systems. In *International Conference on Mechatronics and Embedded Systems and Applications (MESA)*, 2010.

Glossary

Determinism

A physical system behaves *deterministically* if given an initial state at instant t and a set of future timed inputs, then the future states and the values and times of future outputs are entailed. [Kop11]. 2, 68, 106

Dynamic Priority Scheduling

In dynamic priority scheduling the priority of a task can change over time. We differentiate between fixed job priority and dynamic job priority. In the former, the priority of each job does not change whereas in the latter, also the priority of one job can change over time. 9, 18, 24, 26

Event-Triggered Paradigm

In the event-triggered paradigm actions are triggered by internal or external events. 2, 5, 6, 9, 10, 12, 14, 15, 18, 106, 110–115, 118, 120, 121, 123, 125, 128, 129, 132, 139–141, 155–157

Fixed Priority Scheduling

In fixed priority scheduling one priority value is assigned to each task and applied to all of its jobs. This priority does not change over the lifetime of the system. 7, 9, 10, 12, 15, 18, 24, 26, 30, 87, 89, 166

Hyper-period

The hyper-period is determined by the least common multiple of all periods in the system. It determines the point in time when a schedule of a synchronous task set repeats itself. 16, 23, 30, 31, 54–60, 63–65, 156, 159

Least Common Multiple

The least common multiple of two integer numbers is the smallest possible integer that is divisible by both numbers. 30, 54, 56–58

Real-Time System

Real-time systems are software and hardware systems in which the correctness of computations is not only defined by their values, but also the time when the results are provided. 2

Reserved Execution Time

The reserved execution time of a reservation is the amount of execution time which is reserved by one instance of the reservation task. 31–33, 35, 37, 38, 46, 47, 49, 50, 54

Time-Triggered Paradigm

In the time-triggered paradigm all actions are triggered by the progression of time. 1, 2, 4–7, 9, 11, 12, 14, 15, 17, 18, 21, 23, 30, 54, 63, 65, 67, 68, 70, 71, 82, 87, 88, 102, 103, 105–107, 110–113, 115, 118, 120, 139, 140, 156, 157, 159

Worst-Case Execution Time

The worst-case execution time of a task or job is an upper bound on the longest execution time needed by a processor to complete the task or job without interruption under specific assumptions considering all possible input data. 2–6, 10, 11, 14–19, 21–24, 26, 28, 30, 31, 33, 35, 37, 38, 46, 54, 67–70, 72, 82, 88, 99, 101, 102, 105–109, 111–115, 118, 120–125, 127–129, 132–136, 139–141, 143, 145, 147–153, 155–158, 166

Worst-Case Response Time

The worst-case response time of a task or job is the longest possible time between release and completion of the execution. 21, 23, 32, 46–51, 54, 155

Summary

Certification-Cognizant Mixed-Criticality Scheduling in Time-Triggered Systems

In real-time systems, timing constraints of jobs and tasks are ensured by real-time scheduling policies. Increasing processing power and reduced structure sizes in modern processing units led to the trend of integrating safety-critical and non-safety-critical functionalities on the same platform. These mixed-criticality systems resulted in new challenges in the development of efficient and effective scheduling algorithms. In dual-criticality systems, we use the criticality levels LO and HI. In this thesis, we focus on the development of such algorithms aiming at reduced certification costs and resource efficient scheduling of mixed-criticality systems.

Chapter 1 – Introduction

In this chapter, we present a brief introduction to the research areas which constitute the major body of this thesis. We introduce the concepts of real-time systems and show the properties and challenges of mixed-criticality systems. Additionally, the problem statement and contribution of this thesis are summarized. Finally, an overview of the structure of this thesis is given.

Chapter 2 – Related Work

In this chapter, we review related work in the area of mixed-criticality systems. We organize the presented approaches into event-triggered and time-triggered ones. The presented scheduling policies are classified into fixed and dynamic priority scheduling approaches.

Chapter 3 – Fundamentals

In this chapter, we show terms and notations used in this thesis. Furthermore, we present the task models of periodic, sporadic and aperiodic tasks. In addition, the job models of event-triggered and time-triggered jobs are shown. Next, we derive properties of the used task and job models. The Vestal mixed-criticality model which forms the basis for the presented mixed-criticality schedulers concludes the first section. In the second section, we classify real-time scheduling algorithms based on the priority assignment. Furthermore, we present the differences between event-triggered and time-triggered, and between preemptive and non-preemptive schedulers. A brief overview of scheduling policies of uni-processor and multi-processor systems and basic scheduling approaches used in this thesis conclude the scheduler classification. Finally, we define fundamental characteristics of schedules including the feasibility and schedulability of task and job sets.

Chapter 4 – Task Parameter Transformation

In this chapter, we describe a proactive method to transform sporadic tasks into periodic reservation tasks. The method aims at the inclusion of the event-triggered sporadic tasks, which are characterized by an unknown arrival time, into an offline scheduling process to reserve execution time for the runtime execution of them. The computation of the reservation tasks allows for an adaptation of the needed utilization to guarantee the sporadic tasks. Furthermore, the worst-case response time can be improved. We analyze the feasibility of reservation tasks based on their utilization including the switching overhead and based on the resulting worst-case response time of the sporadic tasks. Finally, we evaluate the applicability of the method by applying the procedures on periodic tasks to determine new periods to minimize the hyper-period of a time-triggered system. The evaluation shows that by an increase of the periodic task utilization, we can shorten the length of the hyper-period. A shorter hyper-period reduces the length of the schedule table and thus, reduces also the certification complexity of safety-critical time-triggered systems.

Chapter 5 – Time-Triggered Schedule Tables with Mode Changes

In this chapter, we show a search tree based algorithm to effectively and efficiently construct time-triggered schedule tables for dual-criticality systems. The algorithm consists of several steps to reduce the complexity of the scheduling process. First, we separate the demand in the LO-criticality case from the additional demand in the HI-criticality case by splitting HI-criticality jobs. As a consequence, we do not need to schedule jobs with two worst-case execution times but only jobs with one worst case execution time. Second, we introduce the concept of *leeway* which is based on the deadlines and on the demand of scheduled jobs. Using the leeway, we can detect paths in the search tree that result in infeasible schedules earlier and thus, reduce the complexity of the schedule search. Finally, we also use the leeway to guide our backtracking procedure. Instead of exploring all possible backtracking steps, we check only backtracking decisions for which the leeway does not indicate a resulting infeasible schedule. We conclude the chapter, evaluating the algorithm by a comparison with a fixed priority based scheduler presented by Baruah and Fohler [BF11]. The evaluation shows that our scheduler results in more feasible schedule tables which can be computed in a shorter amount of time.

Chapter 6 – Mixed-Criticality Slot-Shifting without Mode Changes

In this chapter, we present a dual-criticality scheduler based on slot-shifting [Foh95]. A major benefit of the presented approach is that we can use an already certified schedule table resolving possible complex constraints, e.g., by setting earliest start times and deadlines. Based on this schedule tables and the job parameters, we determine capacity intervals and spare capacities, i.e., amount of available resources within these intervals. We differentiate between two spare capacity types: LO-criticality and HI-criticality spare capacities, i.e., spare capacities based on all jobs with LO-criticality worst-case execution time and spare capacities based HI-criticality jobs with HI-criticality worst-case execution time. At runtime, a HI-criticality spare capacity value of zero indicates that we need to schedule a HI-criticality job to guarantee its HI-criticality timing requirements. As a consequence, we schedule the ready HI-criticality job with the earliest deadline. If the HI-criticality spare capacities are greater than zero, we select a job with earliest deadline from the set of all ready jobs. Additionally, we can use the spare capacities to perform an acceptance test for event-triggered activities, i.e., firm aperiodic and sporadic tasks. The test checks whether the aperiodic (sporadic) instance can be executed until its deadline without jeopardizing the execution of already guaranteed jobs. If this is not possible, the aperiodic (sporadic) instance is rejected. The main advantage of the approach is that we can react on the actual behavior of jobs. Early completions result in more available resources for other jobs which can be directly used. In case of a job showing HI-criticality behavior, we skip LO-criticality jobs only if it is necessary to guarantee HI-criticality jobs. As soon as resources are available, we continue scheduling LO-criticality jobs again.

Chapter 7 – Slot-Shifting with Generic Mode Changes

In this chapter, we extend the scheduler shown in Chapter 6 to mixed-criticality systems with more than two criticality levels. Based on slot-shifting [Foh95], we develop a scheduler to handle generic mode changes and show how we can use this approach to schedule mixed-criticality systems. The different criticality levels of the system are represented by modes including jobs with same or higher criticality level and the corresponding job parameters. In the offline phase of slot-shifting, we determine capacity intervals and spare capacities for each mode based on the jobs active in this mode. At runtime, we execute the current mode using EDF to select the next executing job and update the spare capacities of the other modes according to the executed job. We assume that the execution of a job in the current mode, which is active in several modes, is also valid in another mode. The spare capacities of a mode indicate whether we can switch to a mode without violation of the timing constraints of jobs in this mode. As a result, we can determine the feasibility of a mode change at every instant of the scheduler. On the contrary, the guaranteed execution of certified jobs with high criticality levels requires a mode change to a higher criticality level if otherwise the timing constraints are jeopardized. In conclusion, the scheduler can determine the feasibility of a requested mode change and can also trigger mode changes to ensure the guaranteed behavior of high criticality jobs. As a consequence, we can schedule mixed-criticality job sets with varying parameters, i.e., varying worst-case execution times and deadlines of specific criticality levels, and can also handle added and removed jobs in other modes. This flexibility comes at the cost of an increasing overhead with the number of modes.

Chapter 8 – Conclusions

In this chapter, we draw the conclusions and present the major contribution of this thesis. Concluding remarks complete the chapter.

Appendix A – Extended Results

In this chapter, we present further evaluation results of the effectiveness and efficiency of the presented period transformation method shown in Chapter 4. Furthermore, we show an extended evaluation including additional comparison results of the schedule table generation algorithm presented in Chapter 5. Finally, further examples of the slot-shifting algorithm with generic mode changes complete the appendix.

Zusammenfassung

Zertifizierungsorientierte Aufgabenplanung
mit gemischten Kritikalitätsstufen in zeitgesteuerten Systemen

In Echtzeitsystemen wird die Einhaltung zeitlicher Bedingungen von Programmen und Programminstanzen, im folgenden Tasks und Jobs genannt, durch Echtzeit-*Scheduling*[1]-Strategien sichergestellt. Steigende Rechenleistung und verkleinerte Strukturgrößen in modernen Prozessoren führten zu dem Trend der Integration sicherheitskritischer und sicherheitsunkritischer Funktionalitäten auf einer gemeinsamen Plattform. Diese sogenannten *Mixed-Criticality*[2] Systeme warfen neue Herausforderungen bei der Entwicklung effizienter und effektiver *Scheduling*-Algorithmen auf. In *Dual-Criticality* Systemen, d.h. Systemen mit lediglich zwei unterschiedlichen Kritikalitätsstufen, werden die Kritikalitätsstufen LO (engl. "low" für niedrig) und HI (engl. "high" für hoch) benutzt. Diese Dissertation konzentriert sich auf die Entwicklung solcher Algorithmen mit dem Ziel die Zertifizierungskosten zu reduzieren und die Ausführungsreihenfolge in *Mixed-Criticality* Systemen effizient zu planen.

Kapitel 1 – Einleitung

Dieses Kapitel umfasst eine kurze Einführung in das Forschungsgebiet, das den Hauptteil dieser Dissertation darstellt. Es werden die grundlegenden Konzepte von Echtzeitsystemen zusammen mit den Eigenschaften und Herausforderungen von *Mixed-Criticality* Systemen gezeigt. Außerdem wird die Problemstellung und der Beitrag dieser Arbeit zusammengefasst. Eine Übersicht der Gliederung dieser Dissertation schließt dieses Kapitel ab.

[1] *Scheduling* beschreibt den Prozess der Planung der Ausführungsreihenfolge von Tasks und/oder Jobs

[2] *Mixed-Criticality* (dt.: gemischte Kritikalitätsstufen) Systeme zeichnen sich durch Tasks und Jobs aus, deren Versagen zu unterschiedlich schweren Konsequenzen für das System führt. Die Konsequenzen reichen von keinen sicherheitsrelevanten Auswirkungen bis hin zum Totalausfall inklusive Zerstörung des Systems und möglichem Personenschaden.

Kapitel 2 – Themenbezogene Fachliteratur

Dieses Kapitel betrachtet relevante Fachliteratur im Bereich der *Mixed-Criticality* Systeme. Ihre Ansätze werden in ereignisgesteuerte und zeitgesteuerte Ansätze differenziert. Die gezeigten *Scheduling*-Strategien werden in Strategien mit festzugewiesenen und dynamischen Prioritäten eingeteilt.

Kapitel 3 – Grundlagen

Dieses Kapitel erläutert die genutzten Begriffe sowie deren Beschreibungen dieser Dissertation. Außerdem werden die Task-Modelle von periodischen, sporadischen und aperiodischen Tasks präsentiert. Zusätzlich werden die Job-Modelle ereignisgesteuerter und zeitgesteuerter Jobs gezeigt. Danach werden die sich ergebenden Eigenschaften der benutzten Task- und Job-Modelle gezeigt. Das von Vestal vorgeschlagene *Mixed-Criticality*-Modell, das die Basis der präsentierten *Mixed-Criticality Scheduling*-Algorithmen bildet, schließt den ersten Abschnitt ab. Im zweiten Abschnitt des Kapitels werden Echtzeit-*Scheduling*-Algorithmen, basierend auf der verwendeten Prioritätszuweisung, gruppiert. Danach werden die Unterschiede zwischen ereignisgesteuerten und zeitgesteuerten sowie zwischen unterbrechbaren und nicht unterbrechbaren *Scheduling*-Methoden gezeigt. Eine kurze Übersicht der *Scheduling*-Strategien für Systeme mit einem bzw. mehreren Prozessoren und grundlegende *Scheduling*-Ansätze schließen die Gruppierung der *Scheduling*-Strategien ab. Die Definition charakteristischer Eigenschaften von *Scheduling*-Plänen, inklusive der Ausführbarkeit und Planbarkeit der Task- und Job-Sets, schließt das Kapitel ab.

Kapitel 4 – Task-Parameter-Transformation

Dieses Kapitel beschreibt eine proaktive Methode um sporadische Tasks in periodische Reservierungstasks zu transformieren. Diese Methode zielt auf die Einbeziehung der ereignisgesteuerten, sporadischen Tasks, die durch eine unbekannte Ankunftszeit gekennzeichnet sind, in den offline durchgeführten *Scheduling*-Prozess ab. Die Reservierungstasks garantieren Ausführungszeit für die Laufzeitausführung der sporadischen Tasks. Eine Anpassung der erforderlichen *Utilization* der Reservierungstasks, d.h. dem Anteil an Ausführungszeit auf dem Prozessor zur Garantie der sporadischen Tasks, ist während der Bestimmung ihrer Parameter möglich. Außerdem kann die längst mögliche Antwortzeit der sporadischen Tasks durch diese Anpassung verbessert werden. Im Anschluss wird die Ausführbarkeit der Reservierungstasks, basierend auf ihrer *Utilization* unter Beachtung der *Scheduling*-Kosten und auf der sich ergebenden längst möglichen Antwortzeit der sporadischen Tasks, analysiert. Schließlich wird die Anwendbarkeit der Methode, durch das Erzeugen neuer Perioden für periodische Tasks um das kleinste gemeinsame Vielfache (kgV) der Perioden eines zeitgesteuerten Systems zu minimieren, evaluiert. Das kgV der Perioden bestimmt die Länge der *Scheduling*-Tabelle. Die Auswertung der Ergebnisse zeigt, dass durch einen Anstieg der *Utilization* der periodischen Tasks das kgV der Perioden verkleinert werden kann. Die sich dadurch ergebende

Verkürzung der Scheduling-Tabelle reduziert die Komplexität der Zertifizierung von sicherheitskritischen, zeitgesteuerten Systemen.

Kapitel 5 – Zeitgesteuerte Scheduling-Tabellen mit Moduswechseln

Dieses Kapitel zeigt einen suchbaumbasierten Algorithmus zur effektiven und effizienten Konstruktion von *Scheduling*-Tabellen für *Dual-Criticality* Systeme. Der Algorithmus besteht aus drei Schritten, um die Komplexität des *Scheduling*-Prozesses zu reduzieren. Im ersten Schritt wird der Rechenzeitbedarf von HI-*Criticality* Jobs im LO-*Criticality*-Fall vom zusätzlich benötigten Rechenzeitbedarf im HI-*Criticality*-Fall separiert. Das Ergebnis sind dann Jobs, die nur noch durch eine Worst-Case Execution Time[3] (WCET) charakterisiert sind. Dadurch müssen nicht mehr Jobs mit zwei WCETs geplant werden. Im zweiten Schritt wird der *leeway* eingeführt, der eine charakteristische Größe eines Scheduling-Slots darstellt und auf den Ausführungsfristen und dem Rechenzeitbedarf des in diesem Slot geplanten Jobs basiert. Durch den *leeway* können Pfade im Suchbaum, die zu nicht ausführbaren *Scheduling*-Plänen führen, früher erkannt und somit die Komplexität des *Schedulings* reduziert werden. Im letzten Schritt wird der *leeway* genutzt, um beim *Backtracking* nicht alle Schritte überprüfen zu müssen und statt dessen nur noch Entscheidungen in Slots zu überprüfen, deren *leeway* nicht einen daraus resultierenden, nicht ausführbaren *Scheduling*-Plan, ergibt. Das Kapitel wird mit einer Evaluierung des Algorithmus, durch einen Vergleich mit der Methode von Baruah und Fohler [BF11], die auf festzugewiesenen Prioritäten basiert, abgeschlossen. Das Ergebnis der Evaluierung zeigt, dass der Algorithmus, der in dieser Dissertation vorgestellt wurde, mehr ausführbare *Scheduling*-Pläne in kürzerer Zeit liefert.

Kapitel 6 – Mixed-Criticality Slot-Shifting ohne Moduswechsel

Dieses Kapitel stellt einen *Dual-Criticality* Algorithmus, der auf dem von Fohler gezeigten *Slot-Shifting* Algorithmus [Foh95] basiert, vor. Ein Vorteil dieses Ansatzes ist die Möglichkeit bereits zertifizierte Scheduling-Tabellen nutzen zu können. Diese Tabellen lösen etwaige komplexe Einschränkungen, z.B. durch früheste Startzeiten und Ausführungsfristen, auf. Basierend auf diesen *Scheduling*-Tabellen werden Kapazitätsintervalle und *Spare Capacities*, die ein Maß für die verfügbaren Ressourcen innerhalb dieser Intervalle darstellen, bestimmt. Es werden zwei Typen dieser *Spare Capacities* unterschieden: LO-*Criticality Spare Capacities*, die auf allen Jobs mit LO-*Criticality* WCET basieren, und HI-*Criticality Spare Capacities*, die auf HI-*Criticality* Jobs mit ihren HI-*Criticality* WCETs basieren. Zur Laufzeit zeigt ein *Spare-Capacity*-Wert von Null an, dass ein HI-*Criticality* Job als nächstes ausgeführt werden muss, damit dieser seine garantierten HI-*Criticality* Anforderungen erfüllen kann. Der Job mit der frühesten Ausführungsfrist wird dann als nächstes zur Ausführung ausgewählt. Wenn die HI-*Criticality* *Spare-Capacity*-Werte größer als Null sind, dann wird ein ausführbarer Job mit der frühesten Ausführungsfrist aus dem gesamten Job-Set ausgewählt. Außerdem werden die *Spare Capacities* genutzt, um einen Test zur Annahme von ereignisgesteuerten Aktivitäten, d.h. sporadische und aperiodische Tasks mit Ausführungsfristen, durchzuführen.

[3]Die Worst-Case Execution Time ist eine Abschätzung für die maximal benötigte Ausführungszeit.

Der Test überprüft, ob die Annahme des aperiodischen (sporadischen) Jobs die Ausführung bereits garantierter Jobs gefährdet. Außerdem muss der aperiodische (sporadische) Job vollständig ausführbar bis zu seiner Ausführungsfrist sein. Wenn diese beiden Bedingungen nicht erfüllt sind, dann wird der aperiodische (sporadische) Job abgelehnt. Der Hauptvorteil dieses Ansatzes ist die Möglichkeit auf das tatsächliche Verhalten der Tasks zur Laufzeit zu reagieren. Eine frühere Beendigung der Ausführung ermöglichen die Nutzung freiwerdender Ressourcen für andere Tasks. Falls ein Job HI-*Criticality*-Verhalten zeigt, wird die Ausführung von LO-*Criticality* Jobs, nur wenn dies notwendig ist, übersprungen, um HI-*Criticality* Jobs zu garantieren. LO-*Criticality* Jobs werden wieder ausgeführt sobald Ressourcen für diese zur Verfügung stehen.

Kapitel 7 – Slot-Shifting mit allgemeinen Moduswechseln

In diesem Kapitel wird eine Erweiterung des *Slot-Shifting*-Algorithmus aus Kapitel 6 gezeigt. Die Erweiterung umfasst *Mixed-Criticality* Systeme mit mehr als zwei Kritikalitätsstufen. Der *Scheduling*-Algorithmus behandelt allgemeine Moduswechsel und kann außerdem in *Mixed-Criticality* Systemen verwendet werden. Die verschiedenen Kritikalitätsstufen des Systems werden durch Modi dargestellt, die Jobs gleicher oder höherer Kritikalitätsstufe beinhalten. In der Offline-Phase von *Slot-Shifting* werden Kapazitätsintervalle und *Spare Capacities*, basierend auf den aktiven Jobs des Modus, bestimmt. Zur Laufzeit wird der als nächstes auszuführende Job mittels EDF[4] ausgewählt. Außerdem werden die *Spare Capacities* aller Modi entsprechend dem ausgeführten Job aktualisiert. Die *Spare Capacities* eines Modus bestimmen die Ausführbarkeit eines Moduswechsels in den entsprechenden Modus. Daraus lässt sich die Ausführbarkeit von Moduswechseln für jeden Zeitpunkt während des *Scheduling*-Prozesses bestimmen. Im Gegensatz dazu können Zeitpunkte bestimmt werden, zu denen ein Moduswechsel nötig ist, um die Anforderungen von Jobs mit höherer Kritikalitätsstufe zu garantieren. Dadurch kann der *Scheduling*-Algorithmus die Ausführbarkeit von Moduswechseln überprüfen und Moduswechsel veranlassen, um die Ausführungsfristen der Jobs mit hoher Kritikalitätsstufe einzuhalten. Als Ergebnis dieser Möglichkeiten können *Mixed-Criticality* Jobs mit veränderlichen Parametern, d.h. veränderlichen WCETs und Ausführungsfristen in spezifischen Kritikalitätsstufen, geplant werden. Außerdem ist es möglich Jobs hinzuzufügen und zu entfernen auch in anderen Modi. Diese Flexibilität wird erreicht durch erhöhte *Scheduling*-Kosten, die mit Anzahl der Modi steigen.

Kapitel 8 – Fazit

Dieses Kapitel präsentiert das Fazit und den Hauptbeitrag dieser Dissertation. Abschließende Bemerkungen vervollständigen dieses Kapitel.

Anhang A – Erweiterte Ergebnisse

In diesem Kapitel werden weitere Evaluierungsergebnisse, die die Effektivität und Effizienz der Perioden-Transformationsmethode belegt, gezeigt. Außerdem wird eine erwei-

[4] *Scheduling*-Algorithmus, der den nächsten Job durch die früheste Ausführungsfrist auswählt.

terte Evaluierung, inklusive zusätzlichen Vergleichen des *Scheduling*-Tabellen-Konstruktionsalgorithmus, dargestellt. Schließlich vervollständigen weitere Beispiele des *Slot-Shifting*-Algorithmus mit allgemeinen Moduswechseln den Anhang.

JENS THEIS • jtheis@eit.uni-kl.de

LEBENSLAUF

PERSÖNLICHE DATEN

Staatsangehörigkeit:	deutsch
Familienstand:	ledig
Lehrstuhladresse:	Lehrstuhl für Echtzeitsysteme TU Kaiserslautern Postfach 3049 67653 Kaiserslautern

HOCHSCHULBILDUNG

05/2010 – heute

Technische Universität Kaiserslautern
Doktorand bei Prof. Dipl.-Ing. Dr. Gerhard Fohler
Fachbereich Elektro- und Informationstechnik, Lehrstuhl Echtzeitsysteme

- *Dissertation: "Certification-Cognizant Mixed-Criticality Scheduling in Time-Triggered Systems"*
- *Reviewer bei internationalen Wissenschaftskonferenzen im Bereich Real-Time Systems*
- *Lehrauftrag für das Labor "Real-Time Systems Lab" im Wintersemester 2011/12*
- *Vorlesungen und Übungen für die Kurse "Operating Systems", "Real-Time Systems I" und "Real-Time Systems II"*
- *Betreuung des Seminars "Real-Time Systems"*
- *Betreuung studentischer Forschungsarbeiten*

10/2004 – 01/2010

Technische Universität Kaiserslautern
Diplom in Elektro- und Informationstechnik

- *Diplomarbeit: "A Generic Berlekamp-Massey Hardware Implementation for a Flexible High-Throughput BCH Decoder"*
- *Studienarbeit: "Design and Implementation of a Wireless Sensor Network for Breathing Protection Monitoring"*

BERUFLICHE ERFAHRUNG

06/2004 – 09/2004

Praktikum in der Lehrwerkstatt für Feinmechanik und Elektrotechnik
Fresenius Medical Care, St. Wendel, Saarland

- *Spanabhebende und spanlose Formung und weitere industrielle Techniken*
- *Elektrotechnische Arbeitsverfahren*
- *Betrieblicher Einsatz moderner informationstechnischer Mittel und Verfahren*

06/2003 – 08/2003

Aushilfe im Rasterelektronenmikroskop-Labor
Fresenius Medical Care, St. Wendel, Saarland

- *Probenvorbereitung*
- *Probenanalyse mittels Rasterelektronenmikroskop*

AUSBILDUNG

08/1994 – 06/2003 Allgemeine Hochschulreife am Gymnasium Wendalinum, St. Wendel, Saarland

SPRACHEN

Deutsch:	Muttersprache
Englisch:	Fließend
Französisch:	Grundkenntnisse

VERÖFFENTLICHUNGEN

12/2013	Jens Theis und Gerhard Fohler "Mixed Criticality Scheduling in Time-Triggered Legacy Systems" *Proceedings of the 1st International Workshop on Mixed Criticality Systems at Real-Time Systems Symposium, Vancouver, Kanada*
12/2013	Jens Theis, Gerhard Fohler und Sanjoy Baruah "Schedule Table Generation for Time-Triggered Mixed Criticality Systems" *Proceedings of the 1st International Workshop on Mixed Criticality Systems at Real-Time Systems Symposium, Vancouver, Kanada*
09/2011	Jens Theis und Gerhard Fohler "Transformation of Sporadic Tasks for Offline Scheduling with Utilization and Response Time Trade-offs" *Proceedings of the 19th International Conference on Real-Time and Network Systems, Nantes, Frankreich*

KENNTNISSE

Betriebssysteme:	Microsoft Windows, Linux
Programmiersprachen:	C/C++
Office-Programme:	Microsoft Word, Excel, PowerPoint, Outlook
Grafikprogramme:	Microsoft Visio, Corel Draw
Sonstige Programme:	The MathWorks Matlab, LaTex-Textsatz