TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Informationstechnik im Maschinenwesen

Robust Scheduling of Real-Time Applications on Efficient Embedded Multicore Systems

Michael Deubzer

Vollständiger Abdruck der von der Fakultät für Maschinenwesen der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. rer. nat. Tim C. Lüth
Prüfer der Dissertation:
1. Univ.-Prof. Dr.-Ing. Birgit Vogel-Heuser
2. Prof. Dr. rer. nat. Jürgen Mottok
   Hochschule für Angewandte Wissenschaften Regensburg
3. Univ.-Prof. Dr.-Ing. Markus Siegle
   Universität der Bundeswehr München

Die Dissertation wurde am 03.02.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Maschinenwesen am 06.09.2011 angenommen.

Abstract

The increasing demand of computing capacity in embedded real-time systems, especially in the automotive powertrain domain, has been satisfied so far by increased processor frequency of singlecore processors. Due to thermal dissipation, increasing processor frequency is technically limited. Multicore processors provide additional computing capacity at constant or even reduced processor frequency. However, most embedded real-time systems have tight real-time, efficiency, and robustness requirements, resulting respectively from time-critical interaction with other devices, high piece numbers, and realization of control functions. Multicore processor real-time scheduling approaches, namely partitioning heuristics for existing scheduling algorithms for singlecore processors and global scheduling algorithms for multicore processors, have to fulfill these requirements.

This work discusses multicore processor real-time scheduling approaches as well as examination approaches of real-time, efficiency, and robustness properties for complex multitasking applications (task sets) on multicore processors in embedded real-time systems.

Related work on multicore processor real-time scheduling mainly considers simplified task sets, i.e. periodic or sporadic task sets with constant task execution times and implicit deadlines. This work examines multicore processor real-time scheduling approaches for more complex practical task sets, common in automotive systems, with heterogeneous task activation patterns, variable task execution times, and hard or soft explicit task deadlines. The work considers both global scheduling algorithms and partitioning heuristics for local scheduling algorithms for symmetric multicore processors.

For this purpose, a task set model for complex multitasking applications, e.g. in the field of automotive control units, is presented, which extends numerous existing task set models: (I) by a description of task activation sources, specifying the modification of task activation times for a sub group of tasks in a task set; (II) by a probabilistic description of task section execution times, splitting task execution time in blocks in order to model non-preemptive scheduling. Furthermore, the global multicore processor real-time scheduling algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2} are proposed. Both algorithms are able to schedule the presented task sets in a way that the multicore processor is used very efficiently while at *P-ERfair-PD*^{2} at the same time the compliance of task deadlines is robust against perturbation of task set properties. For schedulability evaluation, a simulation-based examination approach with mechanisms to approximate the worst-case response time of tasks is proposed. Finally, a sensitivity analysis approach for embedded real-time systems with a probabilistic description of task sets is proposed. The approach allows to compare scheduling algorithms according to the mentioned properties and allows a sensitivity analysis of system metrics, by considering task sets variations over the lifetime of a real-time system. The application of these contributions is shown in case studies based on task set data sets of automotive applications.

Kurzfassung

Durch den stetig steigenden Bedarf an Rechenkapazität in eingebetteten Systemen, besonders im Bereich von Motorsteuerungen für Automobilsysteme, und der Begrenzung der maximalen Taktfrequenz von Singlecore und Multi–Prozessor Systemen, werden Multicore-Prozessoren für den Einsatz in diesen Systemen zunehmend interessanter. Aufgrund eines hohen Kostendrucks im Bereich von eingebetteten Systemen ist die effiziente Auslastung der Prozessoren hier besonders wichtig. Zudem fordern zeitkritische Interaktionen eine hohe Robustheit bei der Einhaltung von Echtzeitanforderungen gegenüber Störungen der zeitlichen Eigenschaften der Anwendung. Sowohl die Effizienz als auch die Robustheit von eingebetteten Echtzeitsystemen sind durch den Scheduling–Algorithmus maßgeblich beeinflusst.

Der Fokus dieser Arbeit liegt auf Scheduling–Algorithmen für Echtzeitanwendungen sowie Verfahren zur Untersuchung des Echtzeitverhaltens von eingebetteten Multicore–Prozessor Systemen.

Bisherige Ansätze untersuchen vorwiegend Scheduling–Algorithmen unter der Annahme eines vereinfachten Modells für die Echtzeitanwendung (Taskset), periodische oder sporadische Aktivierungen, konstante Tasklaufzeiten und implizite Deadlines. Diese Arbeit untersucht Scheduling–Algorithmen für Taskset, welche heterogene Aktivierungs–Muster aufweisen, variable Laufzeiten besitzen und harten oder weichen Echtzeitanforderungen unterliegen. Hierbei werden sowohl Partitionierungsheuristiken für lokalen Scheduling–Algorithmen als auch globale Scheduling–Algorithmen betrachtet.

Zu diesem Zweck stellt diese Arbeit eine Erweiterung für Tasksets um eine Beschreibung der zeitgleichen Änderung von Aktivierungsmustern einer Teilmenge der Tasks eines Tasksets und eine Beschreibung von variablen Laufzeiten für Task–Sektionen vor. Hierdurch ist es z.B. möglich Motorsteuerungen detailliert zu beschreiben und kooperative Scheduling Algorithmen zu untersuchen. Für das erweiterte Taskset werden die zwei globalen Scheduling–Algorithmen *Partly-Pfair-PD*^{2} und *P-ERfair-PD*^{2} vorgestellt. Diese ermöglichen es heutige komplexe Echtzeitanwendungen auf Multicore–Prozessoren hocheffizient auszuführen. Dabei weist *P-ERfair-PD*^{2} zusätzlich eine hohe Robustheit bei der Einhaltung von Echtzeitanforderungen unter Störungen des Tasksets auf. Für die Untersuchungen der Echtzeiteigenschaften eingebetteter Systeme wird ein simulationsbasiertes Verfahren für globale Scheduling–Algorithmen mit dynamischer Prioritätsvergabe beschrieben, welches sich ebenfalls auf andere Algorithmen, wie z.B. lokale Scheduling–Algorithmen, anwenden lässt. Zudem wird ein Verfahren vorgestellt, welches es ermöglicht Scheduling–Algorithmen unter der Berücksichtigung der Variation von Eigenschaften des Tasksets über die Lebensdauer eines eingebetteten Systemen zu bewerten und zu vergleichen. Die praktische Anwendbarkeit wird an verschiedenen Fallstudien, basierend auf Daten existierender automobiler Systeme, evaluiert.

Acknowledgment

This work was written in the context of the BMBF founded project $DynaS^3$ („Dynamic software architectures in electronic control units in automotive systems with consideration of requirements for functional safety“). $DynaS^3$ was a cooperative research project between the University of Applied Sciences Regensburg, the Continental Automotive GmbH, the Technical University Munich, the Bundeswehr University of Munich, and the University Magdeburg. I have been fortunate to have been mentored by people who have both inspired and challenged me.

Foremost, I would like to thank Prof. Dr. rer. nat. Jürgen Mottok for educating and guiding me over the past years with great care, enthusiasm, and patience. I am much obliged to Prof. Dr.-Ing. Birgit Vogel-Heuser for the great support and for giving me the opportunity to write this thesis in her research group at the Technical University Munich. I thank my further advisors Univ. Prof. Dr.-Ing. Markus Siegle and Prof. Dr.-Ing. Frank Schiller. I am deeply thankful for their help and the feedback they gave me over the years. I am deeply grateful to my advisors Dr. Ulrich Margull and Dr. Michael Niemetz. We had a lot of inspiring discussions and they taught me in research work and scientific writing.

I thank my research colleague Martin Hobelsberger for the inspiring discussions and the great time we had as PhD students at the LaS^3. I am indebted to all colleagues of the LaS^3, who have made my time here very enjoyable and also helped me countless times. In particular, I would like to thank Michael Schorer and Michael Steindl. Furthermore, I want to thank my students, especially Stefan Schmidhuber and Max Raith, for supporting me at the development of the model generation environment and the cluster computing cloud.

So many friends have supported me over the last years (I apologize to any friend I may have omitted): Jennifer Bystry, Anton Achatz, Stefan Till, Katharina Fregin, Andrea Weber, Marcus Kandelbinder, Michael Graupner, Jasmin Bystry, Heike Schwarz, Christian Köhler, and Frank Jahn.

I am fortunate to have a loving and supportive family, who put great trust in me. All my life they believed in me and gave me a strong motivation.

January 2011,
Regensburg

Contents

Abstract i
Kurzfassung ii

1 Introduction **1**
1.1 Motivation 2
1.2 Contribution 4
1.3 Structure of the Work 6

I Preparation and Related Work **8**

2 Real-Time System Model **9**
2.1 Fundamentals 9
2.2 Real-Time System Model 10
2.2.1 Task Set 12
2.2.2 Processor 16
2.2.3 Scheduler 19
2.2.4 Time Events 19
2.3 Related Work 20
2.3.1 Models for Demand of Execution Time 21
2.3.2 Models for the Execution Time 26
2.3.3 Classification of Deadline Bounds 28
2.4 Characteristics of Task Set Models 29

3 Real-Time Scheduling **31**
3.1 Fundamentals 31
3.1.1 Definitions 31
3.1.2 Classification of Multicore Scheduling Algorithms 32
3.1.3 Scheduling Model 35
3.2 Related Work on Singlecore Scheduling 36
3.2.1 Task-Fix Priority Scheduling 36
3.2.2 Job-Fix Priority Scheduling 38
3.2.3 Dynamic Priority Scheduling 39

3.2.4 Cooperative Scheduling 39
3.3 Related Work on Multicore Scheduling 40
3.3.1 Local Scheduling 41
3.3.2 Global Scheduling 43

4 Real-Time System Examination **52**
4.1 Fundamentals 52
4.1.1 Definitions 52
4.1.2 Real-Time Metrics 55
4.2 Methods for Schedulability Examination 57
4.2.1 Response-Time Analysis 57
4.2.2 Real-Time Calculus 58
4.2.3 Model-Checking Approach 59
4.2.4 Simulation Approach 59
4.3 Anomalies of Multicore Schedulability Analysis 60
4.3.1 Examples 61

II Contribution **63**

5 Focus of Contribution **64**

6 Multiple Time Base Task Set Extension **66**
6.1 Multiple Time Base Extension 66
6.1.1 Problem Formulation 67
6.1.2 Extension 69
6.2 Probabilistic Execution Time Model 73
6.2.1 Discrete Probability Function 74
6.2.2 Weibull Pobability Function 74

7 Global Multicore Scheduling **78**
7.1 Partly Pfair Approach 78
7.1.1 Drawback of Pfair scheduling 78
7.1.2 Partly Pfair 79
7.2 Partly Early Release Fair Extension 82

8 Simulation-Based Multicore Real-Time Examination **84**
8.1 Discrete-event Simulation 84
8.2 Architectural Model 85
8.2.1 Stimulation Subsystem 86
8.2.2 Software Subsystem 86
8.2.3 Hardware Subsystem 87

8.2.4 Operating System Subsystem 88
8.3 Behavioral Model 88
8.3.1 Simulation Sequencer 90
8.3.2 Stimulation Subsystem 91
8.3.3 Software Subsystem 93
8.3.4 Hardware Subsystem 99
8.3.5 Operating System Subsystem 104
8.4 A Metric for Real-Time Examination 106
8.5 Approximation of Bounds for Schedulability 107
8.6 Technical Implementation 109

9 Sensitivity Analysis of Probabilistic System Models 110
9.1 Task Set Parameter Variations 110
9.2 Probabilistic System Model 111
9.2.1 Probability of Task Set Parameters 112
9.2.2 Probability of Task Quantity 112
9.3 Monte-Carlo Randomization 113
9.4 Examination of Characterization Metrics 116
9.5 Classification of System Models 116
9.6 Statistical Evaluation 116

III Case Studies 120

10 Execution of Case Studies 121
10.1 Automotive Systems 121
10.2 Case Study I: A Quadcore System 123
10.3 Case Study II: Porting from Singlecore to Dualcore Processors 126
10.4 Case Study III: Robustness Analysis 132

11 Discussion 134
11.1 Local Scheduling with Bin-Packing Partitioning 134
11.2 Global Scheduling Algorithms 135
11.3 Simulation-Based Multicore Real-Time Examination 136

IV Conclusion and Future Work 137

12 Summary 138

13 Further Work 140

Bibliography 141

Index 153

List of Figures 155

List of Tables 159

V Appendix 160

A Pseudo-Code of Scheduling Algorithms 161

A.1 Algorithm EDF 161

A.2 Algorithm *Partly-Pfair-PD*^{2} 162

A.3 Algorithm *P-ERfair-PD*^{2} 163

Chapter 1

Introduction

„The only reason for time is so that everything doesn't happen at once."
-Albert Einstein

Real-time applications in embedded systems have logical and temporal requirements. Logical requirements imply that the application has to produce correct results. Temporal requirements imply that the application has to produce the results within correct time windows. Additionally, embedded systems have a high demand on efficiency and robustness.

Efficiency is necessary to develop a commercial solution which uses resources economically. Often embedded systems have a high production number, therefore saving resources has two benefits. Firstly, the production wastes less environmental resources like material and energy. Secondly, piece costs decrease, which results in lower production costs and a competitive product.

Robustness is necessary because embedded systems are often integrated in the human environment and take over safety-critical functions. In the case of a car crash, for example, the airbag control system of an automotive system has to react within a few milliseconds by preparing sensor data and switch a current, which ignites a combustible material in order to fill the airbag with gas. This function has to be correct, logically and temporally, also when perturbations stress the system, e.g. when a sensor delivers wrong signals (logical perturbation) or other functions request non-specified processor time (temporal perturbation). For this reason, a robustness consideration is necessary to evaluate the behavior in case of perturbations.

This work considers the temporal part of these requirements for the **scheduling** problem of real-time **applications** on **embedded multicore processors**. The scheduling problem inquires how to assign a real-time application to a number of processors in a way that all temporal requirements are fulfilled.

The following sections of this chapter summarize benefits of multicore processors for the embedded domain and the necessity of appropriate real-time scheduling algorithms, open topics on real-time scheduling algorithms for multicore processor, the contribution of this work to these topics, and the structure of this dissertation.

1.1 Motivation

In embedded real-time systems, the amount and the interaction of software applications has increased rapidly in the last decades. In current automotive systems, up to 100 electrical control units (ECUs) supply computing capacity for the realization of movement, safety, comfort, and entertainment functions. For further functionalities, additional computing capacity is required. However, the mechanisms which were used to increase the computing capacity in the last decades, e.g. raising of processor frequency or using additional processors or ECUs, are either technically limited or not cost-effective. From the current point of view, multicore processors are the only economic way to supply the required computing capacity for automotive systems of the next decades. In other domains of embedded systems, e.g. mobile devices or robotics, the situation is similar.

When using multicore processors, the computing capacity can be increased without raising the processor frequency. Raising the processor frequency causes a quadratic increase of power consumption, which in turn generates heat. Heat has numerous drawbacks in embedded systems: it wastes energy, it requires costly cooling mechanisms, it decreases reliability due to increasing failure rates of semiconductors [PLW90], and it shortens the longevity of the device by aging effects [Fre09]. Additionally, frequency rising results in higher electromagnetic perturbations, which may infringe the emission restriction of the electromagnetic compatibility (EMC) requirement [DRS06] or makes better shielding necessary.

A further benefit of multicore processors is the reduction of costs for caching mechanisms, due to the reduction of the lag between memory access rates and processing rates. When the increase of processor frequency stops and memory access rates still increase, both rates converge and caching mechanisms to compensate this lag will not be necessary anymore.

Furthermore, the high computing capacity of multicore processors allows to merge ECUs, which has a benefit in car cost factor due to a reduction of the redundancy of many components. This is beneficial, because besides the ECU costs, which include case, board, power supply, peripherals, and many more costs, additional signal and power supply cables produce costs and additional weight, which increase the energy consumption of the vehicle.

The application software of an embedded system is implemented in a multi-tasking architecture. For efficient usage of multicore processors in embedded systems, appropriate scheduling algorithms are necessary. A scheduling algorithm has to schedule the multi-tasking application in a way that all real-time requirements are met. Furthermore, the multicore processor has to be used very efficiently and the compliance of real-time requirements has to be robust against perturbations of the temporal properties of the application, meaning deviations of the specified temporal properties of the application.

A scheduling algorithm for an embedded multicore system has to assign the application with real-time requirements to the cores of the processor. In singlecore processor systems, mostly static priority based or job-fix priority based algorithms are applied. Static priority algorithms schedule tasks by a system-developer defined priority or by a priority which is derived by a schedule policy, e.g. Rate-Monotonic (RM). Job-fix priority based scheduling algorithms derive a prioritization of tasks by scheduling policies, using temporal properties of the task set. Earliest-Deadline-First (EDF), for example, assigns the task with the shortest deadline the highest priority. By reason of the additional dimension of scheduling at multicore processors (time and place), scheduling algorithms for singlecore processors are no longer optimal for the multicore processor case and perform poorly in regard of processor utilization and compliance of real-time requirements (see Section 3.3).

On the one hand a high processor utilization is important in embedded systems, because there is a high demand of additional functionalities which is often limited by the available resources, namely processor time and memory. Furthermore, it is desired to decrease processor frequency because of the mentioned reasons, which in turn results in higher execution times and higher processor utilizations. On the other hand embedded systems have a high cost pressure due to a high piece number which tends to decrease resource capacities. For these reasons, processor utilizations in the range of $80 - 95\%$ are common in automotive powertrain systems for certain operation modes [MNW10].

In order to fulfill these demands, multicore scheduling algorithms need information about the requested execution time. For this purpose a model of the temporal properties of the multi-tasking application is made and real-time requirements are specified. This model is called task set. Based on this task set, the scheduling algorithm constructs a schedule, which defines how to assign the multi-tasking application to the cores of a multicore processor.

In this work, a task set specifies the request behavior and the execution time of the tasks of a multi-tasking application. Depending on the kind of embedded system, the temporal properties have different characteristics. Liu and Layland [Liu69] introduced one of the first models of real-time systems, which describes a periodic activation of tasks. A magnitude of extensions have been presented in the meantime (see section 2.3 for an overview). However in today's embedded systems, especially in automotive powertrain

systems, tasks have activation patterns which depend on a trigger source which itself can have an individual behavior. This can result in a complex request pattern. For such kind of task sets less publications are available, mainly due to necessary simplifications of task set properties in order to allow to prove the compliance of real-time requirements.

Since scheduling algorithms construct the schedule based on the temporal properties, the effects of perturbations of the specified temporal properties have to be considered. For the singlecore processor case, these kinds of perturbations are less important, because most scheduling algorithms don't require this information due to static priority scheduling. However, for multicore processor systems this information is essential and therefore a high robustness according perturbations is desired.

1.2 Contribution

This dissertation addresses to the following questions:

Question 1:
„How can software applications with real-time requirements and complex temporal properties be scheduled on embedded multicore systems in a way that all real-time requirements are met, the multicore processor is used highly efficiently, and the real-time requirements are also met at perturbations of temporal properties?„

Question 2:
„How can these properties be analyzed for multicore processors?„

For this purpose, this work assumes that a system has a symmetric multicore processor where each core of the multicore processor is identical to all other cores. Furthermore, asynchronous inter-task communication and data access is assumed and therefore delays through data dependencies are not considered.

The applicability of a real-time scheduling algorithm depends on the temporal properties of the multi-tasking application (task set model) and the processor. For singlecore processor systems, optimal scheduling algorithms (see an overview in Section 3.2) have been introduced for different task set models, describing sporadic or different periodic task activation patterns. Different clocks in a real-time system and the problem of synchronizing these clocks has been intensively discussed and is state of the art (e.g. standardized by IEEE 1588). However, there are less publications which consider the activation of tasks from clock trigger sources, which results in frequency and phase difference of clock time in relation to global time. This scenario is typical for powertrain ECUs of the automotive domain, where the execution of parts of the application depends on the crankshaft position and the execution of other parts of the application is triggered periodically.

An annotation of this information would allow to decrease the level of pessimism in scheduling examination, because otherwise assumptions have to be made for the missing information of the exact activation.

Therefore, this work presents the multiple time base (MTB) extension, which can be applied to a wide range of previously introduced task set models. With the MTB extension it is possible to overlay activation patterns of existing task set models by variations of the underlying clock of the trigger source. These variations are modeled by so called time bases. Furthermore, the MTB extension allows a probabilistic description of execution times and a fragmentation of task execution times. This permits a realistic model of practical embedded systems and a model of cooperative scheduling, which can be used for scheduling algorithms and a more realistic examination of real-time requirements.

A number of optimal and efficient scheduling algorithms for multicore processor systems have been introduced in the last decade (see the overview in Section 3.3). However, in order to permit a proof of optimality, these algorithms assume simplified task sets. These task sets often don't conform to temporal properties of the applications of existing systems. Therefore, the work on these algorithms is very beneficial for theoretical understanding, but the applicability for practical systems is limited due to the overestimation of response times. Consequently, an adjustment of scheduling algorithms is necessary, which allows to schedule applications with a set of temporal properties, considering time bases and probabilistic execution times.

This work contributes the group of Partly Proportionate Fair (Partly-Pfair) scheduling, which extends the Proportionate Fair scheduling approach [BCPV96] by task sets with extended properties, namely: explicit variable task section execution times, sporadic task activation with multiple time base extension, and explicit task deadlines. It is shown that Partly-Pfair with PD^2 is able to schedule periodic and sporadic task sets with the MTB extension highly efficiently up to a system utilization of the complete multicore processor systems.

For the case of local scheduling algorithms with task or job-fix priority assignment, where each instance of a task executes on the same core of a multicore processor and the priority is constant or changes only between instances of a task, worst-case assumptions for the determination of the schedulability[1] exist. However, for the case of global scheduling algorithms with dynamic priority assignment, where instances of a task are able to migrate and the priority may change at each schedule decision, the worst-case scenarios of the local scheduling algorithms will no longer (see Section 4.3). Therefore, new schedulability examination methods are required which efficiently determine worst-case response times but also allow a stochastic evaluation of response times. For schedula-

[1] The term „schedulability“ denotes whether all real-time requirements of all tasks in a task set are met (for a detailed explanation see Section 4.1.1).

bility examination of global scheduling algorithms with dynamic priority assignment no general approach has been presented up to now. Additionally, formal approaches mostly use static analysis mechanisms, which prohibits dynamic changing of temporal properties. However, robustness considerations require an observation of the dynamic changes of temporal properties like short term perturbations in execution times or inter-arrival times.

Due to this, a schedulability examination method is part of the contribution of this work, based on a discrete event simulation. This method allows to approximate worst-case bounds on schedulability for global scheduling algorithms with dynamic priority assignment. Since this kind of scheduling algorithms is a generalization of global and local scheduling approaches with task/job-fix priorities, the proposed method can be applied to these scheduling algorithms as well. Furthermore, the injection of perturbations allows a robustness consideration of the schedulability of embedded multicore systems.

A common approach for the comparison of scheduling algorithms is an algorithm specific utilization bound which guarantees schedulability. These bounds incorporate any possible task set, but many task sets are not used in practical systems and therefore a schedulability bound consideration is not representative for practical application.

This work presents a sensitivity analysis, which allows to consider only a subgroup of task sets, representing existing and upcoming systems which are modeled by a probabilistic task set. For the examination of the probabilistic task set, a Monte-Carlo randomization approach generates task set models which can be analyzed by a schedulability examination approach, e.g. the discrete event simulation. Statistical estimators of the subsequent sensitivity analysis allow a comparison of different scheduling algorithms, whereas a bootstrapping approach determines confidence intervals on the statistical estimators. This approach can be used to benchmark multicore scheduling algorithms according to different metrics, e.g. deadline compliance, in dependence of a certain task set characteristic like the system utilization.

1.3 Structure of the Work

The remainder of this dissertation is organized in the following parts. Part I provides necessary background information for this work. It contains a summary of common fundamentals of real-time theory and gives an overview of related work and latest results in modeling of real-time systems, multicore real-time scheduling, and schedulability examination methods. Part II describes the contribution of this work. After a formulation of the investigation purpose, the new multiple time base extension of task sets is presented. This extension is useful to model automotive powertrain systems and many other sys-

tems, e.g. embedded systems with task activations from bus messages. The extension allows a precise specification of temporal properties of clock based activation sources, variable task execution times and cooperative schedule points and therefore allows a less pessimistic analysis as it is possible with today's task sets. For these task sets, the global multicore real-time scheduling approach Partly Proportionate Fair and a work-conserving and non-work-conserving algorithm, implementing the Partly Proportionate Fair approach, is introduced. Afterwards, a simulation-based approach for multicore schedulability examination is shown which is able to evaluate global real-time scheduling algorithms with dynamic priority assignment and many other algorithms like global scheduling algorithms with tighter limitations on priority assignment or local scheduling algorithms. In the next chapter, this simulation approach is used for a sensitivity analysis of temporal properties for a probabilistic system model, which can be used for benchmarking local and global scheduling algorithms with task-fix, job-fix, or dynamic priority assignment. Part III validates the contribution in three case studies, based on typical systems of the automotive domain, whereas the data sets were kindly provided by the industrial partner of the DynaS^3 project. The first case study evaluates scheduling algorithms for a quadcore automotive system. The second case study evaluates the efficiency of scheduling algorithms for a singlecore automotive powertrain system which is ported to a dualcore system. The third case study evaluates the robustness of scheduling algorithms for this quadcore system by inserting temporal perturbations in task set parameters during simulation according to temporal, efficiency, and robustness properties. In the final part, a summary of this work and proposals for further work are provided.

Part I

Preparation and Related Work

Chapter 2

Real-Time System Model

This chapter concerns real-time system models, describing the temporal properties of a real-time system. This definition is fundamental for all further investigations on scheduling algorithms and real-time examination methods. After a definition of the context of this work, a model for embedded systems is introduced. Based on this model, a conclusion of related work for task set models is given. Finally task set characteristics, which are used for the evaluation of scheduling algorithms, are presented.

2.1 Fundamentals

An analytical investigation of a physical object requires an abstract model, reducing the complexity of the object, to be able to focus on the investigated aspect. This chapter describes the properties of an embedded system and introduces a formal abstract model for further analysis. The abstract model includes properties which are relevant and ignores properties which are non-relevant for the investigation.

In the investigation of this work the real object is an embedded system. By DIN VDI 3633 [VDI96], a system ...

> „... is a delimited collection of components that are in interconnection with each other.“

An *embedded system* is defined in the following way.

Definition 2.1
An ***embedded system*** *is a collection of hardware and software components, which has to fulfill its purpose according to logical and temporal requirements. It consists of input ports that request a functionality, a software application that describes the functionality, one or multiple processor(s) that execute the software application, an operating system that administrates the execution of the software application, and output ports that provide the result of the execution. To the result of the software application, temporal requirements are imposed.*

Since the embedded system interacts with the environment, the results at the output ports have to fulfill temporal requirements, to guarantee a successful interaction between the embedded system and the environment. The requirements result from the device, which is connected to the embedded system, e.g. the power electronic of an electrical engine or an actuator. From an external system view the temporal requirements refer to the results at the output ports. From an internal system view the temporal requirements are assigned to the execution of the workload on the processor. The requirements restrict the range or the variability of the start or the completion of the workload execution. Therefore, the software application is also called a *workload* with real-time requirements.

In order to guarantee all external temporal requirements, the operating system assigns the real-time workload to the processing resources. This part of the operating system, called scheduler, considers internal temporal requirements to fulfill external temporal requirements.

2.2 Real-Time System Model

The real-time system model merges or splits hardware and software components into abstract components for the investigation purpose. This work neglects properties of the embedded system which are nonrelevant for the investigation of the performance and temporal robustness of the embedded system. For accuracy, a formal description of the real-time system model is used, and for legibility, the formal description is extended with phrasings and graphical illustrations.

A real-time system S is a discrete event system because real-time system variables are discrete[1] and real-time system variables change only at time quanta[2]. A real-time system S transforms a vector of input interface signals $\vec{\Phi}^{I}_{\mathrm{IF}}$ into a vector of output interface signals $\vec{\Phi}^{O}_{\mathrm{IF}}$. The vectors represent the interface signals from the environment to the real-time system $\vec{\Phi}^{I}_{\mathrm{IF}}$ and from the real-time system to the environment $\vec{\Phi}^{O}_{\mathrm{IF}}$. The investigation of this work concerns the behavior between $\vec{\Phi}^{I}_{\mathrm{IF}}$ and $\vec{\Phi}^{O}_{\mathrm{IF}}$.

Both vectors $\vec{\Phi}^{I}_{\mathrm{IF}}$ and $\vec{\Phi}^{O}_{\mathrm{IF}}$ have n signals. The real-time system S includes for all n input interface signals a transformation of the ith signal[3] of $\vec{\Phi}^{I}_{\mathrm{IF}}$, called $\Phi^{I}_{\mathrm{IF}}(i)$, to the ith signal of $\vec{\Phi}^{O}_{\mathrm{IF}}$, called $\Phi^{O}_{\mathrm{IF}}(i)$.

[1] Discrete system variables originates from a digital value representation.
[2] Quantized time originates from digital microcontroller.
[3] A Signal $\Phi(i)$ can be a composition of multiple signals.

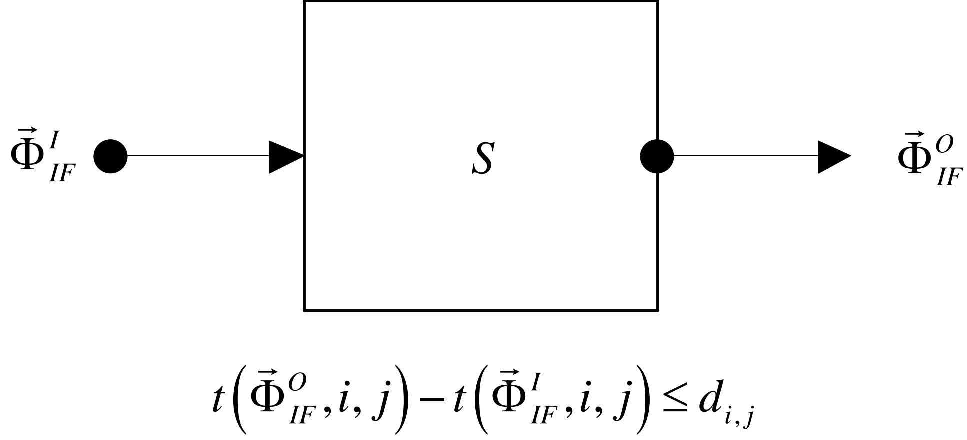

Figure 2.1: Graphical definition of a real-time system.

$$S : \vec{\Phi}^{I}_{\mathrm{IF}} = \begin{pmatrix} \Phi^{I}_{\mathrm{IF}}(1) \\ ... \\ \Phi^{I}_{\mathrm{IF}}(i) \\ ... \\ \Phi^{I}_{\mathrm{IF}}(n) \end{pmatrix} \longrightarrow \vec{\Phi}^{O}_{\mathrm{IF}} = \begin{pmatrix} \Phi^{O}_{\mathrm{IF}}(1) \\ ... \\ \Phi^{O}_{\mathrm{IF}}(i) \\ ... \\ \Phi^{O}_{\mathrm{IF}}(n) \end{pmatrix} \quad (i = 1, ..., n) \tag{2.1}$$

The occurrence of a signal $\Phi(i)$ is numbered with a counter j and the jth occurrence of $\Phi(i)$ is called $\Phi(i, j)$.

When function $t(\Phi, i, j)$ returns the *time* of the jth occurrence of signal $\Phi(i)$, then the transformation S has to fulfill Equation 2.2, whereas $d_{i,j}$ denotes the *deadline* of jth occurrence of the ith outgoing signal.

$$t(\Phi^{O}_{\mathrm{IF}}, i, j) - t(\Phi^{I}_{\mathrm{IF}}, i, j) \leq d_{i,j} \quad \forall i, j \tag{2.2}$$

Informally, each signal entering the real-time system input interface has to produce the corresponding output signal at the output interface at least after d time units. This is the fundamental property of a real-time system[4]. A real-time system S is defined in the following way:

Definition 2.2
A ***real-time system*** $S = (\tau, \Pi, \xi)$ *consists of a task set* τ*, a processing resource* Π*, and a scheduler* ξ*.*

The next sections give a formal definition of these components.

[4]Strictly speaking, this is the definition of a *hard* real-time system. Buttazzo [But05a] give a classification of weakening the relation between deadline and finalization time.

2.2.1 Task Set

For each transformation from input interface signals $\Phi^I_{\mathrm{IF}}(i)$ to output interface signals $\Phi^O_{\mathrm{IF}}(i)$, a task T_i exist. A task T_i is part of the real-time system S and transforms input signals Φ^I_i in output signals Φ^O_i (Fig. 2.2). Input *interface* signals $\Phi^I_{\mathrm{IF}}(i)$ differ from input signals $\Phi^I(i)$. $\Phi^I(i)$ can include signals which come from another task T_x of the real-time system $T_x \in \tau$, $x \neq i$, whereas $\Phi^I_{\mathrm{IF}}(i)$ includes only signals which come from the environment. Analogous, $\Phi^O(i)$ can include signals which are sent to another task T_x of the real-time system $T_x \in \tau,\ x \neq i$, whereas $\Phi^O_{\mathrm{IF}}(i)$ includes only signals which are delivered to the environment.

Definition 2.3
A ***task set*** $\tau = \{T_i\}$ *is the collection of all tasks* T_i *in a real-time system* S $(i = 1, ..., n)$.

Since a real-time system has to fulfill both, temporal and logical requirements, there are two considerations required: the temporal consideration and the logical consideration, shown in Figure 2.2.

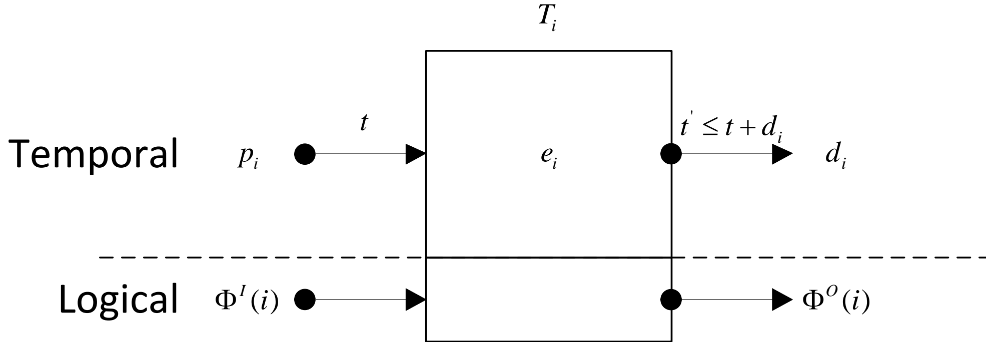

Figure 2.2: A task T_i has temporal properties, including inter-arrival time p_i, execution time e_i, and deadline d_i, and logical properties, including input signals $\Phi^I(i)$ and output signals $\Phi^O(i)$.

The temporal consideration of a task T_i describes the inter-arrival time p_i, the execution time e_i and the temporal requirement to the output d_i. The inter-arrival time p_i is a model for the occurrence of the input interface signal $\Phi^I_{\mathrm{IF}}(i)$ and defines when T_i is requested to be processed. For processing, a processing resource Π executes T_i, this requires execution time e_i. When $\Phi^I_{\mathrm{IF}}(i)$ occurs at request time t, the result of the execution of T_i at time $t^{'}$ is required until d_i time units at $\Phi^O_{\mathrm{IF}}(i)$.

The logical consideration of a task T_i describes which signals Φ^I_i a task consumes and which signals Φ^O_i a task produces, during the time interval between request and end of the execution of T_i.

Definition 2.4

*A **task** $T_i = (p_i, e_i, d_i, \Phi_i^I, \Phi_i^O)$ is defined by the temporal properties: inter-arrival time p_i, execution time e_i, and deadline d_i and by the logical properties: input signals Φ_i^I and output signals Φ_i^O.*

Table 2.1 gives a compact overview of all task symbols. Existing inter-arrival time models and execution time models will be discussed in chapter 2.3.

Finally the multiple task activation scenario in multicore systems is considered. Multiple task activation (MTA) concerns the scenario, when a further job of a task is activated, while the previous job has not finished. MTA can occur when a task has a deadline higher than the minimal inter-arrival time or when a job miss its deadline. In singlecore systems there is only one processor available and the processor is allocated to the first job. After finishing this job, waiting jobs of the same tasks are allowed to execute. At multicore systems, this restriction is a drawback according efficiency because there is a free core which is not used. Therefore, this restriction is revoked and it is assumed that jobs of the same task are allowed to execute concurrently. The code of a task has to be prepared for multiple entry in order to prevent data inconsistencies, e.g. by data duplication. Nevertheless, each time the earliest activated job has to be preferred to each other job of the task. However, it is possible that a later activated job finishes before a previously activated job.

Table 2.1: List of all task symbols.

Name	Symbol
task set	$\tau = \{T_1, ..., T_n\}$
task	$T_i \quad (i = 1, ..., n)$
task inter-arrival time model	p_i
task execution time model	e_i
task deadline model	d_i
input signals	Φ_i^I
output signals	Φ_i^O

Task Sections

As already mentioned, a task T_i gets input interface signals $\Phi_{\mathrm{IF}}^I(i)$ and sends output interface signals $\Phi_{\mathrm{IF}}^O(i)$. But a task also consumes and produces additional signals during execution. Therefore, a task T_i has a number of task sections T_i^k $(k = 1, ..., q)$, which allow to model the temporal behavior of inter-task signals (Figure 2.3).

Each task section T_i^k consumes signals $\Phi^I(i, k)$ and produces signals $\Phi^O(i, k)$. Signals $\Phi^O(i, k)$, which T_i^k produces, can be consumed by T_i^{k+x} of the same task T_i $(\forall x = \{1, ..., (q - k)\} \wedge k < q)$ or by other task sections T_x^k which are not element of the same task $(\forall x \neq i)$. It is also possible, that T_i^k produce signals which a task section T_i^{k-x} of

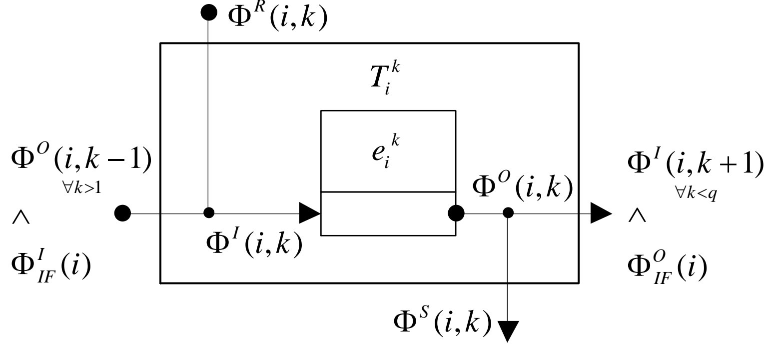

Figure 2.3: Graphical definition of a task section.

Table 2.2: Task section symbols.

Name	Symbol
task section	$T_i^k \quad k = 1, ..., q$
task section execution time model	e_i^k
task section input signals	$\Phi^I(i,k)$
task section output signals	$\Phi^O(i,k)$

the same task T_i ($\forall x = \{0, ..., (k-1)\}$) consumes, e.g. in the case of a feedback loop. However, the signal is then at least p_i time units old, because it can be consumed at the earliest at the next task activation.

Input signals $\Phi^I(i,k)$ which are not element of input or output data of any previous task section has to be loaded from external task sections. Formally, these requested signals $\Phi^R(i,k)$ are defined by Equation 2.3.

$$\Phi^R(i,k) := \Phi^I(i,k) \not\subset \left(\underset{\forall x=\{1,...,k-1\}}{\Phi^O(i,k-x)} \cup \Phi^I_{\text{IF}}(i) \right) \tag{2.3}$$

Similarly, each task section supports signals which are consumed by other external task section T_x^k or by a preceding task section. Formally, these supported signals $\Phi^S(i,k)$ are defined by Equation 2.4.

$$\Phi^S(i,k) := \Phi^O(i,k) \ \cap \ \big((\Phi^R(x,k) \ \forall x \neq i) \cup (\Phi^R(i,k-x) \ \forall x = \{1, ..., k-1\}) \big) \tag{2.4}$$

Shown in Table 2.2, each task section T_i^k requires an execution time e_i^k, has a vector of input signals ${\Phi_i^k}^I$, and has a vector of output signals ${\Phi_i^k}^O$.

The request time of a task section depends on task section position. For a task section T_i^k with $k = 1$, the time t of earliest possible execution of T_i^k is $\Phi^I_{\text{IF}}(i)$.

For all other task sections T_i^k with $1 < k \leq q$, the time $t(\Phi, i, k, j)$ of earliest possible execution of the jth execution of the kth task section of task T_i is $t(\Phi^O, i, k-1, j)$. This *sequentially dependency* is required because otherwise the output signals of the preceding task section would not be available.

Figure 2.4 shows the composition of task sections to a task.

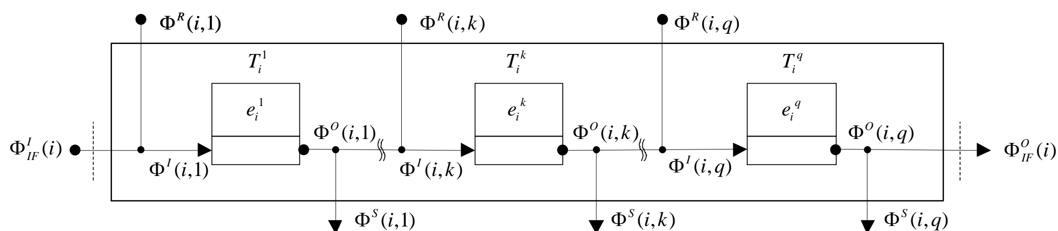

Figure 2.4: Example of a task with task sections.

Runnables

This section gives a brief overview of the runnable architecture. Industrial software architectures and operation systems in automotive systems, e.g. the AUTOSAR standard [AUT10] and the OSEKVDX operating system [Int05], divide the implemented functionality of the system into runnables. Dependent on data dependencies and temporal requirements, runnables are allocated to the task sections of a task.

Definition 2.5
*A **runnable** R is the highest subdivision of a real-time application and defines an execution time e.*

Runnables include the program code for a certain functionality. A runnable requires incoming signals and is able to modify these incoming signals to generate new signals. All outgoing signals are consumed by other runnables or they are sent to the environment. All incoming signals of runnables in a task section T_i^k are composed to the signal input vector $\Phi^I(i,k)$. All outgoing signals of runnables in a task section T_i^k are composed to the signal output vector $\Phi^O(i,k)$.

The premise of this approach is coincident with the interface-based design approach [DH01]: system designer, composing a set of runnables, only need to understand the runnables's interface and not the details of how the functionality offered by the runnable is implemented. This enables to integrate a set of runnables if their input and output signals „match".

Runnables have the temporal requirements sampling period or a delay, measured from any signal. The temporal requirements define a sampling period in the case of

time triggered polling mechanisms. These mechanisms are used e.g. when in a defined time-interval new state changes of the real-time system are detected, processed, and the information is distributed by signals to other runnables. The temporal requirements define a delay in the case of event triggered execution mechanisms. This is used e.g. when changes of environment values trigger an event for processing.

The system designer assembles the system real-time from its components, considering temporal requirements and dependencies. As a result, runnables are allocated to tasks (according the temporal requirements) and they are arranged to task sections and (mostly according the dependency requirements). This work assumes the allocation of runnables to task sections has already be done as a part of the software integration under consideration of temporal and dependency requirements.

2.2.2 Processor

This work defines that a real-time system contains exactly one multicore processor, with m processing elements P_x $(x = 1, ..., m)$. A real-time system with a multicore processor is denoted as *multicore system*.

Definition 2.6
A ***processor*** Φ *describes the number* m *and the processing speed* $\sigma(t)$ *of all available processing elements* P_x *in an embedded system. A processor* Φ *has a shared memory to allow processors* P_x *to share signals.*

This work constrains on symmetric multicore processor systems, where a task is physically able to be executed on each core. Furthermore, all processing elements have the same processing speeds $\sigma(t)$ at a certain time, whereas the processing speed can vary in time. This property is a sub group of *dynamic voltage/frequency scaling* (DVFS) theory, which studies mechanisms of energy saving through reduction of the processor voltage or frequency, to minimize power consumption (see [PS01] for an overview).

Alternatively, for comparing studies, a real-time system contains exactly one singlecore processor, which implies $m = 1$. This kind of real-time system is called a *singlecore system*.

In general, an embedded processor architecture includes beside the external memory a cache architecture which allows to reduce the access time on signal data.

For the inter-task communication mechanisms, a shared memory allows to communicate task signals. In the processor model Φ, a processor has a shared memory, where all processing elements P_x can access in the same way with the same access time. This is also denoted as a symmetric multiprocessing. Heterogeneous approaches, like Non Uniform Memory Access (NUMA), Cache Coherent Non Uniform Memory Access (CC-NUMA), Cache Only Memory Access (COMA), Pseudo Uniform Memory Access (P-UMA), or Private Memory Access (PMA), are not considered in this work. All these architectures

lead to a diverse memory access time from different cores. For these architectures a certain group of scheduling algorithms is required, which considers memory access time, therefore in the most cases local scheduling algorithms are applied. Otherwise, worst-case response time could increase enormously due to a high effect of the caching architecture on the worst-case execution time.

There are different memory access models for a shared memory possible. In general it is distinguished between blocking and non-blocking access. Blocking mechanisms delay all read or write accesses when there is already a write access to a signal, until the signal has been written. Non-blocking mechanisms duplicate a signal for write access. During write access to the original signal, other tasks are able to read the value from the duplicate of the signal. After finishing writing, all tasks are able to access to the original signal again.

The blocking access, also mentioned as exclusive access, has the benefit of data consistency for all processors. However, blocking mechanisms can lead to high task response times. A nonblocking access requires a special system architecture, which duplicates a part of the data and has hardware or software consistency mechanisms.

This work assumes such a non-blocking memory access approach and proposes a multiplexed-segmented shared memory access system. Multiplexed memory, i.e. a multiplexed memory port (MMP), dynamically wires the exclusive access of a memory region to a processor. Segmented memory enables a processor the exclusive access to a memory segment, without interfering other processors. This is beneficial because access, i.e. write access, to memory cells is always exclusive. Common memory sections produce a high interfering through access requests which results in a high memory access latency. At the segmented approach each instance of a task has it's own segment of memory. This has two advantages. Firstly, task instances doesn't access this memory segment because it contains exclusive task signal data. Secondly, in the case of a task instance migration, it is easy to switch the access to another processor by the multiplex memory port without delay.

Now the realization of the multiplexed-segmented shared memory access is discussed. First of all, the approach of task section related signal storage is introduced, afterwards a mechanism of signal distribution is proposed.

Figure 2.5 shows the memory architecture of the multicore system. The memory architecture divides in three parts:

- Shared memory (SM)
- Private memory (PM)
- Cache memory (CM)

The approach avoids blocking times by simultaneous write access on the shared memory. During the execution of a task section, produced signals are located in the local CM

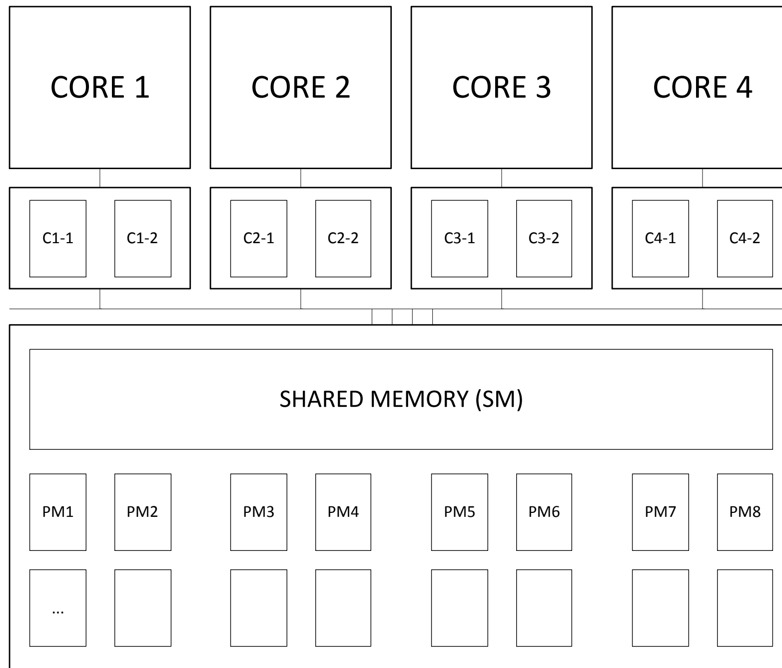

Figure 2.5: Memory architecture of a multicore processor.

(Figure 2.6, step 1). When a task section has finished, the processor Load and Store Unit (LSU) provides the new or changed signals to the PM, located in the main memory area. Triggered by an event or in a defined time interval, a synchronization task provides the signals to the SM (Figure 2.6, step 3). On the SM all other task sections are able to access and get the new value of the signal (Figure 2.6, step 2 and 4). Using an explicit synchronization has two reasons. Firstly, when there are multiple sender of the signal and in the PMs multiple values of the signal are ready to send, a direct access without blocking could lead to data misses and a blocking access effects other processors and leads to high memory access latencies. Secondly, collection pattern could be applied at synchronization, e.g. boolean operation, addition, or summation.

In the case of a multi-layered cache, it is assumed that lower level caches have the same content as CM. The PM's are realized as a port-multiplexed segmented memory, whereas one segment requires the capacity of the maximal signal vector size of all task sections. Additionally each CM has the same size, because it has to support the same data of the PM to the processing core.

Effects of additional devices e.g. communication buses, peripherals, or further blocking resources are not considered in this work. Tasks are assumed to be independent according blocking, meaning they have no blocking resource or precedence relationship[5].

[5]Due to signal duplication in the SM, a task is able to receive the actual instance of a signal during another task produce a new instance of the signal.

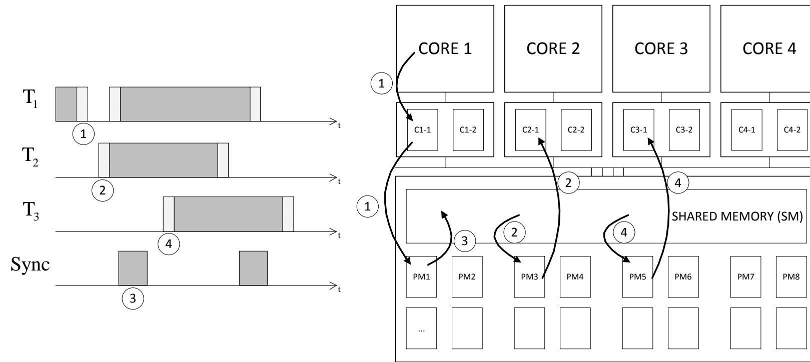

Figure 2.6: Signal storage and distribution in the task section based communication approach. A task stores signals during calculation in the local cache (C), which is synchronized with the private memory (PM) (Step 1). Collectively shared signals can be loaded from the shared memory (SM) (Step 2). A synchronization routine (sync) gathers all signals from the private memories and provides them to the shared memory (Step 3). Afterwards, the actual signal value can be loaded from the shared memory (Step 4).

2.2.3 Scheduler

Definition 2.7

*A **scheduler** model ξ describes the algorithm of a real-time system, which allocates the task set τ to the processor Φ.*

In multicore scheduling theory, it is distinguished between local, global, and clustered approaches.

Local scheduling (also mentioned as *partitioned scheduling*) means that for each processing element P_x an instance ζ_y of the scheduling algorithm ξ exist. Tasks are assigned before runtime to a scheduler instance ζ_y $(y = 1, ..., h)$. A scheduling approach is called *local* if $|\{\zeta_y\}| = |\{P_x\}|$. However, the assignment of multiple tasks on multiple processing elements was proved by Garey and Johnson to be NP-hard [GJ79].

Global scheduling means that there is only one scheduler instance ($|\{\zeta_y\}| = 1$) which manages all processing elements.

Clustered scheduling is a combination of both and means that one scheduler instance is able to manage multiple processing elements but the real-time system has at least two scheduler instances ($|\{\zeta_y\}| < |\{P_x\}| \wedge |\{\zeta_y\}| > 1$).

2.2.4 Time Events

This section presents all time events of a task execution, necessary for further examinations.

Figure 2.7 shows timestamps of the execution of a job $T_{i,j}$ of a task T_i. At job activation $T_{i,j}.A$, the scheduler requires time in order to assign the job for execution to a processing resource. Then, job execution starts at $T_{i,j}.S$. If there is no preemption of the

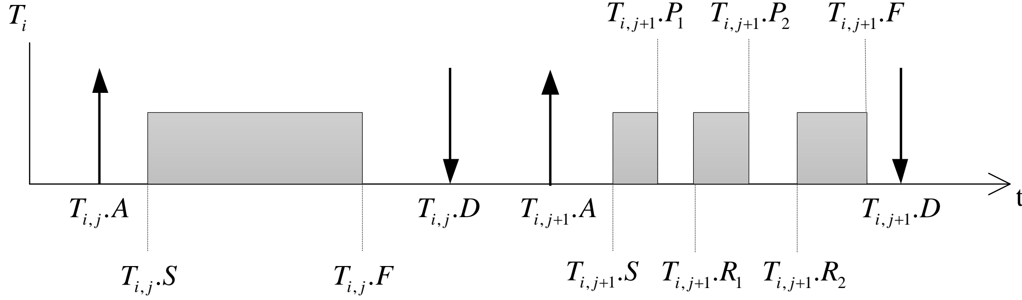

Figure 2.7: Notations of timestamps during the execution of job $T_{i,j}$ of task T_i.

Table 2.3: Job time intervals and timestamps.

Name	Symbol
job	$T_{i,j}$ $(j = 1, ..., g)$
job execution time	$T_{i,j}.e$
job deadline (relative)	$T_{i,j}.d$
job activation	$T_{i,j}.A$
job start	$T_{i,j}.S$
job finish	$T_{i,j}.F$
job deadline (absolute)	$T_{i,j}.D$
job suspension	$T_{i,j}.P$
job resume	$T_{i,j}.R$

job, the next time stamp is the job finish $T_{i,j}.F$. The temporal requirements are fulfilled, when the absolute job deadline $T_{i,j}.D$ is later as $T_{i,j}.F$. In the case of a preemptions, which enter in Figure 2.7 at the next task instance $T_{i,j+1}$, job suspensions $T_{i,j}.P_1$ and $T_{i,j}.P_2$ and job resumes $T_{i,j}.R_1$ and $T_{i,j}.R_2$ are additional timestamps.

Table 2.3 gives a compact overview of all job time intervals and timestamps. For differentiation, time intervals are expressed by lower case symbols and time stamps are expressed by upper case symbols.

2.3 Related Work

In order to examine whether an embedded system fulfills the real-time requirements (Equation 2.2), task set models for demand and execution time are necessary. These models are used at real-time system examination methods e.g. to determine the maximal response time of a task, which equates the time between the request and the finalization of a task. The response time is determined under consideration of all tasks of the task

set, the scheduling algorithm, and the processor. In past work, several models were introduced, which allow to describe and classify temporal properties.

This section gives an overview of existing task set models, even when they are mostly used to describe the workload for uniprocessors. Chapter 6 presents extensions which will be used in the following for multicore scheduling considerations. This considerations are divided in models for the demand of execution time, i.e. tasks activation models, and models for the execution time, i.e. execution time models.

2.3.1 Models for Demand of Execution Time

The demand model p_i of a task T_i describes the quantity of task activations as a function of time, time interval, or in dependence of other signals. The task demand models is classified in three groups:

- Recurrent demand model
- Arrival Curve demand model
- Hierarchical demand model

The recurrent demand model [Liu69] originates from *sampling* mechanisms (Figure 2.8), also know as *polling* approach [But05a]. At this kind of activation, a task *actively* queries signals $\Phi^I_{IF}(t)$, process them, and sends the output signals to a receiver. The time interval between two successive activations depends on the required update frequency. This kind of activation is also mentioned as *active* triggering, because a system timer or another configurable trigger source activates the task. This demand model allows to calculate the absolute task activation time of the jth instance of a task in independence of the previous task activations.

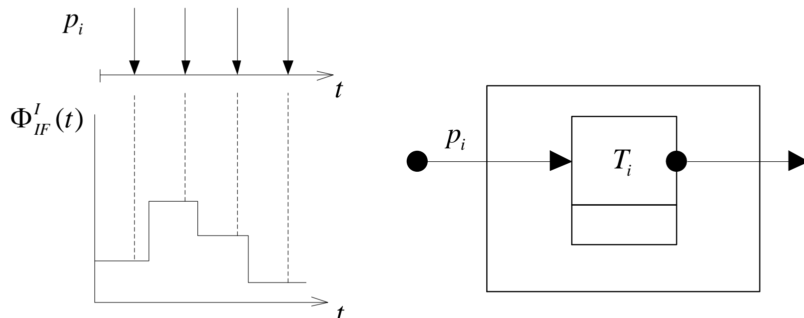

Figure 2.8: Example of a recurrent task activation. The input interface signal $\Phi^I_{IF}(t)$ is sampled and a task activation is triggered in a periodic manner with an inter-arrival time p_i.

The arrival curve demand model [LBT01] originates from *client-server* mechanisms (Figure 2.9). A change in an external signal, originating from the environment changes or from internal system changes, activates a task. This kind of activation is also mentioned as *reactive* activation, because a task waits for requests. This demand model allows to calculate only the relative task activation time of the jth instance of a task in dependence of the previous entered activations.

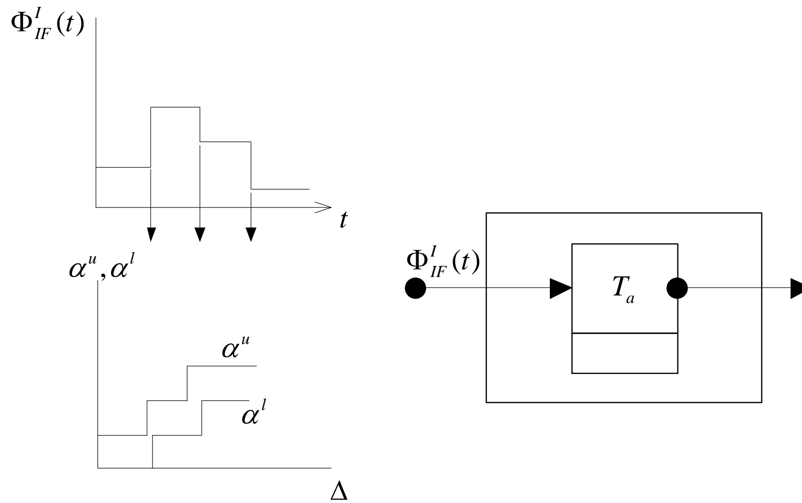

Figure 2.9: Example of an arrival curve task activation. A change in the interface signal $\Phi^I_{IF}(t)$ is immediately processed by an activated task. The upper and lower number of activations is expressed by an upper α^u and lower α^l arrival curve respectively as a function of any time interval with the size Δ.

Hierarchical demand models allow to model inter-task activation mechanisms (Figure 2.10). The recurrent demand model and the arrival curve demand model describe independent task demands. At both models, the reason of demand variation is not considered and is assumed to be in a „black box“. For the case of that a task T_a activates after finish another task T_b, these approaches would be to pessimistic because a delay of task T_a's output signals, which produce a delay in the activation of T_b, is not considered.

Recurrent Demand Model

The recurrent demand model describes task activations by an inter-arrival time. For each task instance $T_{i,j}$, the activation time $T_{i,j}.A$ can be determined by the instance counter

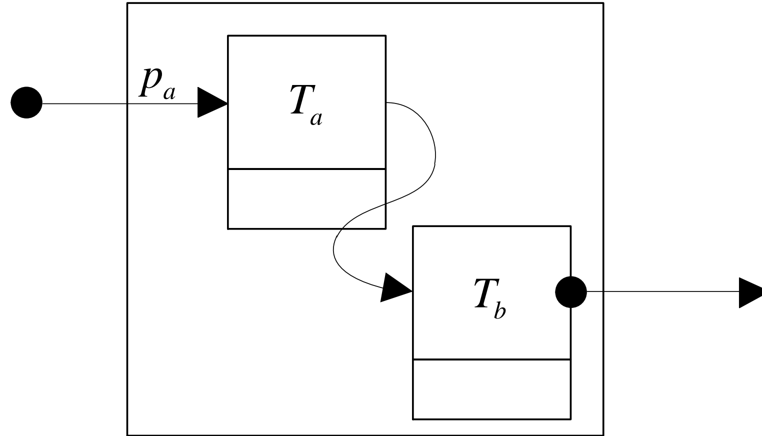

Figure 2.10: Example of a hierarchical task activation. A task T_a (activated in a periodic manner with inter-arrival time p_a) activates another task T_b at end of execution or at any other point during execution. The activation time of T_b depends on the delay of task T_a.

j. The job inter-arrival time $p_{i,j}$ represents the difference between the jth task activation time $T_{i,j}.A$ and the $j-1^{th}$ task activation time $T_{i,j-1}.A$ of task T_i.

$$p_i(j) = \begin{cases} T_{i,j}.A & j = 1 \\ T_{i,j}.A - T_{i,j-1}.A & \forall j > 1 \end{cases} \tag{2.5}$$

In 1973, Liu and Layland introduced the periodic task set [LL73]. It is the simplest request model because it assumes a constant inter-arrival p_i time between all successive activations $T_{i,j}.A$ and $T_{i,j-1}.A$. The periodic arrival time is defined by Equation 2.6.

$$p_{i,j} = p_i \ \forall j \tag{2.6}$$

A special case of periodic task set is called *harmonic task activation.* Harmonic task activation means that for each task or for a subgroup of tasks in a task set the inter-arrival time is an exact integer multiple of the next shorter period. A common approach, e.g. for the sampling based tasks of the task set of automotive systems, applies following design pattern.

1. The task with the lowest inter-arrival time represents the Greatest Common Divisor (GCD) for all tasks with a higher inter-arrival time.

2. When the tasks of a task set are sorted by inter-arrival time $\{T_1, ..., T_i, ..., T_n\}$, task T_i has an inter-arrival time p_i equal a multiple of the inter-arrival time of the task with the next lower inter-arrival time T_{i-1}.

The harmonic task activation increase the schedulability bound of many singlecore scheduling algorithms (see Section 3.2.1). Therefore, practical task sets are often designed to fulfill harmonic requirement.

For periodic activation, the offset extension [Tin92, PH98] considers dependencies between the release of tasks in a task set. In basic periodic task sets [LL73], there is a **synchronous task activation**, meaning the first activation of all tasks in a task set enters at time zero. In periodic task sets with offsets, there is a **asynchronous task activation**[6], meaning the first activation of each task in a task set has an offset o_i, referring the time between zero and the first activation. All successive activations have a constant inter-arrival time p_i (Equation 2.7).

$$p_{i,j} = \begin{cases} o_i & j = 1 \\ p_i & \forall j > 1 \end{cases} \tag{2.7}$$

The periodic task set with offsets is widely used in practical embedded systems, e.g. in sampling based systems[7]. For periodic task sets with offsets and a harmonic activation, offsets allow to deskew the task execution demand by shifting the activations over the inter-arrival time of the task with the next higher inter-arrival time.

Palencia and Harbour [PH98] considered task offsets for schedulability considerations, but this work differs from whose work according to the definition of the relative task deadline. In [PH98], offsets for periodic task activations represent a forced delay of task release time from an originally periodic activation pattern, mentioned as external event. The delay results from limitation in reaction time for activation or from initial effort for transferring local task data.

This work considers the offset as a conscious design decision, in order to apply load balancing mechanisms. This differentiation is fundamental at consideration of deadline requirements. In this approach the deadline is measured relatively from the shifted activation, including the task offset. Palencia and Harbour measure the deadline from occurrence of the external event, excluding the task offset. Therefore, the definition of Palencia and Harbour results in a lower available time for task execution in comparison with the definition of this work.

Sporadic Activation

Instead of the notation of sporadic task sets [Mok83], this work considers the representation of sporadic activation through arrival curves.

[6] Also mentioned as desynchronized task activation

[7] Sampling based systems use modified polling mechanisms to detect state changes of the system. With a constant task activation pattern, sensor values are requested, processed and propagated to environment or other tasks.

Arrival Curve Demand Model

The arrival curve demand model concerns an event based task demand behavior. A task activation is not triggered in a periodic manner, but with an event. An arrival curve describes the entrance of the event. Arrival curves are used in Real-Time Calculus theory, which will be introduced in Section 4.2. This section gives a brief overview on arrival curves.

Assuming a trace of task activation events, called event stream, with time stamps of event occurrence. Then this event stream can conveniently be described by means of a cumulative function $R(t)$, defined as the number of events seen on the event stream in the time interval $\Delta = [0, t)$. While $R(t)$ describes one concrete trace of an event stream, a tuple $\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)]$ of upper and lower arrival curves [Cru02] provides an abstract event stream model, representing all possible traces of an event stream. The upper arrival curve $\alpha^u(\Delta)$ gives the upper bound on the number of events in a time interval Δ. Similarly, the lower arrival curves $\alpha^l(\Delta)$ gives the lower bound on the number of events in a time interval Δ. $R(t)$, α^u, and α^l are related by the following equations:

$$\alpha^l(\Delta) = \min_{\lambda \geq 0} \{R(\Delta + \lambda) - R(\lambda)\}$$
$$\alpha^u(\Delta) = \max_{\lambda \geq 0} \{R(\Delta + \lambda) - R(\lambda)\}$$

The arrival curves are right-continuous, non-negative, subadditive functions [CKT03]. Informally, given any finite length event trace (from measurements or from simulation) and a real number Δ, it is possible to determine the values of $\alpha^l(\Delta)$ and $\alpha^u(\Delta)$ corresponding to the event trace by sliding a window of length Δ over the trace and recording the minimum and maximum number of events lying within the window respectively [CKT03].

The arrival curve is an abstract representation, which can be specified for the analysis of different system properties. In [CKT03], Chakraborty et al. used this representation for the execution time requests for tasks.

This is introduced with an example, based on periodic activation pattern[8]. For the construction of the upper and lower arrival curve the time interval $[0, T_i.p)$ is considered. In any time interval of the length $[0, T_i.p)$, the maximum number of activations of task T_i are 1 and the minimum is 0. In the time interval $[0, 2 \cdot T_i.p)$ the maximum number of activations is 2 and there is at least 1 activation. Proceeding in this way, the arrival curve can be constructed up to the interested time interval. Figure 2.11 shows the result for a task with $T_i.p = 7$.

[8]This simple task activation pattern is used for explanation reasons, it can easily adapted to any arrival curve

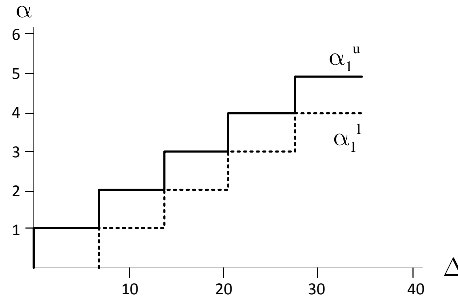

Figure 2.11: Arrival curve of periodic task activations with $T_i.p = 7$. The upper α_i^u and lower α_i^l number of task activations are specified in dependence of the size of any time interval Δ.

Hierarchical Demand Model

The hierarchical demand model [ABS06] derives from message processing real-time systems. In those systems, a message requires different processing steps and each step is located in a task. During a certain execution point in a task or when a task has finished, the task with the next processing step is activated. The following tasks are able to activate the next tasks in the same manner.

The hierarchical sequence of task activations is also mentioned as *task chain* [Int05]. In order to model the activation of this sequence, the first task of a task chain requires a recurrent demand model or an arrival curve demand model. For the activation of all further tasks in the task chain, the task section which activates the next task in the chain is defined. Whenever this task section finishes, the next task in the chain is activated.

2.3.2 Models for the Execution Time

The execution time model e_i describes how the execution time changes between the jobs of a task T_i. There are several effects, which cause execution time variations. Examples are different code branches, cache misses, instruction pipelines, out-of-order execution, etc. The following sections give an overview of commonly used models and its purpose.

WCET

Scheduling theory is mainly driven by approaches validating deadline compliance for hard real-time systems.

In sustainable systems[9] it is sufficient to show that the task set is schedulable, when all tasks have the maximum execution time, denoted as worst-case execution time (WCET).

[9] A system is called sustainable, when a schedulable task set remains schedulable at a decrease of execution time (see also Section 4.3).

The execution time for the worst-case execution time model $WCET$ is defined by Equation 2.8.

$$WCET := T_{i,j}.e = T_i.e^{WCET} \ \forall j. \tag{2.8}$$

The WCET analysis computes upper bounds for the execution time. This complex approach has to determine the longest execution time, effected by code branches, loop cycles, cache model, processor pipeline architecture and further influences[10]. Additionally the range of variables influences the executed code branch or limits the number of loop executions. However this ranges are difficult to bound because they often depend on the use case and the environment of the real-time system. Therefore, the WCET analysis approximates upper bounds for the WCET, i.e. the WCET analysis does not guarantee to determine the WCET exactly but determines a value which is at least as high as the exact WCET. Since assumptions are necessary, which in general overestimate execution times in order to guarantee the upper bound on execution time, the WCET model has a high degree of pessimism.

The WCET is mainly used in formal schedulability approaches, e.g. Response Time Analysis (see Section 4.2.1), in order to determine whether a task set is schedulable at the worst-case scenario.

For systems where the sustainability is not given, effects occur which are also known as timing anomalies (see Section 4.3 and [LS02, Gra71, RWTW06]). These effects provoke that the response time possibly increases when tasks have a lower execution time. In case of such systems, using common static approaches for worst-case examination are not sufficient. Therefore, an extension of the execution time model is presented in chapter 6.

BCET-WCET

The oppositional execution time model of the WCET model is the best case execution time (BCET) model. BCET analysis calculates the minimal execution time.

The execution time for the best case execution time model $BCET$ is defined by Equation 2.9.

$$BCET := T_{i,j}.e = T_i.e^{BCET} \ \forall j. \tag{2.9}$$

Figure 2.12 shows the bounds on WCET and BCET which are determined by different approaches. Profiling methods trace the execution time of tasks on a simulated or real hardware. They allow to detect the distribution of execution time, but in general doesn't achieve the WCET (maximal observed execution time) and the BCET (minimal observed execution time) exactly. Static methods determine bounds on the WCET (upper timing bound) and BCET (lower timing bound) and guarantee that the execution time doesn't

[10] See [WMM$^+$08] for a comprehensive overview.

exceed bounds, but they are in general pessimistic and doesn't describe how often a certain execution time enters.

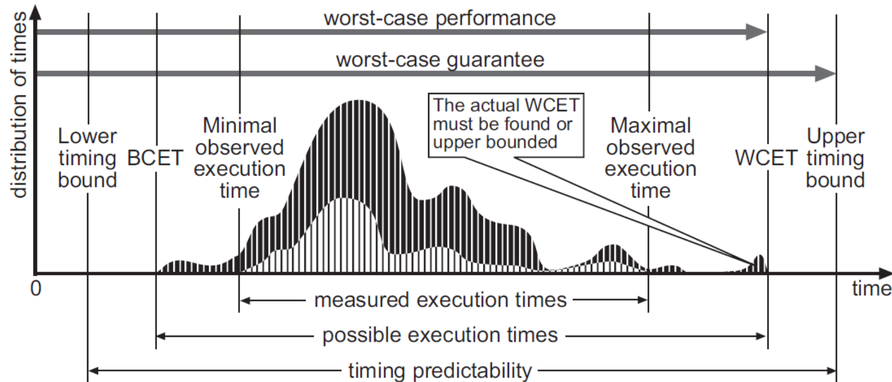

Figure 2.12: Comparison of WCET and BCET bounds, detected by different approaches [WMM+08]. The maximum of the measured execution time approximates the exact WCET from the left side by systematically executing measurement scenarios. Analytically determined maximum execution time approximates the exact WCET from the right side by reducing the pessimism of the prediction. The BCET determination works analogous.

The BCET model is widely used in combination with the WCET model. The BCET model in combination with formal methods, are used to determine the lower bounds on the best achievable response times. Comparing the BCET model and the WCET model allows to estimate the range of changing response times and can be used when not only deadline compliance is required but also a periodic requirements are defined for tasks (e.g. see Section 4.1.2 for jitter metrics).

2.3.3 Classification of Deadline Bounds

Finally this section gives a classification of deadline bounds. The classification limits application of both, scheduling algorithms and methods for examination of schedulability. The specification of bounds relatively to the inter-arrival time of tasks derives from methods and metrics for schedulability.

- **Implicit deadline**: The basic model, introduced by Liu and Layland [Liu69] assumes deadlines are implicit given by the minimal task inter-arrival time.
- **Explicit deadline**: The extension of Mok [Mok83] explicitly defines deadlines but restricts deadlines to be smaller than the minimal inter-arrival time.
- **Arbitrary deadline**: The final extension of Lehoczky [Leh90] allows arbitrary deadlines. When task deadlines are greater than the minimal inter-arrival time, it is possible that task instances of the same task overlap in execution without violating deadline constraints. Therefore, multiple activations of a task could enter

between the inter-arrival time span, which is not considered at the most schedulability examination methods.

2.4 Characteristics of Task Set Models

For a task set τ, there are several characteristics, used for further schedulability analysis or real-time system evaluation.

For task sets with implicit deadlines and periodic or sporadic activation, the *system utilization* $U_{sum}(\tau)$ equates both, the resource consumption and the percentage time, reserved to execute tasks to fulfill all temporal requirements.

$$U_{sum}(\tau) \stackrel{\text{def}}{=} \sum_{T_i \in \tau} \frac{T_i.e}{T_i.p} \tag{2.10}$$

$$U_{max}(\tau) \stackrel{\text{def}}{=} \max_{T_i \in \tau} \frac{T_i.e}{T_i.p} \tag{2.11}$$

$T_i.e$ equates the Worst-Case-Execution Time and $T_i.p$ equates the minimal inter-arrival time of a task T_i. The *maximal task utilization* $U_{max}(\tau)$, i.e. *maximal task weight*, is the maximal utilized task of all tasks in a task set. The *task weight* $\mathrm{wt}(T_i)$ of a certain task T_i is defined in the following way (in some work, *task utilization* is used instead of task weight).

$$\mathrm{wt}(T_i) \stackrel{\text{def}}{=} \frac{T_i.e}{T_i.p} \tag{2.12}$$

For task sets with explicit deadlines, the *constrained system density* $\delta_{sum}(\tau)$ and the maximal constrained density $\delta_{max}(\tau)$ define the execution time relatively to the temporal requirement.

$$\delta_{sum}(\tau) \stackrel{\text{def}}{=} \sum_{T_i \in \tau} \frac{T_i.e}{T_i.d} \tag{2.13}$$

$$\delta_{max}(\tau) \stackrel{\text{def}}{=} \max_{T_i \in \tau} \frac{T_i.e}{T_i.d} \tag{2.14}$$

For task sets with arbitrary deadlines, the *generalized system density* $\lambda_{sum}(\tau)$ or the maximal generalized density $\lambda_{max}(\tau)$ equate the utilization or the constrained density, depending on the relative deadline.

$$\lambda_{sum}(\tau) \stackrel{\text{def}}{=} \sum_{T_i \in \tau} \frac{T_i.e}{\min(T_i.p, T_i.d)} \tag{2.15}$$

$$\lambda_{max}(\tau) \stackrel{\text{def}}{=} \max_{T_i \in \tau} \frac{T_i.e}{\min(T_i.p, T_i.d)} \tag{2.16}$$

A further important characteristic is the demand bound function $DBF(T_i, \Delta)$ [BMR90] ($\lfloor \chi \rfloor$ is the highest integer, smaller or equal χ; $\lceil \chi \rceil$ is the smallest integer, higher or equal to χ).

$$DBF(T_i, \Delta) = \max\left(0, (\left\lfloor \frac{\Delta - T_i.d}{T_i.p} \right\rfloor + 1)T_i.e\right) \tag{2.17}$$

For any time interval Δ, the $DBF(\tau, \Delta)$ of a sporadic task T_i bounds the maximal cumulative execution requirement by jobs of T_i that both arrive in and have deadlines within any time interval Δ. The $DBF(T_i, \Delta)$ function is used for schedulability evaluation of multiprocessor scheduling algorithms in Section 3.3.

Chapter 3

Real-Time Scheduling

This chapter summarizes results of the scheduling problem for singlecore and multicore systems. First of all, fundamental definitions are given and a theoretical classification of multicore scheduling algorithms is shown. This classification is extended by additional properties, important for practical considerations and the scheduling algorithms which are part of the contribution of this work. Afterwards the related work on singlecore and multicore scheduling algorithms is summarized.

3.1 Fundamentals

This section summarizes common definitions in scheduling theory. Afterwards a classification of scheduling algorithms is reviewed and extended by practical criteria. Based on this classification, a generic scheduler model is presented.

3.1.1 Definitions

As introduced in the previous section, during system execution, a task generates a job at activation, depending on the demand model.

The pool of waiting and executing jobs is called a *job sequence*. It contains a list of all jobs, which are activated and need to be assigned to a core. A *real-time scheduling algorithm* is an approach, which constructs a *schedule* to assign a job sequence to a number of processing resources in a manner that all job deadlines are fulfilled. A *schedule* is a list of jobs, sorted by the scheduling algorithm by application of *policies* to each job of the job sequence. The schedule defines the order how to assign a job sequence to a core. The position in a schedule is called *rank* and the highest ranked job has the highest rating to be allocated to the core. At job execution, several steps are required like context switching and cache consistency mechanisms. A *dispatcher* is responsible for these steps and manages the execution of a schedule.

Table 3.1: Classification of general utilization based schedulability bounds on multicore scheduling [CFH+04]. The maximal system utilization U depends on the number of cores m and for some groups of scheduling algorithms on the maximal task utilization $\alpha = U_{max}(\tau)$.

	No migration	Bounded Migration	Full Migration
Task-fix Priority	$(\sqrt{2}-1)m \leq U \leq \frac{m+1}{1+2^{\frac{1}{m+1}}}$	$U \leq \frac{m+1}{2}$	$U = \frac{m+1}{2}$
Job-fix Priority	$U = \frac{m+1}{2}$	$m - \alpha(m-1) \leq U \leq \frac{m+1}{2}$	$m - \alpha(m-1) \leq U \leq \frac{m+1}{2}$
Dynamic Priority	$\frac{m^2}{3m-2} \leq U \leq \frac{m+1}{2}$	$\frac{m^2}{2m-1} \leq U \leq \frac{m+1}{2}$	$U = m$

3.1.2 Classification of Multicore Scheduling Algorithms

In [CFH+04], Carpenter et al. give a categorization of multicore scheduling algorithms. They used a migration- and priority-based classification and determined bounds on maximal system utilization. This model is helpful for the case of comparing classes of algorithms, independent from task set properties. They assumed a preemptive, periodic task set and derived the schedulability bounds from Table 3.1. A schedulability bound defines the schedulability of a task set in dependence of a characteristic of the task set (see also Section 2.4). E.g. one characteristic of a task set is the system utilization U_{sum}. As long as the value of the task set characteristic is lower than the schedulability bound, all deadlines are met.

A usage of the complete system utilization, meaning a system utilization U_{sum} which is equal to the number of cores m, can only be achieved by full migration scheduling with dynamic priorities. At bounded migration scheduling, maximal system utilization also depends on the maximal task utilization α in task set.

For global scheduling of more general implicit-deadline sporadic task sets, the following theorem has been proven:

Theorem 3.1 ([Bak07])
Any implicit deadline sporadic task set τ, satisfying $U_{sum}(\tau) \leq m$ and $U_{max}(\tau) \leq 1$ is schedulable upon a platform comprised of m unit-capacity cores by a [global] scheduling algorithm with dynamic priority assignment.

To see why this holds, observe that a *processor-sharing* schedule [Bak07], in which each job of each task T_i is assigned a fraction $\mathrm{wt}(T_i)$ of processor time between its release time and its deadline, would meet all deadlines. Such a processor-sharing schedule may subsequently be converted to one in which each job executes on zero or one processor tick at each time instant by means of the technique of Coffman and Denning [CD73]. However such an algorithm is clearly very inefficient and not implementable in practical

Table 3.2: Classification of multicore scheduling algorithms.

	I	II	III	IV
Allocation	Local	Clustered	Global	
Disruption	Non-preemptive	Cooperative	Preemptive	
Migration	No	Job-	Section-	Full-
Prioritization	Task-fix	Job-fix	Section-fix	Dynamic
Processing	Work-Conserving	Non-Work-Conserving		

embedded systems.

This work gives a more practically oriented categorization for multicore scheduling [DSM+10a], which is used in the following for the design of a generic scheduler model. This classification of multicore scheduling algorithms can be found in Table 3.2.

Allocation

Allocation concerns the number of cores, where the job of an activated task is able to execute. This work differs between global, local (also called partitioned or static scheduling) and clustered scheduling.

Global scheduling means that a job of a task is able to execute on each core of the m multicore processor. Therefore, only one global queue of ready tasks exist and the scheduling algorithm generates a schedule which contains a ranking of at least m ready jobs. Dependent on further properties, e.g. migration or disruption, the dispatcher assigns the m ready jobs to the cores.

Local scheduling means that tasks are assigned to a core by a partitioning approach before runtime and each core has an instance of the same scheduling algorithm. All jobs of a task have to execute on the same core.

Clustered scheduling means that the m cores of a multicore processor are grouped in a number of core clusters. Each cluster has a scheduling algorithm, whereby it is possible that different kind of scheduling algorithms are used for the different clusters. Then, the following classification according to the other properties has to be done individually for each scheduling algorithm.

Disruption

Disruption defines when a job is able to preempt another job, caused from a changed job ranking in the schedule. It is differed between preemption and interruption, due to the difference in costs. *Preemption* occurs by another job and is very costly due to saving the complete task context. *Interruption* occurs by an ISR which is less costly because only a subpart of the context has to be stored due to limited functionality of ISR's. This work differs between non-preemptive, cooperative, and preemptive scheduling. Non-

preemptive scheduling means that a task never gets preempted by another task, also when this task has a higher rank. Since the higher ranked task has to wait for the complete execution time of the executing task, this could cause a high response time for the waiting task. *Cooperative* scheduling means that a task gets preempted only between two successive task sections. *Preemptive* scheduling means that a task can be preempted by other task at any point of execution.

In general, it is assumed that all tasks have the same class of disruption. In practical systems, there are also task sets, where a sub group of tasks of the task set have a cooperative disruption pattern to all other task of this group, but the tasks of this group are preemptive according tasks of a disjunct sub group of tasks of the task set.

Migration

Migration restricts the selection of cores for execution of a job. Migration can be used only for global or clustered allocation pattern. This work differs between no migration, bounded-migration, task section-migration and full migration. *No migration* means that each job of a task has to execute on the same core. *Bounded-migration* means that different jobs of a task can execute on different cores, but a job once started on a core has to resume after a preemption on the same core. *Task section-migration* means that a task can migrate only at task section end. *Full migration* means that a job can change the core after each preemption.

Prioritization

Prioritization defines the variability of priorities which are assigned by a scheduling policy of a scheduling algorithm to jobs of a tasks. This work coincides with the general classes of prioritization: task-fix, job-fix, dynamic priority and extends the classification with the class section-fix prioritization. *Task-fix priority* means that each job of a task has the same priority. This prioritization is also mentioned as *static priority* scheduling. *Job-fix* prioritization means that the priority can differ between different jobs of a task, but it is constant for one job. *Section-fix* prioritization means that the priority of a job can differ only between task sections, but is constant during one task section. *Dynamic* prioritization means that the priority of one job can differ at any schedule decision.

The benefit of section-fix prioritization is a lower sorting effort in comparison with dynamic prioritization and a higher degree of flexibility in resource allocation than job-fix prioritization.

Processing

Processing defines how a scheduler proceeds with the allocation of ready jobs. The two groups are work-conserving and non-work-conserving algorithms. A *work-conserving*

algorithm allocates ready jobs to a core as soon as there is a free core. A *non-work-conserving* algorithm is able to leave a core idle also when there are ready jobs.

3.1.3 Scheduling Model

This section gives a description of a generic scheduler model which can be applied to all scheduling algorithms of Table 3.2. The simulation, introduced in Section 8 includes this model for schedulability examination. In future work it can be used for a generic scheduler module implementation in embedded systems.

The generic multicore scheduler model has following phases:

- Phase I: Resource Analysis
- Phase II Task Nomination
- Phase III: Task Nominee Sorting
- Phase IV: Execution
- Phase V: Non-work-conserving preparation

A scheduler has a list of available cores with entries $\{1, ..., m\}$. Additionally the scheduler has a list of running jobs and a list of ready jobs. A task instance provides its state and all policy parameters, required for scheduling.

Resource Analysis

In the first phase, the scheduler determines for each core whether the core is blocked or unblocked. This depends on the blocking capability of the running jobs on a core. A job blocks a core if and only if the scheduling algorithm is non-preemptive or cooperative and the executing job has not reached task section end. Whenever there is a job which is able to preempt the running job, the core is unblocked.

Task Nomination

In this phase, the scheduler determines which jobs of the job sequence are able to execute. This is required because a job can be prohibited to execute, e.g. in the case of a non-work-conserving algorithm or in the case of non fulfilled precedence constraints. Afterwards the scheduler determines whether the job is schedulable on each of the unblocked cores. Limitation of migration at scheduling algorithms with bounded migration or task-section migration bounds additionally the selection of unblocked cores and migration overhead [SPDM09]. For example when an algorithm works in a job-fix migration manner and a job has already started execution on one core it can not execute on another core (e.g. at Fixed-Migrating EDF [ABD08]).

Task Nominee Sorting

In this phase, the scheduler sorts all jobs of the nomination list by a scheduling policy. The scheduler can have a single policy or a multi level policy. A multi level policy defines tie-breaking rules, applied when a policy is not distinct.

Execution

In this phase, the scheduler preempts running jobs and resumes waiting jobs according to the sorted nominee list. When a scheduler has a list of x cores and a list of f unblocked cores, then all jobs on the blocked cores, namely all $x - f$ jobs, remain in the state running. For all other jobs, the following procedure applies: Running jobs with a rank $> f$ getting preempted, running jobs with a rank $\leq f$ stay in the state running. All other jobs with a rank $\leq f$ resume the execution on the core where they were preempted the last time as long as the core is free. Otherwise they resume execution on any other free core. This preferred selection of a core allows to reduce the number of migrations.

Non-work-conserving preparation

For the case of non-work-conserving algorithms, the job sequence could include waiting jobs even if there is a free core. In general the scheduling algorithm is called at different points of execution of a job, e.g. at job termination, activation or at an explicit scheduler call. However non-work-conserving scheduling algorithms need to be called additionally when a waiting job becomes ready for execution. Waiting for the next scheduler call, e.g. by a finished job, could lead to a miss of ready time for a waiting job. Therefore, the first time when a waiting job gets ready for execution is determined. At this time a scheduler event is set where the scheduling algorithm executes.

3.2 Related Work on Singlecore Scheduling

This section discusses results of the scheduling problem of singlecore processor real-time systems. Singlecore scheduling algorithms are used for local scheduling of multicore processors.

3.2.1 Task-Fix Priority Scheduling

Today practical real-time systems mostly use task-fix priority scheduling [LL73]. Task-fix priority scheduling has practical and theoretical benefits, in comparison with job-fix or dynamic scheduling. Practical benefits are a low runtime complexity, because priorities don't have to be calculated during runtime and the runtime for sorting ready task queue is lower because of a presorted list of the previous scheduler execution. Further benefits are the possibility to apply formal schedulability analysis techniques due to a lower number

of cases that have to be considered, when determining worst-case response times (see Section 4.2 for more details).

In the last decades, many priority assignment policies for task-fix priority scheduling were introduced and discussed. The following part summarizes results on the scheduling algorithms Rate Monotonic and Deadline Monotonic.

Rate Monotonic (RM) [LL73] is a simple priority assignment policy for periodic task sets with implicit deadlines. It assigns tasks a priority with respect to the inverse of the task period, where the task with the highest value has the highest priority. RM is preemptive, therefore a running task is immediately preempted by an arriving task with a lower period. Liu and Layland [LL73] proved that RM is optimal among all fixed-priority assignments, in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM. For task sets τ, containing n tasks, the maximal system utilization is

$$U_{sum}(\tau) = n(2^{\frac{1}{n}} - 1). \tag{3.1}$$

For $\lim_{n \to \infty}$ the maximal system utilization is $\ln 2 \approx 0.69$. However, the schedulability bound improves when task periods have a harmonic base, defined in the following way.

Definition 3.1 ([BB03])
Let $P = \{p_1, p_2, ..., p_n\}$ a set of periods of the task set $\tau = \{T_1, T_2, ..., T_n\}$ of periodic tasks. A subset $R \subseteq P$ is said to be a harmonic base of τ if there is a partition Z of P into $|R|$ subsets such that:

1. *Each member of P is a multiple of the smallest element in exactly one member of the partition Z;*

2. *If x and y are two elements in the same member of the partition Z, then either x divides y or y divides x.*

Each subset in the partition Z is a harmonic chain.

However, a common misconception is to believe that the schedulability bound becomes $U_{sum} = 1$ when the periods are a multiple of the smallest period. Kuo and Mok [KM91] proved for harmonic chains the achievable task set utilization $U_{sum}(\tau) = K * (2^{\frac{1}{K}} - 1)$, whereas K is the number of harmonic chains with $K \leq n$. Due to the increase of maximal system utilization, harmonic chains are also often used in practical systems.

For more general periodic task sets with explicit deadlines, Leung and Whitehead [LW82] proposed the algorithm *Deadline Monotonic* (DM). At DM, each task has priority according to the inverse of its relative deadline. Since in general deadlines are constant, DM is also a task-fix priority algorithm. However, for task sets with explicit

deadlines, the system utilization bound of RM is no longer representative for schedulability considerations. A general, but pessimistic bound on schedulability of DM can be derived on the Rate-Monotonic schedulability test, modified by the constrained system density [But05a]. Therefore, a task set of periodic tasks with constrained deadlines is schedulable, as long $\delta_{sum} \leq n\left(2^{\frac{1}{n}} - 1\right)$. For a tighter bound on schedulability, the exact interleaving of higher-priority task must be evaluated for each task. Audsley et al. [ABRW91, ABR+93] proposed a method which evaluates necessary and sufficient conditions for DM scheduling (for more details see Section 4.2.1).

3.2.2 Job-Fix Priority Scheduling

For more general task sets, e.g. task sets with sporadic tasks, also more general task assignment policies have to be applied, to achieve a high system utilization. One of the best-know and discussed job-fix priority scheduling algorithm is Earliest Deadline First (EDF). EDF assigns tasks a priority according to there absolute deadline, where the task with the earliest absolute deadline has the highest priority. Dertouzos [Der74] proved that EDF is optimal according feasibility for sporadic task sets, which means that whenever a task set is feasible, it is also schedulable with EDF [But05a]. This proof also holds for task sets with periodic task sets, because periodic task sets are a sub group of a sporadic task sets.

For task sets with explicit deadlines Baruah et al. [BRH90] proposed a schedulability test, called *processor demand* criterion. Informally, the processor demand criterion evaluates the amount of processing time, requested in a time interval by task instances with activation and deadline in the time interval. The feasibility (and for EDF also the schedulability) is guaranteed, if and only if the requested processing time does not exceed the available processing time for *any* time interval.

When the execution time of the scheduling algorithms is considered, EDF scheduling has a drawback in comparison with task-fix priority scheduling, because at task activation, the absolute deadline has to be calculated for each job. But, from a practical view, the higher execution time compensates, when context switching costs are taken into account. The number of preemptions that typically occur at RM is much higher than at EDF. In experimental studies, Buttazzo [But05b] showed that the average number of preemptions increases almost linearly at RM, while it decreases for high loads under EDF. A further advantage of EDF is that it is robust against permanent overload [CEBA02]. At overload, EDF automatically performs a periodic rescaling, which means that tasks are behaving like they are requested at a lower rate. This is beneficial in comparison with disregarding job instances of lower priority task at RM scheduling.

3.2.3 Dynamic Priority Scheduling

Since EDF is optimal regarding feasibility, for the purpose of a higher system utilization, there is no need to use a more general priority assignment method. Additionally, dynamic priority scheduling algorithms have the drawback of additional execution time for the determination of priorities by scheduling policies at each schedule point. In comparison with previously discussed algorithms, there is only a low number of publications on dynamic priority scheduling algorithms, mainly examining benefits of dynamic task priority scheduling for special task set models.

Mok [Mok83] proposed a dynamic scheduling algorithm, called Least Laxity First (LLF). The *laxity* defines the time a ready job can be maximally delayed for execution and hold its deadline, when the job executes without preemption. LLF prefers the job with the lowest laxity. The laxity of a job has to be calculated at each schedule point, because the laxity of a non-executed job decrease proportionally with the past time. LLF has been discussed in several works as algorithm for task sets, containing soft and hard task deadlines [DTB93, SB96]. LLF guarantees better response times for tasks with soft deadlines, while keeping all hard deadlines, in comparison with mechanisms which set all soft deadline tasks to background priority level.

3.2.4 Cooperative Scheduling

Many practical systems use *cooperative scheduling*, also mentioned as *deferred preemption* scheduling [BW09, BLV09] or non-preemptive scheduling [LA10]. At cooperative scheduling, a task can be non-preemptive for a limited time of the task execution time. This is a trade-off between full-preemptive and non-preemptive scheduling and has a number of benefits.

The schedulability bound of task sets increases, when deadline misses of a task set mainly enter at lower priority tasks [BW09]. Additionally, in systems using cache memory, arbitrary preemptions induce additional cache flushes and reloads [BLV09]. As a consequence, system performance and predictability degrade, complicating system design, analysis and testing [LLL+98]. Whenever a preemption takes place, different sources of overhead must be taken into account: the current task is suspended and inserted in the ready queue, the dispatcher stores executing task context and loads the next task context. The time of these operations is referred to as context switch cost. When the preempted task resumes, there are other indirect costs to be considered, related to cache misses, pipeline refills, bus contentions etc. Additional overhead occurs at refilling the pipeline of the pre-fetch mechanism, since preemption typically destroys program locality of memory references, and at reloading cache lines, evicted by the preempting task. These costs have a high variance and depend on the specific point in the task when preemption takes place, resulting in a low predictability. Non-preemptive sections have no increase in task execution times and therefore make execution times more predictable

[YBB10], since the task section executes until completion once it is started. A drawback of cooperative scheduling is an increased blocking time for higher priority tasks.

There are two types of cooperative scheduling:

- *Floating Non-Preemptive Regions* (FNPR) [YBB09]: Each task T_i can disable preemption for a time interval of at most Q_i units of time. When a higher priority task arrives, the running task can switch to non-preemption mode for Q_i units of time, before the preemption is triggered. Since the running task can switch to non-preemptive mode at any time, because the arrival of high priority tasks can be at any time, the non-preemptive regions are assumed to be floating inside the task code.

- *Fixed Preemption Points* (FPP) [Bur95]: preemption is limited to pre-defined positions, called preemption points. Thus, each task is divided into a set of fixed non-preemptive sections, the longest of which play a crucial role in the schedulability analysis.

The FPP approach is comparable with the introduced task section model in Section 2.2.1, because the task section execution time describes the length between two successive preemption points. However, the FPP approach assumes constant preemption points and the task section model allows a stochastic specification of preemption points.

For the dimension of the size of non-preemptable sections, Lee [LLL+98] introduced an approach which derives schedule points from the number of useful cache blocks in a task. At a certain execution point in a task, a useful cache block contains a memory block that may be referenced before being replaced by another memory block of the same task. The number of useful cache blocks at a given execution point in a task can be calculated by using data flow analysis technique [LHS+98]. Bertogna et al. [BBM+10] introduced an algorithm to set preemption points at task-fix priority scheduling algorithms or EDF scheduling in order to reach a trade-off between small blocking times and reduced preemption overhead. For the FNPR floating model, Baruah [Bar05] showed how to compute the longest non-preemptive interval for each task that does not jeopardize the schedulability of the task set under EDF, with respect to the fully preemptive case. Yao [YBB09] addressed the same problem with task-fix priorities. Other methods incorporate the effect of instruction cache on response time, e.g. for a single task [LHS+98, GMM99] or for interference of other tasks [RM06, SSE08]. A practical approach, used at OSEK scheduling [Int05], sets preemptive points between runnables or group of runnables, selected by the system developer.

3.3 Related Work on Multicore Scheduling

In 1973, Liu and Layland addressed in their seminal paper [LL73] the problem of assigning task to multiple processors during runtime. More than two decades later, Baruah et al.

proposed the first optimal solutions for this problem, called Proportionate Fair (Pfair) scheduling [BCPV96]. This section gives a brief overview of fundamental and latest results on local and global multicore scheduling algorithms.

3.3.1 Local Scheduling

At local scheduling, tasks are allocated to cores before runtime and a singlecore scheduling algorithm at each core schedules the allocated tasks. According feasibility, optimal singlecore algorithms like EDF and LLF are no longer optimal for the case of local scheduling on a multicore processor [DM89]. However local scheduling is a convenient bridge when changing from singlecore to multicore systems. Concurrency support is not required for local memory sections, and operating system functions for task migration are not required. Furthermore, existing scheduling algorithms and schedulability test for singlecore algorithms can be used. Optimal assignment of task to cores is a kind of bin-packing problem, which is NP-hard [GJ79]. However, an optimal[1] bin-packing solution is not required, as long a partition can be found that is schedulable on each core.

The most studied partitioning approach is the group of algorithms for the classical bin-packing problem [CJGJ78]. One dimensional bin-packing algorithms assign a number of boxes to a number of bins. Each box has a size which is a fraction of the size of a bin, equal to 1. The objective of bin-packing is to find a partition of boxes to bins, where no bin exceeds its capacity. Bin-packing algorithms distinguish between two criteria: the order of selecting a box and the order of selecting a bin.

Bin-packing can be used for partitioning of tasks in the following way: the task weight $\mathrm{wt}(T_i)$ indicates the size of a box and the available capacity of a core $C(P_x)$ indicates the size of a bin. The assignment of a task set τ on a multicore processor M expires in the following way.

In the first step, tasks are sorted by task weight $\mathrm{wt}(T_i)$ (Equation 2.12). The index i is used for the position of a task T_i in an ordered list $\{T_1, ..., T_i, ..., T_n\}$. It is distinguished between the following *sorting policies*:

- Increasing: Non-decreasing sorting of boxes implies $\mathrm{wt}(T_i) \leq U(T_{i+x})\ \forall i < n; x = \{1, ..., n-i\}$
- Decreasing: Non-increasing sorting of boxes implies $\mathrm{wt}(T_i) \geq U(T_{i+x})\ \forall i < n; x = \{1, ..., n-i\}$

Afterwards, the core sequence is defined in arbitrary way and remains for the further partitioning process. An empty core P_x has a capacity of $C(P_x) = 1$ at start of partitioning. After assignment of tasks, the actual capacity calculates in the following way:

[1] Optimality in terms of bin packing address following term. Having a bin size V and a list $a_1, ..., a_n$ of sizes of items to pack, find an integer P and a P-Partition $S_1 \cup ... S_P$ of $\{1, ..., n\}$ such that $\sum_{i \in S_k} a_i \leq V\ \forall k = 1, ..., P$. A solution is optimal if P is minimal.

$$C(P_x) = 1 - \sum_{T_y \in \tau; T_y \in P_x} U(T_y)$$

At each assignment, the next task T_i from the ordered list of tasks $\{T_1, ..., T_i, ..., T_n\}$ is selected, starting with task T_1.

For each assignment T_i to P_y, the number of fitting cores $\Re = \{P_y\}$ $\{P_y | P_y \in M \wedge C(P_y) \geq \mathrm{wt}(T_i)\}$ has to be determined from the ordered list of all cores $M = \{P_1, ..., P_x, ..., P_m\}$. From $\Re$, a core P_y is selected by one of the following *assignment policies*:

- **F**irst **F**it: Select from $\Re$ the core with the lowest x.
- **B**est **F**it: Select from $\Re$ the core P_x with $C(P_x) = \min_{P_y \in \Re}(C(P_y))$.
- **W**orst **F**it: Select from $\Re$ the core P_x with $C(P_x) = \max_{P_y \in \Re}(C(P_y))$.
- **N**ext **F**it: When in the previous assignment of T_{i-1} the core P_a has been selected, then select from sequence $\{P_{a+1}, ..., P_m, P_1, ..., P_{a-1}\} \in \Re$ the core at the first position.
- **A**ny **F**it: Select from $\Re$ an arbitrary core P_x.

A certain bin-packing algorithm derives from a combination of *sorting* and *assignment policy*. For example the *FFD* algorithm selects cores with First Fit and sorts tasks in Decreasing order. Lopez et al. [LGDG00] provide a comparison of the different combinations. They showed that FFD, BFD, and WFD allow the highest schedulability bound of utilization. In combination with partitioned preemptive Earliest-Deadline-First (pEDF) scheduling, the proved utilization bound $U_{sum}^{pEDF}(\tau)$ exists for m cores with a maximal task utilization of $U_{max}(\tau)$ for arbitrary periodic task sets with implicit deadline.

$$U_{sum}^{pEDF}(\tau) = \frac{m \left\lfloor {}^{1}\!/\!U_{\max}(\tau) \right\rfloor + 1}{\left\lfloor {}^{1}\!/\!U_{\max}(\tau) \right\rfloor + 1} \quad (3.2)$$

For the unrestricted case $U_{max}(\tau) = 1$, the maximal system utilization is $U_{sum}(\tau) = \frac{m+1}{2}$.

For partitioned Rate-Monotonic (pRM) Scheduling, a utilization bound of $U_{sum}^{pRM}(\tau)$ was derived by [LDG04a], for arbitrary periodic task sets with implicit deadlines, consisting of n tasks and m cores.

$$U_{sum}^{pRM}(\tau) = (n-1)(2^{\frac{1}{2}} - 1) + (m - n + 1)(2^{\frac{1}{(m-n+1)}} - 1) \quad (3.3)$$

However this bound is only valid for periodic task sets and it is not proven to be a tight bound.

3.3.2 Global Scheduling

For the case of global scheduling, ready tasks are sorted in a single queue and one scheduling algorithm allocates them to available cores. In [CFH+04], Carpenter et al. derived that global scheduling with dynamic priority assignment is the only way to achieve a schedulability bound of $U_{sum}(\tau) = m$ for periodic task sets. For sporadic task sets, Fisher et al. [FGB10] showed that there are feasible task sets which can not be scheduled by any online scheduling algorithm, therefore there is no optimal online scheduling algorithm for the general case of any sporadic task set. In the following, global scheduling algorithms from different priority assignment groups are presented.

Task-fix and Job-fix priority scheduling

The application of singlecore scheduling algorithms for global scheduling implies that the m highest priority tasks are selected for execution. However, EDF is no longer optimal for global scheduling. Dhall and Liu [DL78] showed that the uniprocessor EDF utilization bound test does not extend directly to global multicore scheduling. They showed that there are implicit-deadline sporadic task sets τ with $U_{sum}(\tau) = 1 + \epsilon$ for arbitrarily small positive ϵ, such that τ is not schedulable on m cores, for any value of m. Reasons are certain task sets, essentially leaving all but one core idle nearly all of the time. Denoted as Dhall's effect [Dha77], for sporadic task sets with explicit deadlines these poorly behaving examples have two kinds of tasks: tasks with a high ratio of execution time to deadline, and tasks with a low ratio of execution time to deadline. For implicit deadline task sets, at least one task needs a very high utilization [Bak07].

For sporadic task sets with implicit deadline, [GFB03] showed that a sufficient condition for schedulability is

$$U_{sum}(\tau) \leq m - (m-1)U_{max}(\tau). \tag{3.4}$$

For explicit deadlines, minor extensions derive a generalized density based test [GFB03]:

$$\delta_{sum}(\tau) \leq m - (m-1)\delta_{max}(\tau). \tag{3.5}$$

Further schedulability bounds for other kind of task sets can be found in [Bar07].

Task-Section-fix and Dynamic Priority Scheduling

Up to now, two groups of optimal global dynamic priority scheduling algorithms have been proposed: Pfair [BCPV96] and LLREF [CRJ06]. Both use the fluid scheduling concept to determine scheduling policies. Fluid scheduling is a model for optimal resource allocation.

Theorem 3.2
Fluid scheduling *defines: When each task T_i of a periodic implicit deadline task set τ has to be executed with an individual processing speed, where the processing speed is $\delta \cdot \frac{T_i.e}{T_i.p}$ units of time in an interval δ, then all deadlines are met as long $U_{sum} \leq m$.*

Formally, the fluid schedule $\mathit{fluid}(T, t_1, t_2)$ represents the processing time, which has to be assigned to a task T_i during a time interval between t_1 and t_2 with regard to its weight $\mathrm{wt}(T_i)$.

$$\mathit{fluid}(T_i, t_1, t_2) = \mathrm{wt}(T_i) \cdot (t_2 - t_1) \qquad (t_1 < t_2) \tag{3.6}$$

Figure 3.1 shows the arrival of two periodic tasks and the fluid scheduling graph of both tasks. The y-axis of the lower diagram corresponds to the remaining execution time of each task and the graph represents the reduction of execution time by a fluid schedule.

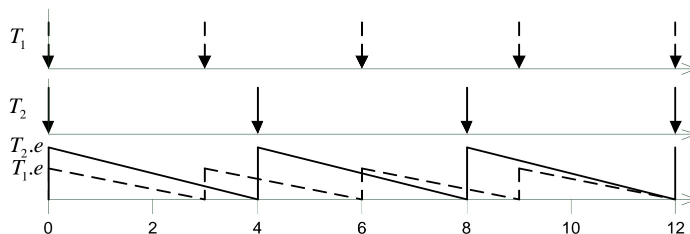

Figure 3.1: Arrival of two periodic task (upper curves) and related fluid scheduling graph (lower curve).

At time 0 both tasks arrive but have different execution times $T_1.e$ and $T_2.e$. Task T_1 has to be finished until $t = 3$ and task T_2 has to be finished until $t = 4$. In order to fulfill fluid scheduling, both task are executed simultaneously with an execution speed, corresponding to Equation 3.6. At time 3 task T_1 is activated again and the remaining execution time rises to the task execution time.

The ideal model of fluid scheduling requires that the processing time can be subdivided arbitrarily and allocated to tasks. Practically there is a bounded resolution in subdivision, caused by the core frequency. Additionally, there is an overhead through context switching. This overhead can be reduced by multiple context registers, as implemented in

many hardware architectures [May09], but overall the overhead is not negligible. Therefore, the fluid schedule model is not applicable for practical purpose. Nevertheless it is used for the algorithms LLREF and Pfair as a model to derive approximations which are also optimal in sense of feasibility.

Least Local Laxity First (LLREF) was proposed by Cho et al. [CRJ06]. LLREF is optimal to schedule a periodic task set with implicit deadlines up to a utilization $U_{sum} = m$. LLREF is based on an abstraction of the time and local execution time domain plane (T-L plane). A T-L plane derives from two successive activations of any periodic task in the task set, as shown in Figure 3.2.

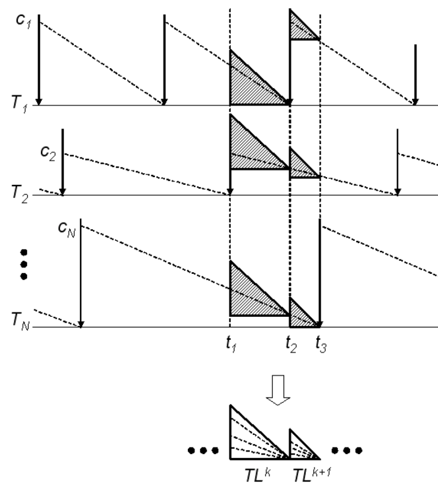

Figure 3.2: LLREF: Determination of T-L plane [CRJ06].

A T-L plane is a right isosceles triangle, where the x-axis is the time and the y-axis is the local remaining execution time. Each job, which is active during this time of the T-L plane, has a token, starting at the left side of a T-L plane. At the beginning, the token of a job $T_{i,j}$ has a local execution time of $l_{i,j} = t_f \cdot \frac{T_i.e}{T_i.p}$, where t_f represents the size of a T-L plane. An example in Figure 3.3 shows the start and the movement of job tokes in a T-L plane.

When a job executes, the token moves downwards in the T-L plane with a gradient of -1. When a job is suspended, the token moves horizontally in the T-L plane. LLREF schedules with a *largest local remain execution time first* policy. At the beginning of a T-L plane, LLREF selects for execution the m tokens with the largest local remaining execution time. These jobs execute until they have no local execution time, or until a ceiling hitting event enters. The *ceiling hitting event* enters, when a token moves horizontally and has no local laxity, which means the job has to execute immediately, otherwise the job will not finish within this T-L plane. When a job has no local remaining execution time, the job suspends until the next T-L plane starts. The following T-L planes

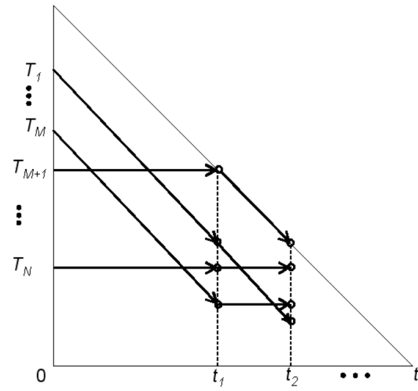

Figure 3.3: LLREF example of token movement through job execution in a T-L plane [CRJ06].

are calculated and processed in the same way. Since jobs have to wait even if there are idle cores, LLREF is a non-work-conserving algorithm.

Cho et al. [CRJ06] proved that LLREF is optimal for periodic task sets with implicit deadlines. They showed that there is a bounded number of job preemptions and a bounded number of scheduling events. In recent work, Funk and Nadadur [FN09] proposed the extension LRE-TL which is optimal for sporadic task sets and reduces the number of migrations by a factor of n in a T-L plane, where n is equal to the number of tasks in a task set.

Proportionate Fair Scheduling

Baruah et al. [BCPV96] introduced Proportionate Fair (*Pfair*) scheduling for periodic task sets with implicit deadlines. *Pfair* is a global multicore scheduling algorithm and preemptively schedules jobs with dynamic task priorities.

As mentioned in Section 3.3.2, *Pfair* is an approximation to the fluid schedule. The approximation uses a quantum time model for the scheduling events task activation, start, suspend, resume, and termination. Task periods $T_i.p$ and task execution times $T_i.e$ of a task set have to be a multiple of the quantum size Q. Events can enter at time quanta $t = \{x \cdot Q\}$ with $x \in \mathbb{N}_0$.

Whenever a task T_i executes, the received processing time increase with time t. When a task is suspended, the received processing time stays constant. The difference between the received processing time $\text{received}(T_i, 0, t)$ and the fluid schedule $\text{fluid}(T_i, 0, t)$ for a task T_i at time t is defined as *Lag*:

$$Lag(T_i, t) = \text{fluid}(T_i, 0, t) - \text{received}(T_i, 0, t)$$

Pfair algorithm's approximation to the fluid schedule is a limitation of the $Lag(T_i, t)$ of each task.

$$-Q < \ Lag(T_i, t) \ < +Q \quad \forall\, T_i \in \tau \tag{3.7}$$

The limitation of $Lag(T_i, t)$ is called *upper* and *lower lag boundary.* Through the combination of lag boundaries and the limitation of scheduling events to time quanta, upper and lower bounds for executing a fraction Q of task execution time derive.

Therefore, a task T_i divides in a number of task quanta[2] Υ_i^l ($l = 1, ..., q$). For practical purposes, it is important to consider that task quanta differ from task sections in Section 2.2.1. A task section defines an enclosed section of software code, which ends at task section end, but a task quantum has no relation to the software code and preemptively suspends the execution at task quanta times, which can be at arbitrary point in a task section through execution time variations. Each Υ_i^l has the execution time of one quantum Q. The number of task quanta q of a task derives from the task execution time $T_i.e$ by Equation 3.8.

$$q = \frac{T_i.e}{Q} \tag{3.8}$$

To fulfill Equation 3.7, a task quantum Υ_i^l has to be scheduled in a Pfair window $w(\Upsilon_i^l)$, starting with the pseudo[3]-release $r(\Upsilon_i^l)$ and ending with the pseudo-deadline $d(\Upsilon_i^l)$. A certain task quantum is called *pseudo-activated*, when its pseudo-release time is elapsed.

$$r(\Upsilon_i^l) = \left\lfloor \frac{l-1}{\mathrm{wt}(T_i)} \right\rfloor \tag{3.9}$$

$$d(\Upsilon_i^l) = \left\lceil \frac{l}{\mathrm{wt}(T_i)} \right\rceil \tag{3.10}$$

The smallest time a task quantum can be completely processed is called slot S. Depending on the weight of a task T_i, a Pfair window $w(\Upsilon_i^l) = \{S_1, S_2, ..., S_{\text{end}}\}$ has a number of slots, available to schedule a task quantum Υ_i^l. ($\left|w(\Upsilon^l)\right| = |\{S_1, S_2, ..., S_{\text{end}}\}|$ denotes the quantity of slots of a Pfair window.)

$$\left|w(\Upsilon_i^l)\right| = \left\lceil \frac{l}{\mathrm{wt}(T)} \right\rceil - \left\lfloor \frac{l-1}{\mathrm{wt}(T)} \right\rfloor \tag{3.11}$$

Figure 3.4 shows the fluid schedule of an example for a job of task T_1, including lag boundaries and Pfair windows. The dotted line represents the fluid scheduling for a task T_1 with execution time $T_1.e = 3$, inter-arrival time $T_1.p = 5$, and quantum $Q = 1$. The dashed lines represent the lag boundaries, which have the same gradient as the fluid schedule, but have a y-offset for the upper and lower lag boundary with $+Q$ and $-Q$, respectively. The Pfair windows of the time quanta Υ_i^1, Υ_i^2, and Υ_i^3 are shown as

[2] Υ_i^l denotes the kth quantum of task T_i.

[3] The appendix *pseudo* is used to differ between task and task quantum properties.

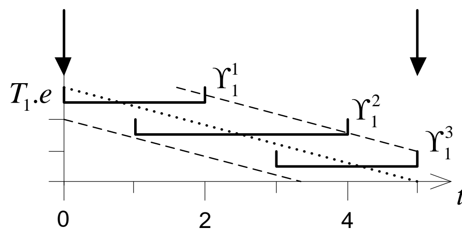

Figure 3.4: Approximation of *Pfair* to the fluid schedule. The fluid schedule defines that the execution time $T_1.e = 3$ of a task is allocated to a processor between two successive activations (arrows, $T_1.p = 5$) in a way that the remaining execution time constantly decreases (dotted line). *Pfair* defines a minimum and maximum lag (dashed line) from the fluid schedule in continuous time. In discretized time, task quanta have to execute in windows (continuous line, $\Upsilon_1^1, \Upsilon_1^2, \Upsilon_1^3$), derived from lag boundary.

thick continuous line and they are defined by the time intervals $(0, 2]$, $(1, 4]$, and $(3, 5]$ respectively. Due to the limitation of events to time quanta, scheduling events like task suspension or resume can only enter at $t = \{0, 1, 2, 3, 4, 5\}$. The scheduler has to start task quanta Υ_1^1 at $t = 0$ or $t = 1$, otherwise task quanta has not finished until $t = 2$, where the task quanta's window ends and the pseudo-deadline would be violated. Since task quanta are sequentially dependent, they can not execute before the previous task quantum has finished. For the case that task quanta Υ_1^2 is started at $t = 1$, task quanta Υ_1^3 can not execute before $t = 3$, otherwise the pseudo-release time is not reached and the lower lag boundary would be ignored.

Proportionate Fair scheduling is a mechanism to assign periodic task sets to multiple cores. For scheduling decisions, a policy or a number of policies is required for prioritization. Up to now, four optimal algorithms have been proven as optimal for scheduling synchronized periodic task sets on multicore processors: *PF* [BCPV96], *PD* [BGP95], PD^2 [AS00a], and *BF* [ZMM03]. All algorithms schedule task quanta by Earliest-Pseudo-Deadline-First (EPDF) policy. For the case, that task quanta of different ready tasks have the same pseudo-deadline, the algorithms apply tie-breaking rules, depending on the algorithm.

One tie-breaking rule that is common to all Pfair algorithms (excepted BF) is called overlapping-bit. The *overlapping-bit* concerns the position of two successive Pfair windows. When the pseudo deadline of Υ_i^l's window is later than the pseudo-release time of Υ_i^{l+1}'s window, the overlapping bit $b(\Upsilon_i^l)$ is one, otherwise it is zero.

$$b(\Upsilon_i^l) = \left\lfloor \frac{l}{wt(T_i)} \right\rfloor - \left\lceil \frac{l}{wt(T_i)} \right\rceil \tag{3.12}$$

The following part describes the Pfair algorithms.

The Pfair-PF Algorithm

Pfair-PF prioritizes tasks as follows: at schedule decision, when task quanta Υ_x^a and task quanta Υ_y^b are ready to execute, Υ_x^a has a higher priority than Υ_y^b, denoted $\Upsilon_x^a \succ \Upsilon_y^b$, if one of the following holds:

i) $d(\Upsilon_x^a) \ < \ d(\Upsilon_x^b)$,

ii) $d(\Upsilon_x^a) \ = \ d(\Upsilon_x^b)$ and $b(\Upsilon_x^a) \ > \ b(\Upsilon_y^b)$,

iii) $d(\Upsilon_x^a) \ = \ d(\Upsilon_x^b)$, $b(\Upsilon_x^a) \ = \ b(\Upsilon_y^b)$, and $\Upsilon_x^{a+1} \succ \Upsilon_y^{b+1}$

At each start of a slot, at $t = a \cdot Q$, the m highest priority tasks are selected and get processed for one time quanta. At time $t = (a+1) \cdot Q$, all running tasks getting suspended and the procedure starts again. However, the algorithm Pfair-PF is quite inefficient due to the recursive calculation of policy *iii*. Therefore, Anderson et al. replaced the last policy by a non-recursive policy and presented the algorithms Pfair-PD^2 and Pfair-PD.

Algorithm Pfair-PD^2

Pfair-PD replaces policy *iii* of Pfair-PF by three rules. Since Pfair-PD^2 requires only one of these rules Pfair-PD is less efficient than as Pfair-PD^2. Therefore, this work considers only Pfair-PD^2 and refer to [BGP95] for Pfair-PD. The new policy concerns the effect of a scheduling decision for task quanta Υ_i^l on successive task quanta and is called *group-deadline* $D(\Upsilon_i^l)$. Assume there is a sequence of task quanta Υ_i^l with $l = \{a, ..., b\}$ and a Pfair window $|w(\Upsilon_i^l)| = 2$ and $b(\Upsilon_i^l) = 1$. Then, processing Υ_i^a in its last slot results in processing all other task quanta in their last slot. Therefore, the group deadline for all task quanta Υ_i^l of this sequence is the pseudo-deadline of the last task quantum Υ_i^b in the sequence, because after this task quantum there are at least 2 slots left for scheduling.

Anderson et al. [AS01] proved that the constellation $|w(\Upsilon_i^l)| = 2$ and $b(\Upsilon_i^l) = 1$ only occurs with high weighted tasks, meaning $0.5 \geq wt(T_i) < 1$. The group deadline for high weighted task calculates by Equation 3.13.

$$D(\Upsilon_i^l) = \left\lceil \frac{\left\lceil \left\lceil \frac{l}{wt(T_i)} \right\rceil \ \cdot \ (1 - wt(T_i)) \right\rceil}{1 - wt(T_i)} \right\rceil \text{ if } 0.5 \geq wt(T_i) < 1 \tag{3.13}$$

For light task, when $0 < w(T_i) < 0.5$, the group deadline is 0 for all task quanta.

$$D(\Upsilon_i^l) = 0 \text{ if } 0 < wt(T_i) < 0.5 \tag{3.14}$$

At Pfair-PD^2, the policies *i* and *ii* are equal to Pfair-PF policies. Policy *iii* is replaced as follows:

iii) $d(\Upsilon_x^a) \ = \ d(\Upsilon_x^b)$, $b(\Upsilon_x^a) \ = \ b(\Upsilon_y^b)$, and $D(\Upsilon_x^a) < D(\Upsilon_y^b)$

The reason of this prioritization is that scheduling a task quantum with a higher group deadline prevents (or at least reduces the extend of) *cascades*, meaning a task quantum is possibly processed in a slot within its schedule window which overlaps with the schedule window of the subsequent task quantum. Cascades are undesirable since they constrain the scheduling of future slots. Anderson et al. [AS01] proved that the *Pfair* scheduling algorithm *PD*[2] is optimal for scheduling a task set τ in a multicore processor system with m cores iff Equation 3.15 holds.

$$\sum_{i=1}^{n} \mathrm{wt}(T_i) \leq m \tag{3.15}$$

Figure 3.5 illustrates a schedule of Pfair-PD[2] for a synchronized periodic task set $\tau = \{T_1, T_2\}$, with $T_1 = (10, 6, 10, -, -)$ and $T_2 = (7, 3, 7, -, -)$ on a singlecore processor[4]. Three representative schedule points are considered: At timestamp 0 *Pfair-PD*[2] prefers the task quantum of task T_1 before the task quantum of T_2, because $d(\Upsilon_1^1) = 2 \succ d(\Upsilon_2^1) = 3$. At timestamp 3 *Pfair-PD*[2] prefers the task quantum of task T_2 before the task quantum of T_1, because $d(\Upsilon_1^3) = d(\Upsilon_2^2) = 5$ and $b(\Upsilon_1^3) = 0 \prec b(\Upsilon_2^2) = 1$. At timestamp 10, it is arbitrary if the task quantum of task T_2 is preferred before the task quantum of T_1 or otherwise, because $d(\Upsilon_1^1) = d(\Upsilon_2^2) = 5$, $b(\Upsilon_1^1) = b(\Upsilon_2^2) = 1$, and $D(\Upsilon_1^1) = D(\Upsilon_2^2) = 0$.

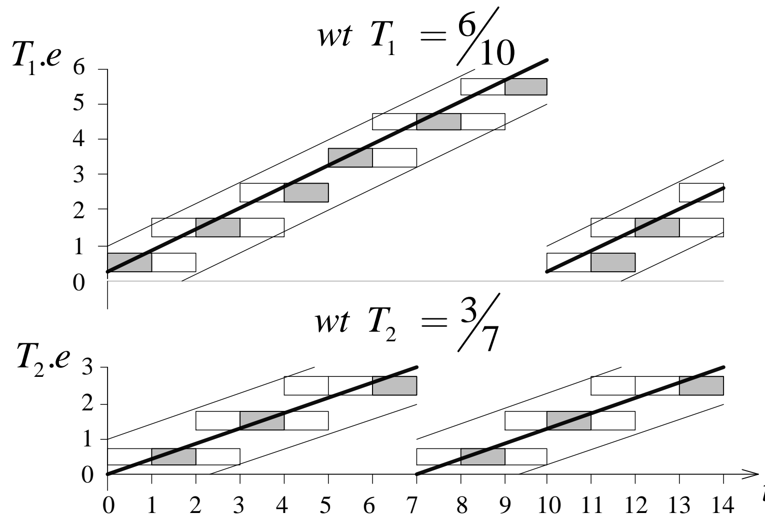

Figure 3.5: Example of a schedule of algorithm Pfair-PD[2]. The x-axis represents the time and the y-axis defines the processed execution time. Task quanta (grey) are processed between the upper and lower lag boundary (slim line) which results from the fluid schedule graph (thick line).

[4] *Pfair-PD*[2] scheduling on multicore processors behaves analogously, with the difference that instead of one task, m tasks are selected for execution.

Algorithm BF

In [ZMM03], Zhu et al. addressed the problem of a high number of context switches at Pfair scheduling and proposed an extended Pfair algorithm, called Boundary Fair (BF).

In systems with a periodic implicit deadline task set, a task can miss its deadline only at inter-arrival time boundaries. Using this assumption, Zhu et al. construct a scheduling algorithm which makes schedule decisions only at inter-arrival time boundaries and allows a violation of the lag condition (Equation 3.7) during this inter-arrival time boundaries, as long as the lag condition holds at inter-arrival time boundaries.

BF determines how many task quanta of all tasks in a task set have to be assigned to the cores from schedule time until next inter-arrival time boundary and place these task quanta in a way that the context switches are minimal. Although the BF algorithm has the same complexity as all other Pfair algorithms, the number of scheduling points decrease dramatically in practice. In experiments, the number of of schedule points could be reduced up to 75% [ZMM03]. Since the task activation has to be known, Pfair-BF is limited to periodic task sets.

Early Release Extension

Pfair scheduling assigns a task quantum between its pseudo-activation and pseudo-deadline to cores. Due to the pseudo-activation of task quanta, a ready task can only be assigned to a core, when time exceeds pseudo-activation of the actual task quantum. Especially at low core utilizations, this leads to non-work-conserving behavior, because task quanta are often executed at the beginning of their window and as long window size is larger 2 or the overlapping bit is zero the pseudo activation of the next task quantum is not reached at finalization of the actual task quantum. Non-work-conserving scheduling has a number of benefits, e.g. reduced task jitters, which is shown in detail in the experiments of Section 10. Unfortunately, in general non-work-conserving scheduling is not as robust as work-conserving scheduling, which is derived from the experiments of Section 10, too.

Anderson and Srinivasan proposed in [AS00a] a work-conserving version of Pfair scheduling, called Early-Release fair (ERfair) scheduling. ERfair scheduling defines that task quanta have no longer a pseudo-activation time and therefore, when a task quantum Υ_i^{l-1} finishes before pseudo-release time of the next task quantum at $t < r(\Upsilon_i^l)$, Υ_i^l is allowed to be processed.

Chapter 4

Real-Time System Examination

This chapter gives an overview of existing methods for the evaluation of real-time systems, i.e. the verification whether the real-time requirements of tasks in a task set are fulfilled. First of all, fundamental information on real-time evaluation is given. Then, schedulability and performance evaluation methods are presented. Finally, the difference between singlecore and multicore schedulability evaluation is discussed.

4.1 Fundamentals

This section presents necessary definitions and metrics for the evaluation of performance and temporal robustness properties of real-time system models.

4.1.1 Definitions

Several fundamental definitions are required, common in real-time theory, which will be introduced in the following.

Classical real-time theory contains two problems: *feasibility* and *schedulability* analysis [BG04]. The feasibility-analysis problem concerns: when given a certain task set, a processor model, and constraints on the scheduling environment (e.g. a global scheduling algorithm with dynamic priority assignment and full-migration, see also Table 3.2), determine whether there is any schedule for the task set that will meet all deadlines. The constraints on the scheduling environment are necessary to limit the possible schedule operations on a job sequence. The complexity of scheduling a job sequence strongly depends on the kind of task set. As introduced in Section 2.3.1, recurrent task sets produce one single sequence of jobs. The request behavior of a task is defined by the task offset $T_i.o$ and the task inter-arrival time $T_i.p$; the execution time of task $T_i.e$ is constant. These kind of task sets are feasible, as long as for the collection of jobs in this sequence all deadlines are held. Event-based task sets produce an undefined, usually very high, number of sequences of jobs. The only limitation is the upper and lower arrival curve and the discrete resolution of accepted time stamps of task activations. This kind of task

sets are feasible, as long as for the collections of jobs of all sequences all deadlines are held.

Definition 4.1
A task set is said to be ***feasible*** *with respect to a given system if there exists any scheduling algorithm that can schedule all possible sequences of jobs that may be generated by the task set on that system without missing any deadline [DB09]. A* ***feasibility test*** *validates if a certain task set is feasible.*

The schedulability-analysis problem concerns: when given a certain feasible task set, determine whether a certain scheduling algorithm can schedule the task set to meet all deadlines. Similarly to the feasibility of a task set, for recurrent task sets, a schedulability test has to determine if the scheduling algorithm produces a schedule for the sequence of jobs, such that for the collection of jobs of the sequence all deadlines are held. For non-periodic event-based task sets, a schedulability test has to determine if the scheduling algorithm produces schedules for all possible sequences of jobs, such that for all collections of jobs of all sequences all deadlines are held.

Definition 4.2
A task is referred to as ***schedulable*** *according to a given scheduling algorithm, if its worst-case response time under that scheduling algorithm is less than or equal to its deadline [DB09]. A task set is referred to as schedulable according to a given scheduling algorithm if all of its tasks are schedulable. A* ***schedulability test*** *checks if a certain task set is schedulable.*

The applicability of a schedulability test depends on the task set, the scheduling algorithm class, and the processor model. Most schedulability tests are scheduling algorithm specific, meaning they are only applicable to a certain algorithm or a class of algorithms. This work gives an overview of the most relevant schedulability tests in Section 4.2. Before, schedulability tests are classified according to their accuracy.

Schedulability tests are mainly driven from hard real-time system theory, where a deadline violation results in a system destruction or comparable perilous situation. Embossed by this fact, the class of *pessimistic* schedulability tests was developed, rather judging a task set as non-schedulable, when the method can not solve the problem more precisely.

Definition 4.3
A ***pessimistic*** *schedulability test surveys sufficient conditions for schedulability, meaning a positive outcome guarantees that all deadlines are always met.*

Sufficient but not necessary test have a lower runtime complexity, but they are pessimistic [SAC$^+$04]. Figure 4.1 shows the iteration steps of a pessimistic schedulability

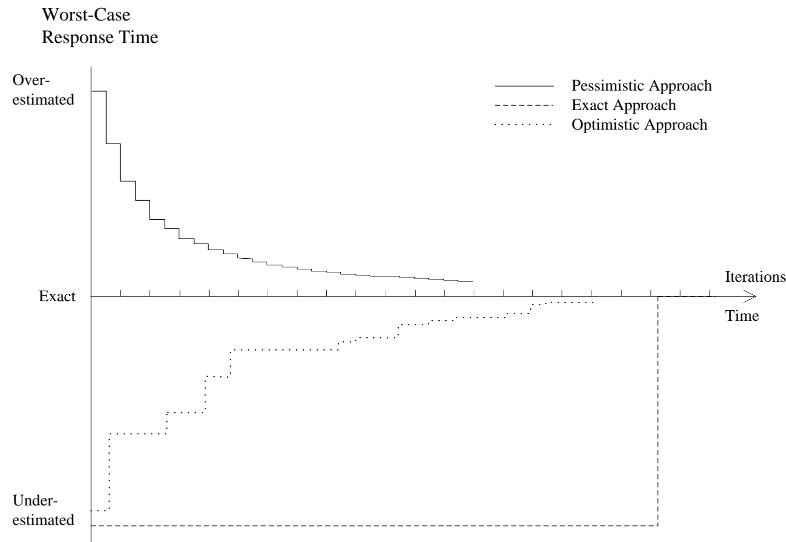

Figure 4.1: Examination methodologies of worst-case response time. The x-axis represents the number of iterations (for iterative approaches) or the calculation time (e.g. for simulation approaches) and the y-axis represents the estimation of the quality criteria. A pessimistic approach approximates the worst-case response time from the upper side (overestimated) and the optimistic approach from the lower side (underestimated). An exact approach determines the worst-case response time directly.

test. Each step approximates closer to the real schedulability result. The obtainment of the maximal response-time for a task is not guaranteed for all kinds of schedulability tests. Since a pessimistic test can produce a result which overestimates the real value, system resources are not used efficiently, e.g. processor computation capacity is wasted. For particular cases this is acceptable, e.g. when over-dimensioning of system resources is non-relevant.

When efficient dimensioning of system resources is relevant, a pessimistic test is often not appropriate and another class of tests is required. The class of optimistic schedulability tests rather judge a task set as schedulable, when the method can not solve the problem more precisely. Since this behavior is non-consistent with the definition of test, this kind of approaches are also called *schedulability examinations* .

Definition 4.4
*A **optimistic** schedulability examination surveys necessary conditions for schedulability, meaning there might be a deadline miss at some point during the execution of the system.*

Figure 4.1 shows that the result of an optimistic schedulability examination approximates over the time to the exact value.

The class of exact schedulability tests determines always the accurate result.

Definition 4.5
*An **exact schedulability test** surveys necessary and sufficient condition for schedulability.*

Unfortunately this kind of approaches are mainly only applicable for very simple task set configurations, due to a state explosion. Therefore, sufficient and necessary tests are ideal, but for many computational models such tests are intractable [SAC+04].

4.1.2 Real-Time Metrics

This section describes metrics for system evaluation, applied to the schedule of a task set. The term „metric“ is used in the parlance of software engineering, denoting a measure of some property of a piece of software or its specifications. This meaning differs from the strong definition of a mathematical metric.

In the following, these metrics are used to compare scheduling approaches. It is differed between the following task job metrics, which can be used for real-time examination on task level.

Response Time The *response time* describes the required time for the *Finalization* of a job of a task.

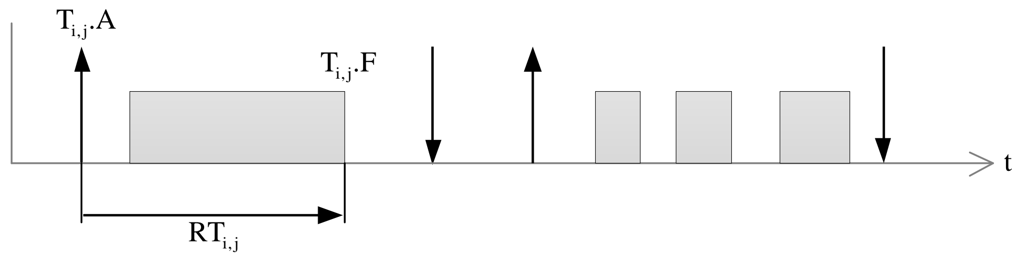

Figure 4.2: Task job metric response-time, measured between the activation $T_{i,j}.A$ and the finalization $T_{i,j}.F$ of a job $T_{i,j}$.

$$RT_{i,j} = T_{i,j}.F - T_{i,j}.A \tag{4.1}$$

The response time is the fundamental property to measure the task deadline compliance, namely when condition $RT_{i,j} < d_i$ is fulfilled.

Lateness A further metric which characterizes the deadline compliance, i.e. the delay of a task, is called *lateness* .

$$L_{i,j} = T_{i,j}.F - T_{i,j}.D + T_{i,j}.A \tag{4.2}$$

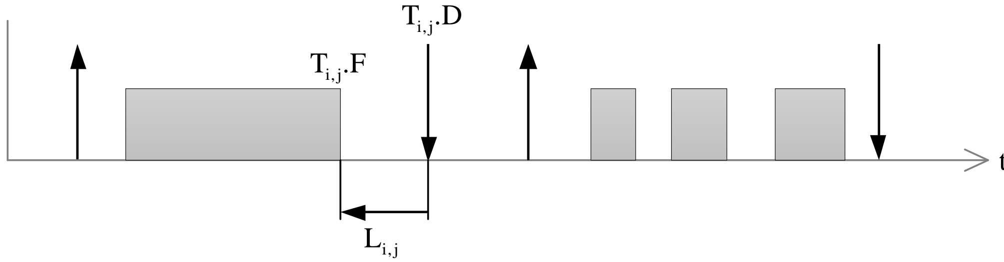

Figure 4.3: Task job metric lateness, measured between the deadline $T_{i,j}.D$ and the finalization $T_{i,j}.F$ of a job $T_{i,j}$.

The benefit of the lateness in comparison with the response time is a direct relation to the deadline miss. When the lateness is negative, the task deadline is held. The lateness can be calculated from response time through Equation 4.3.

$$L_{i,j} = RT_{i,j} - T_{i,j}.D \tag{4.3}$$

End-to-End The previous metrics concern the deadline compliance, but e.g. for sampling systems it is not necessary that a task returns a computation result until a deadline has reached. Instead it is required that the result of a periodic calculation is returned in a constant time-interval with less jitters. The *end-to-end* jitter is one of such metrics.

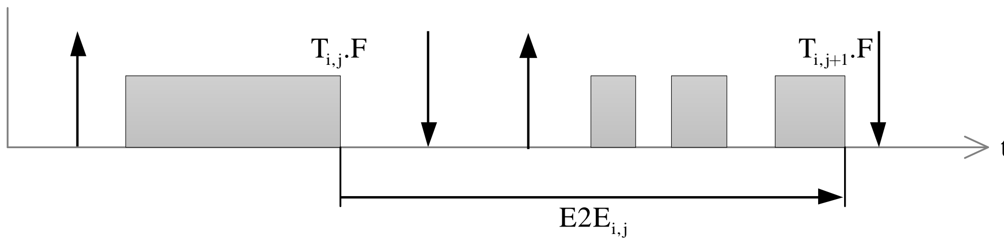

Figure 4.4: Task job metric End-to-End, measured between the finalization $T_{i,j}.F$ of two subsequent jobs $T_{i,j}$ and $T_{i,j+1}$.

$$E2E_{i,j} = T_{i,j+1}.E - T_{i,j}.E \tag{4.4}$$

Mostly, this metric is used to evaluate the temporal behavior for periodically triggered calculation, e.g. in multimedia applications. For example, consider a computer communication network that carries some packet voice connections for web camera conversations. For those connections that carry voice and video packets, delays are disruptive to the

application, which needs to feed data at a constant rate to the conversation. Otherwise the quality of acoustic and visual conversation decreases.

Start-to-Start The counter part of the end-to-end jitter is the start-to-start jitter. When the end-to-end jitter is task output oriented, the start-to-start jitter is task input oriented. It assumes that a task reads data from a source at beginning of the task. When this task is the element of a control cycle which collects data with a certain sampling rate, then it is important that the data is collected in a constant time-interval.

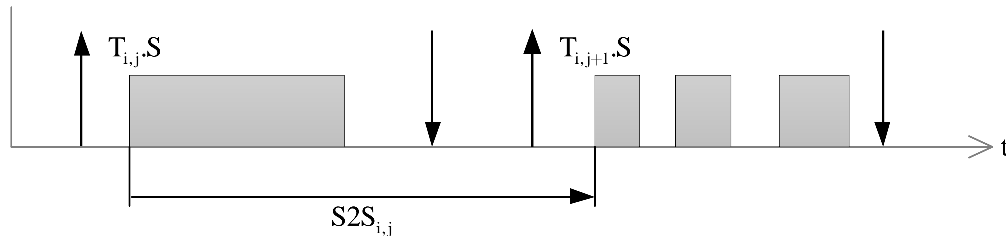

Figure 4.5: Task job metric Start-to-Start, measured between the start $T_{i,j}.S$ of two subsequent jobs $T_{i,j}$ and $T_{i,j+1}$.

$$S2S_{i,j} = T_{i,j+1}.S - T_{i,j}.S \tag{4.5}$$

The start-to-start jitter is used for periodically triggered calculation, too. Both metrics start-to-start and end-to-end are metrics for soft real-time systems.

4.2 Methods for Schedulability Examination

This section surveys existing approaches of schedulability examination and gives a brief overview of the mechanisms. Furthermore, the application to multicore processor scheduling is discussed. The various approaches are very heterogeneous in terms of modeling scope, modeling effort, tool support, accuracy and scalability. Most of the approaches for performance analysis proposed so far can broadly be divided into the two main classes of analytic techniques and simulation-based methods [PWTH09].

4.2.1 Response-Time Analysis

The Response Time Analysis (RTA) [JP86, Leh90, ABR+93] is an exact analytical schedulability test. The basic principle of the RTA is to determine the worst-case interference that a task can suffer from higher priority tasks. The approach uses a fix point iteration method which successively extends the window length in which the execution

time of higher priority tasks is considered in order to determine the interference in this window. Equation 4.6 shows the fundamental iterative formula for calculating the response time R_i of a periodic task T_i, which was extended for many other properties like task offsets [MTN08]. For all higher priority task $hp(T_i)$, it is determined how many activations from task T_j with a minimal inter-arrival time p_j enter in the response time R_i^n of the current iteration n. This number of activations is multiplied with the execution time e_j. Finally the execution time of task T_i is added. This response time R_i^{n+1} is used for the next iteration. The calculation terminates when $R_i^n = R_i^{n+1}$.

$$R_i^{n+1} = e_i + \sum_{j \in hp(T_i)} \left\lceil \frac{R_i^n}{p_j} \right\rceil \cdot e_j \tag{4.6}$$

The approach can be applied to determine worst-case and best-case bounds of response time. This analysis guarantees that all observable response times will fall into the calculated [best-case, worst-case] interval [HHJ+05]. Initially developed for periodic task sets and task-fix priority scheduling [JP86], several extensions for support of task release jitters, conditioned deadlines, and burst activation have been presented [ABR+93]. A further extension of the RTA is the compositional system analysis [HHJ+04, HHJ+05]. Compositional system level analysis alternates local component analysis and output event model propagation. More precisely, in each global iteration of the compositional system level analysis, local analysis is performed for each component to derive the output event models. An output event stream of one component turns into an input event stream of a connected component in order to reach a global analysis result. The SymTA/S [RE02, RZJE02, RRE03, RJE03] approach couples local scheduling analysis algorithms by using event streams. Event functions resemble arrival curves [Cru02] which have been successfully used by Thiele et al. [TCGK02] for compositional performance analysis of network processors. They are piecewise constant step functions with unit-height steps, each step corresponding to the occurrence of one event.

For the singlecore case, Baruah et al. [BB06] proved that RTA is sustainable (see Definition 4.8) with respect to all task set parameters, and that RTA remains sustainable even when additional non-preemptable resources must be shared among jobs of different tasks. However, for analysis of global scheduling algorithms for multicore system the RTA is not appropriate, because it determines the maximal interference on a task, which is the worst-case scenario at local scheduling but not at global scheduling (see Section 4.3).

4.2.2 Real-Time Calculus

The Real-Time Calculus (RTC) [CKT03] is a performance analysis approach of distributed embedded systems. The Real-Time Calculus is based on the well-know Network Calculus [LBT01] which is based on max-plus algebra [BCOQ92]. Instead of describing the minimum inter-arrival time, as it is done at sporadic task sets, and instead of record-

ing the precise arrival times of events, as it is done at hardware measurements, RTC uses a count-based abstraction. Upper and lower arrival curves, $\beta^u(\Delta)$ and $\beta^l(\Delta)$, define the maximal and minimal number of events as a function of a time interval Δ. Each event activates a task and the requested execution time in a time interval can be derived by multiplying the events with the execution time. Comparable to the arrival curves, the upper and lower *service curves* define the free capacity of a resource as a function of a time interval Δ. The RTC allows to process a very general model of arrival curves and service curves beyond the classical event models such as periodic, sporadic, periodic with jitters, etc. [PTCT07]. There are different resource sharing mechanisms available like task-fix priority scheduling, EDF, TDMA and generalized processor sharing (GPS) which assigns to each active task a processor share proportional to its utilization in a perfectly fair manner.

In [LCA09], Leontyev and Chakraborty introduced an extension of the RTC in order to support global scheduling algorithms with job-fix priorities like global EDF. Beside the arrival curve of tasks, an assumption of response time of all tasks is required. With these parameters the extension is able to check whether the response time assumptions can be met.

4.2.3 Model-Checking Approach

Another approach of real-time system examination is the application of model-checking algorithms, using e.g. timed automata [AD94] (TA) for the specification of real-time systems. In [Cor94], TA was used for modeling preemptive scheduling with task-fix priorities. In [HV06], a general approach was shown which determines temporal properties of real-time systems. The benefit of Model-checking is the determination of exact upper bounds on temporal properties like the response time.

In [MHK+08], the application of TA to multicore processors with local scheduling algorithms with static or dynamic priorities has been shown. However, due to state explosion, the application is often limited for more complex real-time systems with a high number of tasks [GGD+07] or different clocks [ALM10].

4.2.4 Simulation Approach

Simulation-based approaches for schedulability examination use a model which describes the temporal behavior and a systematic approach for changing model parameters in order to determine the worst-case response time. Samii et al. [SREP08] proposed a simulation-based methodology for worst-case response time estimation of distributed real-time systems with periodic task sets and variable task execution times. The discrete-event simulation was based on top of System–C [Ope10]. Since an exhaustive search of all execution times is not feasible in adequate time, Samii et al. analyzed methods for reducing the exploration space of execution times and how to explore the reduced

space efficiently in order to determine the worst-case response time. In [LNKN10], Lu et al. proposed a simulation-based approach for schedulability examination for task sets containing execution dependencies. In the presented approach, the execution time is randomly varied in the specified range and multiple simulation runs were performed. Finally, by application of Extreme Value Theory [BGST04], an approximation of the distribution function of the maximal response-times of all runs was derived in order to estimate the worst-case response time.

4.3 Anomalies of Multicore Schedulability Analysis

This section shows why schedulability analysis of global multicore scheduling differs from schedulability analysis of singlecore scheduling. Furthermore, the background and theorems of singlecore schedulability analysis are presented and it is shown why a lot of these assumptions fail at multicore systems, which requires new schedulability tests or schedulability examination methods. First of all, the following definitions are required:

Definition 4.6
Anomalies *consider the case when a task set* τ' *with weaker properties, meaning a lower execution time* $T'_i.e$*, a lower inter-arrival time* $T'_i.p$*, or higher deadline* $T'_i.d$ *of one or more tasks* T'_i *in comparison with the corresponding task* T_i *in a task set* τ*, exceeds*[1] *the worst case of task set* τ*.*

Definition 4.7 ([Ber07])
A scheduling algorithm A is ***predictable*** *if and only if the A-schedulability of a task set* τ *implies the A-schedulability of another task set* τ' *with identical arrival times and deadlines, but smaller execution requirements.*

The concept of predictability of scheduling algorithms can be extended to feasibility and schedulability tests. A schedulability test is predictable, if a schedulable task set τ remains schedulable for a modified task set τ' with reduced temporal requirements (in terms of predictability).

Definition 4.8 ([BB06])
A schedulability test for a scheduling policy is ***sustainable*** *if any system deemed schedulable by the schedulability test remains schedulable when the parameters of one or more individual job(s) are changed in some or all of the following ways: (i) decreased execution requirements; (ii) later arrival times; (iii) smaller jitter; and (iv) larger relative deadlines. A non-sustainable system remains not schedulable under this conditions.*

[1] A quantitative formulation of „exceeds“ can be given by use of the maximum normed lateness (mNL, Equation 8.8): $mNL(\tau') > mNL(\tau)$.

In general, schedulability tests are based on the *critical instant* theorem. A critical instant for a task is a release time for which the response time is maximized (or exceeds the deadline, in the case where the system is overloaded enough that response times grow without bound [SAC+04]). Using task offsets, i.e. asynchronous task activation,

Theorem 4.1 ([LL73] and [BMR90])
For singlecore systems with periodic task sets and preemptive task-fix priority [LL73] or dynamic priority [BMR90] scheduling the critical instant occurs when a task starts with all other higher priority task at the same time, i.e. tasks have synchronous activation.

Using task offsets, i.e. asynchronous task activation, allows to improve the worst-case response time, because the response time of a periodic task is better or not worse than the critical instant. However, at asynchronous task activation with jitters, the schedulability test is not sustainability anymore, because activating a task later can result in a higher response time [BB06].

Global multiprocessor scheduling is intrinsically a much more difficult problem than uniprocessor scheduling due to the simple fact that a task can only use one processor at a time, even when severals are free [LL73].

Considering only implicit deadline systems, it is possible to prove that Pfair algorithms are sustainable for periodic and sporadic task sets: increasing the inter-arrival time or decreasing the computation time of any task, the total utilization decreases and the inequality $U_{max}(\tau) \leq m$ is still valid and the task set is still P-fair-schedulable [Ber07].

4.3.1 Examples

This section gives examples, showing why multicore schedulability considerations differ from singlecore schedulability considerations.

The Example 4.1 (proposed in [Bar07]) shows that for global job-fix priority scheduling of sporadic task sets, the critical instance theorem is not valid.

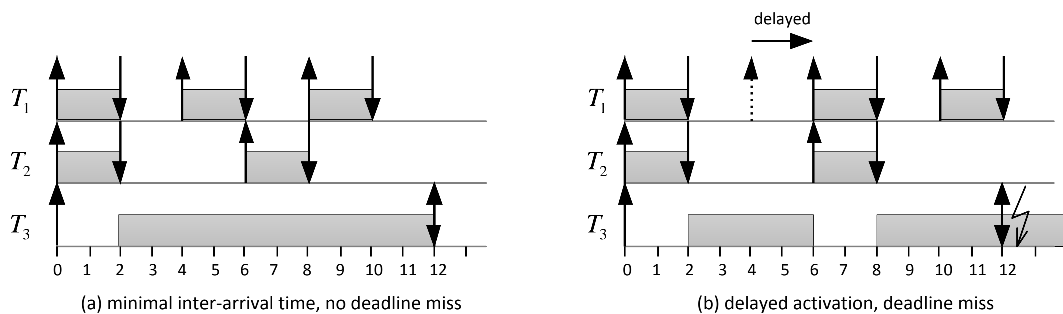

Figure 4.6: Example 4.1: Multiprocessor Anomalies - sporadic activation. Dualcore processor with three tasks.

Example 4.1

For a sporadic task set $\tau = \{(2,1,1,-,-),(6,1,1,-,-),(12,10,12,-,-)\}$*, a processor with* $m = 2$*, and EDF scheduling, the synchronous activation and minimum inter-arrival time of all tasks doesn't represent the worst-case. This is shown in Figure 4.6 where a delay of task* T_1 *of one time unit at the second job, results in a deadline miss of task* T_3 *at time* 7 *(b), whereas all tasks meet their deadline at a minimal inter-arrival time (a).*

Also for periodic task sets, simultaneous release does not has to be the scenario that represents critical instant, as shown in Example 4.2 (proposed in [DB09]).

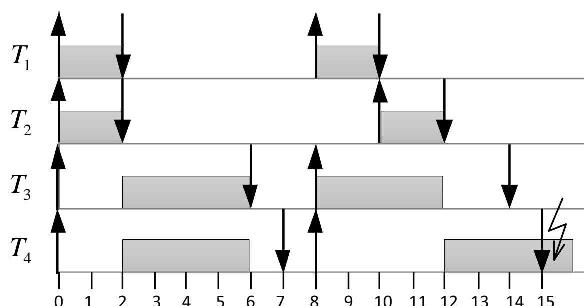

Figure 4.7: Example 4.2: Multiprocessor Anomalies - synchronized periodic activation. Dualcore processor with four tasks.

Example 4.2

For a periodic task set $\tau = \{(8,2,2),(10,2,2),(8,4,6),(8,4,7)\}$*, scheduled on a processor* $m = 2$*, the critical instance for task* T_4 *may not be in the interval* $[0,7)$ *when all tasks are activated synchronized, as shown in Figure 4.7. However, when higher priority task block all processors at the time* $[8,12)$*, then task* T_4 *misses its deadline at* 15*.*

Therefore, the maximal interval where ***all*** processors are occupied by higher priority tasks results in worst-case response time. This shows that the critical instant assumption for singlecore / local multicore scheduling, which assumes that the maximal interval of higher priority tasks on any processor results in worst-case response time, can not be applied to global multicore scheduling.

Part II

Contribution

Chapter 5

Focus of Contribution

The objective of a real-time analysis is the determination whether a given real-time task set fulfills all temporal requirement on a given processing resource with a defined scheduling approach.

This work extends this objective by focusing on the performance and temporal robustness of task-sets, executing on embedded multicore processors.

This work defines *performance* for embedded real-time systems in the following way:

Definition 5.1
Performance *is the capability of an embedded real-time system to execute a workload on a processing resource and to fulfill all temporal requirements. The ratio between requested and theoretically available processing time is called utilization. A high performance system allows a maximal utilization.*

For an embedded real-time system, high performance can be interpreted in two ways. On the one hand, high performance means to be able to execute real-time workload up to a maximal task-set utilization to the available processors. On the other hand, high performance means to be able to reduce the number or the processing speed of the available processors in a way that the real-time workload produces a maximal task-set utilization. The first interpretation relates to maximizing functionality, the second one relates on maximizing energy efficiency and minimize hardware costs.

The second investigation purpose is temporal robustness. This work defines *temporal robustness* for embedded real-time systems in the following way:

Definition 5.2
Temporal robustness *describes the property of an embedded real-time system to be capable to fulfill its temporal requirements despite the presence of transient or permanent*

perturbations. Temporal robustness is dependent on the kind, intensity, and duration of the perturbation.

A perturbation must be distinguished from a fault. In contrast to fault-tolerance, where the fault-hypothesis precisely specifies the faults that must be tolerated, the concept of a perturbation is intentionally ill-specified. The condition which will perturb the operation of a system is not exactly known. A perturbation can thus be a hardware fault, a software fault, a changing specification, an unanticipated behavior of the environment, or any other phenomenon that impacts the system [Kop08]. It is important to distinguish between transient and permanent perturbations. A permanent perturbation impacts some parts of the system unendingly and thus requires a reconfiguration or repair action, whereas a transient perturbation is temporary and disappears by itself [Kop08].

Robustness in scheduling can be achieved in several ways. Stankovic et al. [Sta98] define robust scheduling as follows: "Whenever a new [job of a] task enters the system, an acceptance test verifies the schedulability of the new task set based on worst-case assumptions. If the task set is found schedulable, the new task is accepted; otherwise, one or more tasks are rejected based on a different policy." This requires a forecast, at least in execution time and activation demand. Unfortunately, the nature of a perturbation is an unexpected entrance of a temporal property variation, possibly exceeding worst-case assumptions.

The definition of temporal robustness in this work conforms with the definition of Kopetz [Kop08] and Baruah [BB06], who said a robust system retains its schedulability even when it operates beyond the worst-case assumptions used in its schedulability test, e.g. when jobs arrive earlier than expected, or have higher execution requirements than permitted. Clearly a system can never be fully robust and will fail when it becomes too overloaded [BB06].

Chapter 6

Multiple Time Base Task Set Extension

This chapter presents an extension for task sets which allows to define arrival patterns for groups of tasks, where tasks of a certain group are triggered by a trigger source which has an own non-synchronized time base, in comparison with the trigger source of other groups. Time bases have the effect of a collective variation of inter-arrival times for all tasks which are triggered from this time base. The extension can be applied to all task sets which allows to determine inter-arrival times, but it is mainly used for task sets with periodic inter-arrival pattern. It allows to increase the accuracy of temporal behavior models for many application cases, for example, rotation speed dependent task activation in automotive powertrain systems or task activation from a bus communication with an own (drifting) clock. Furthermore, a probabilistic model of variable execution time for non-preemptive task sections is introduced, which is closer to real task sets than worst-case execution time models. For global scheduling algorithms, non-preemptive scheduling of task sections has the benefit of reduced response-times for low priority tasks and simultaneously limited overhead of migrations in comparison with preemptive scheduling [YBB10].

6.1 Multiple Time Base Extension

Section 2.3.1 presented existing task demand models. In general, task activations are assumed to be independent of other tasks. This conforms with the recurrent demand model and the arrival curve model. The hierarchical demand model considers precedence constraints of task activations.

In contrast to these models, in many embedded systems, e.g. in the automotive powertrain domain, tasks have activation dependencies that don't depend on execution of other tasks, but on a common trigger source which has a time base of its own. For example, in a typical automotive engine control system two main sources of task activation

exist. The first source is a periodic trigger, which activates tasks with different constant inter-arrival times. The other source is the crank shaft of the engine, which activates tasks depending on the engine position.

In the following, the resulting problem of multiple time bases for schedulability analysis is outlined in an example and afterwards a model extension of task sets is proposed.

6.1.1 Problem Formulation

Phasing describes the variation of distance between activation times of periodic task in different time bases. Assuming the inter-arrival time of a periodic task is a multiple of the inter-arrival time of another periodic task, which is very often the case in practical system. Then, there is a bounded number of distances, representing the time between the activation of the first task and the activation of the n-successive activations of the other task, until the next activation of the first task occurs. These time-intervals can be used to determine the number and especially the point in time of preemptions, in the case of preemptive or cooperative scheduling and different priorities of both tasks.

Assuming another example, two tasks having the same periodic parameters as in the previous example, i.e. inter-arrival time and offset, but the periodic parameters are defined in different time bases and the time bases are not synchronized, meaning the time bases can have a offset in time or a difference in progress of time[1], however both in a defined range. Then the bounded number of distances of the previous example change to a bounded number of distance-intervals in the case of a variable offset between both time bases. Additionally, the bounded number of distances of the previous example change to a bounded number of preemptions with a defined distance-interval between successive activations of the preempting task, resulting by the difference in progress of time.

The problem of multiple time bases for task response-time evaluation will be introduced by an example. Assume, there is a singlecore system with four tasks. All tasks are triggered in a periodic manner, however by two trigger sources. The task parameters offset o_i, inter-arrival time p_i, and execution time e_i for a task $T_i = (o_i, p_i, e_i)$ are shown in Figure 6.1 for the task $T_1 = (0, 4, 1)$, $T_2 = (2, 4, 1)$, $T_3 = (4, 8, 3)$, $T_1 = (0, 24, 2)$. The task priority is assigned in a rate-monotonic manner, therefore the following prioritization $T_1 \succ T_2 \succ T_3 \succ T_4$ is applied, whereas $T_a \succ T_b$ equates T_a has a higher priority than T_b.

All tasks execute on the same processor, however task T_1 and T_2 are activated from time base b^1, whereas task T_3 and T_4 are activated from time base b^2. Reasoned to the allocation of periodic tasks to multiple time bases, which can be interpreted as non-synchronized clocks, the phasing of periodic task activations has to be considered.

[1] Using the term clock from signal processing theory for the term time base, then in analogy the offset represents the phase shift and the progress of time represents the frequency deviation between two time bases.

Due to the phasing effect, the activation of periodic tasks of the same time base is constant, i.e. between T_1 and T_2 and between T_3 and T_4, but the phasing of the activation of periodic tasks of different time bases is variable, i.e. between T_1 and T_3 or T_4, T_2 and T_3 or T_4, and the inversion of the previous.

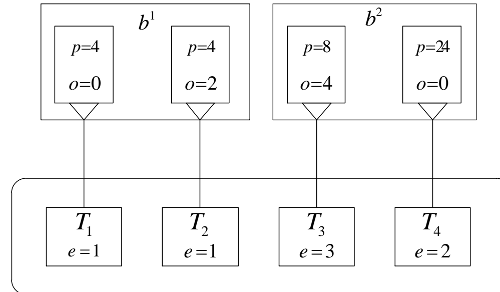

Figure 6.1: Example of task activations from different time bases. Tasks are activated in a periodic manner with an inter-arrival time p and an offset o. Task T_1 and T_2 are activated from time base b^1 and task T_3 and T_4 are activated from time base b^2. All task instances execute on the same processor.

For such kind of task activation, up to now there is no appropriate inter-arrival model. Tindell [Tin92] proposed a periodic task set model with offsets which describes a similar case. In this demand model, a system contains a fixed number of transactions (comparable to time bases) and tasks are assigned to exactly one transaction. A transaction is defined by a period, which equates the minimal distance between two sporadically or periodically arriving transactions. Furthermore, each task has an offset, which defines the task activation, relatively to the start of a transaction. Since the activations from different transactions can have any phasing, this task set model is similar to the stated problem. However, since each task is only activated once in a transaction, all tasks of a transaction share the same period. This restriction makes the arrival model from Tindell not applicable for the example in Figure 6.1.

Analyzing the response time of task T_4 in Figure 6.1 with a sporadic task set model [Mok83], assumes for the worst-case scenario that the first activation of all task arrives at the same time (see Figure 6.2 A,) according to the critical instant theorem. This produces a response time of $RT_i = 16$. Analyzing such systems with a periodic task set model with offsets [Tin92] produces a worst-case response time of $RT_i = 10$. However, the sporadic assumption is to pessimistic, because task offsets are neglected, and the periodic assumption is to optimistic, because it assumes that the phasing between all tasks is constant. Only, when the constant phasing between task T_1 and T_2 and between T_3 and T_4 and the variable phasing between both groups is respected, the correct worst-case response time of $RT_i = 11$ returns. In the following, a task set extension is proposed, which is able to model time bases and therefore increase the accuracy of response time analysis approaches of such systems.

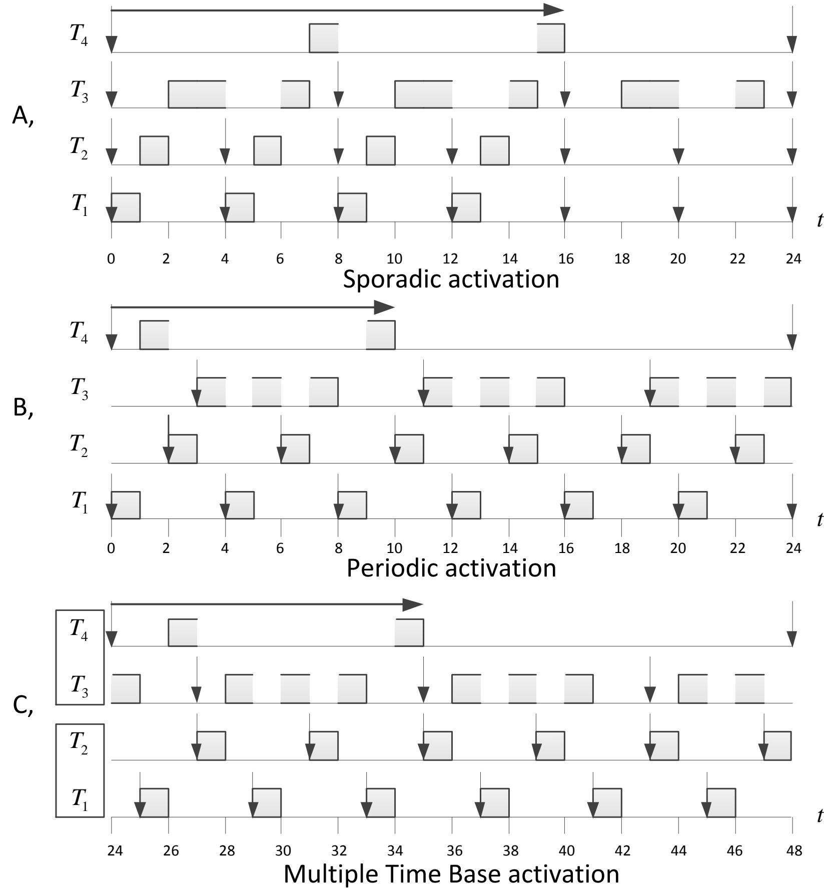

Figure 6.2: Schedule sequence of worst-case response time of task T_4 for different assumed task set models. The sporadic task set model (A) assumes all tasks to be activated simultaneously at critical instant, which results in an overestimated worst-case response time $R_4 = 16$ for task T_4. The periodic task set model with offsets (B) results in an underestimated worst-case response time $R_4 = 10$. Only the consideration of time bases, which defines that the first transactions T_1 and T_2 and the second transaction T_3 and T_4 are able to be shifted against each other, gives the correct worst-case response-time of $R_4 = 11$.

6.1.2 Extension

This section introduces the *Multiple Time Base* (MTB) extension. In MTB task sets, tasks refer to a time base of the system. A time base models a task activation source which triggers tasks with any activation pattern. However, the activation source is not synchronized with the global time. Therefore, a difference in the frequency of the time base and the global time transforms e.g. periodic inter-arrival times of the time base in variable inter-arrival times in the global time. The time base models these differences and allows to transform any activation pattern of the time base into the resulting activation pattern in global time.

Additionally, all periodic tasks which belong to the same time base have a defined phasing in their activation and tasks of different time bases have an undefined phasing in their activation. This fact is important in regard of schedulability examination, because for the worst-case response time estimation task activation constellations have to be considered (see critical instant theorem at singlecore scheduling, Section 4.3).

A system contains w time bases b^v $(v = 1, ..., w)$ and a task has a reference to exactly one time base b^v.

Definition 6.1
A time base b^v is defined by

$$b^v = f^v(t^v).$$

The frequency multiplier $f^v(t^v)$ defines the gradient between the time t^v of the time base b^v and the unique global time t at time t^v.

$$f^v(t^v) = \left.\frac{dt}{dt^v}\right|_{t^v} \tag{6.1}$$

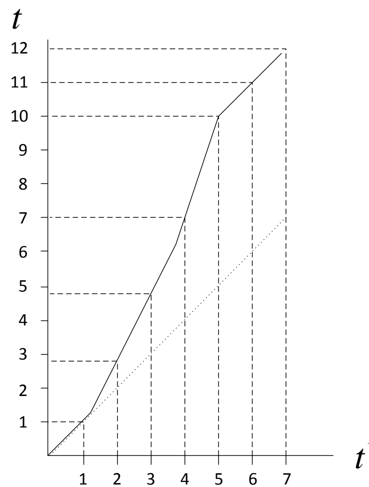

Figure 6.3: Relation between time t^v of time base b^v and global time t.

Figure 6.3 shows for an example the progress of both times in a diagram. The x-axis represents the time t^v and the y-axis represents the time t. The dotted line shows the case when both times would progress with the same speed. The continuous line shows the graphical transformation from time t^v to global time t. It can be seen, that a constant distance in time base b^v turns into a variable distance when this time is transformed to

global time.

The description of the dependency between time t^v and time t by the differential coefficient in Equation 6.1 has the following reason, illustrated by an example in Figure 6.4.

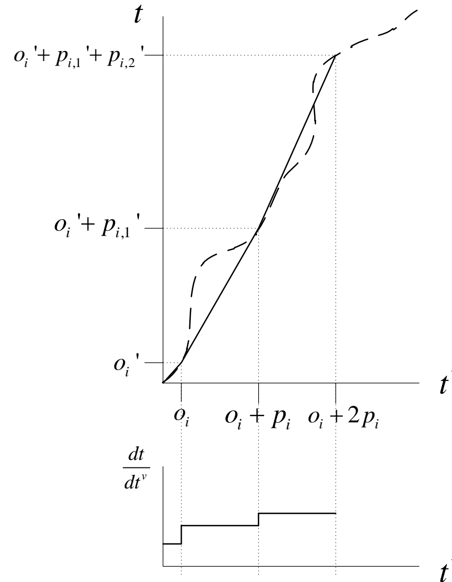

Figure 6.4: Global time t as a function of t^v (upper diagram) and derivative of t over t^v.

The time base is used to model task activations, which are discrete events in the time. Describing the complete function $t^v = f(t)$ is not necessary, which can be easily seen in the figure. Only when an activation occurs, the corresponding global time has to be determined. Therefore, it is possible to approximate the function $t^v = f(t)$ between two successive activation points by a straight line. The differential coefficient of this sequence of straight lines is a step function, as it is shown in the lower diagram of Figure 6.4. This step function allows to model time base variations by a set of frequency multipliers and time values where multiplier change.

A time t_i^v in time base b^v can be transformed in the related time t_i in global time by equation 6.2.

$$t_i = \int_0^{t_i^v} f^v(t^v)dt^v \tag{6.2}$$

The integration of the frequency multipliers $f^v(t^v)$ in the range $[0, t_i^v]$ results in global time t_i.

However, a restriction of the minimal value of the frequency multiplier is necessary. When the frequency multiplier would have values smaller zero, the time t^v moves backwards, therefore this is a tight restriction. However, also for frequency multiplier values equal to zero, tasks would have an inter-arrival time of 0 which results in an unbounded task weight $wt(T_i) = \frac{x}{0} = \infty$. Therefore, the frequency multiplier is limited as follows.

Definition 6.2
For the time base multiplier, the following restrictions exist:

$$f^v(t^v) \in \mathbb{R}_{\geq 0} \;\; \forall \; t^v, v$$

This limitation is only a formal restriction, because the inter-arrival time of a time base can be normed by the frequency multiplier. Therefore, inter-arrival times are specified in a way that they have the minimal inter-arrival time at a frequency multiplier $f^v(t^v) = 1$. However, this definition is beneficial for analysis purposes because p_i^v in time t^v is the minimal inter-arrival time p_i in global time. Otherwise, the frequency multiplier would have to be considered for the determination of the minimal inter-arrival time, which is not necessary when the range of frequency multiplier is defined in this way.

Now it is shown how an activation pattern of a periodic task can be transformed from t^v to t. As task inter-arrival time p_i and task offset o_i of task T_i are defined in the time t_v of the time base b^v, the task inter-arrival time $p'_{i,j}$ of the jth activation of the ith task and the task offset o'_i in global time t can be calculated by Equation 6.3 and 6.4 and the preceding task activation $t_{i,j-1}.A$.

$$p'_{i,j} = \int\limits_{t_{i,j-1}.A}^{t_{i,j-1}.A+p_i} f^v(t^v)dt^v \tag{6.3}$$

$$o'_i(t) = \int\limits_{0}^{o_i} f^v(t^v)dt^v \tag{6.4}$$

The MTB extension can be applied to all introduced demand models of Section 2.3.1, because all introduced demand models allow to determine the inter-arrival time p_i of a task T_i. The recurrent demand model directly defines the parameter p for the inter-arrival time. The arrival curve model defines only the inter-arrival time as a function of any time interval. However, Kuenzli and Thiele introduced a method to generate event traces from an arrival curve [Kue06]. The event trace generator uses an ON/OFF

traffic source, comparable to the approach introduced in [Bar98]. Since this event trace includes absolute task activation times, task inter-arrival times can be derived and MTB extension can be applied.

The hierarchical demand model requires that the inter-arrival time of the first task of a task chain origins from a recurrent demand model or an arrival curve model. Therefore, the inter-arrival time can be transformed analogously. All other tasks in the task chain are indirectly affected by time base variations, because the execution of the time base related task delays which results in a delayed activation of the other tasks in the chain.

6.2 Probabilistic Execution Time Model

There are two cases, where the mentioned BCET and WCET models for description of the execution time are not sufficient. The first case is when systems are non-sustainable, e.g. at global scheduling algorithms with dynamic priorities [LH94]. Then, the WCET doesn't necessarily cause the maximal response time and the BCET doesn't necessarily cause the minimal response time. Therefore, all values of the execution time between worst-case and best-case can potentially lead to maximal schedule length.

The second case is when the probability of achieving a certain response time should be determined. This can be used for example for reliability considerations, e.g. to determine rates of deadline misses.

Then, representative scenarios have to be chosen and statistic analysis or measurement methods of execution time determination have to be applied for these scenarios. In general, multiple scenarios are measured in order to construct the distribution function.

Execution time variation results mainly from different code branches or the different cycles of loops. Since only a limited number of branches exist and the duration for a loop is often constant, also execution time variations have a limited number of possible values.

However, cache misses and pipeline architectures also result in execution time variations. These variations have a smaller range and they are in general independent of the code branch. Therefore, the number of possible execution times increases. In a simplified assumption, a generalization of possible execution time values to all possible values with a resolution of processor instruction processing speed can be assumed.

The probabilistic execution time (PET) is described by a probability function $P() : \mathbb{R} \to [0,1]$. The instance of an execution time $e_{i,j}$ of an instance of task T_i is randomly generated by $P_i(x)$, whereas $P_i(x)$ describes the probability p of $x = e_i = X_{i,j}$ for an instance value $e_{i,j}$. The value $X_{i,j}$ is part of $\{X_i\}$, representing the amount of all possible instance values for e_i. In the following, possible probability functions for the description of execution time are discussed.

6.2.1 Discrete Probability Function

When measurements can be performed, the *discrete distribution* of execution time values can be determined. The range of possible execution times is discretized into a number of bins with a fixed size. The discrete pobability function $P_i^D(x, e_{\min}, e_\delta, \{p_n\})$ (Equation 6.5) of a task T_i has the parameters lower bound of execution time $e_{\min}$, a bin size e_δ, and a list of probabilities $\{p_n\}$. The probability p_n for an value x follows following function:

$$P_i^D(x, e_{\min}, e_\delta, \{p_n\}) = p_n : (e_{\min} + (n \cdot e_\delta) \leq x < (e_{\min} + (n+1) \cdot e_\delta) \tag{6.5}$$

The nth probability value p_n ($n \in \mathbb{N}_0$) indicates the probability of a value x lying in the bin $[e_{\min} + n \cdot e_\delta, e_{\min} + (n+1) \cdot e_\delta[$.

6.2.2 Weibull Pobability Function

When it is not possible to extract a trace of execution times, a discrete distribution can not be used. However, when statistic estimators like average, maximum, and minimum execution time are available, a pobability function can be determined which approximates the statistical estimators. The *Weibull Probability*function $wb(x, \lambda, \kappa)$ is appropriate for this purpose because it allows to change the mean and the probability of a certain value by a shape and scale parameter. Therefore, it is possible to describe the execution time variation with modified parameter Weibull probability function $P_i^{WB}(x, e_{\min}, e_{\text{avg}}, e_{\max}, p_{\max})$ by the mentioned statistical estimators minimum $e_{\min}$, average e_{avg}, and maximum $e_{\max}$ and an additional parameter which defines the probability of the maximum $p_{\max}$.

The probability of the maximum randomly drawn variable of Weibull density function $wb(x, \lambda, \kappa)$ is necessary, because $wb(x, \lambda, \kappa)$ is 1 for $\lim_{x \to \infty}$. Certainly an upper bound of the generated values is necessary when using a probability function for the description of execution time variations.

Therefore, an approximation is applied which defines the probability of the maximum value and determines Weibull parameters for the Weibull probability function. Afterwards, when values are generated from this Weibull probability function, values exceeding the maximum value are removed from the list of generated values.

The Weibull density function $WB(x, \lambda, \kappa)$ [RK08] has the three parameters: probability x, shape parameter κ, and scale parameter λ.

$$WB(x, \lambda, \kappa) = \begin{cases} \frac{\kappa}{\lambda} \left(\frac{x}{\lambda}\right)^{(\kappa-1)} e^{-\left(\frac{x}{\lambda}\right)^{\kappa}} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{6.6}$$

Now, an approach to determine shape and scale parameter from statistical estimators of maximal execution time $e_{\max}$, minimal execution time $e_{\min}$, and average execution time e_{avg} is introduced. The approach approximates shape and scale parameter by an iterative procedure, and uses the average value of the Weibull density function and the cumulative distribution function for the Weibull distribution.

The average value of the Weibull density function [LNR99] can be derived by Equation 6.7.

$$x_{\text{avg}}(\kappa, \lambda) = \lambda \cdot \Gamma\left(1 + \frac{1}{\kappa}\right) \tag{6.7}$$

$\Gamma(z)$ denotes the gamma function[2] which can be determined efficiently by Taylor series expansion.

The Weibull distribution function [Gum04] is shown in Equation 6.8. It describes the normalized probability measure and is the integral of the Weibull density function.

$$wb(x, \lambda, \kappa) = 1 - e^{-\left(\frac{x}{\lambda}\right)^{\kappa}} \tag{6.8}$$

When Equation 6.7 is rearranged the following equation is derived:

$$\lambda = \frac{x_{\text{avg}}}{\Gamma\left(1 + \frac{1}{\kappa}\right)}. \tag{6.9}$$

The probability p_{limit} that a drawn value exceeds a certain value x_{limit} calculates by Equation 6.10.

$$p_{\text{limit}} = 1 - wb(x) = e^{-\left(\frac{x_{\text{limit}}}{\lambda}\right)^{\kappa}} \tag{6.10}$$

By a combination of these formulas, λ and κ parameters can be derived from the distribution data: average value x_{avg}, probability at limit p_{limit}, and a limit value x_{limit}.

Replacing λ in Equation 6.10 by Equation 6.9 results in the following equation:

$$p_{\text{limit}} = \exp\left(-\frac{x_{\text{limit}} \cdot \Gamma\left(1 + \frac{1}{\kappa}\right)}{x_{\text{avg}}}\right)^{\kappa} \tag{6.11}$$

Since Equation 6.11 can not be solved analytically, a numerical approximation approach has to be applied. The approach, comparable to the nested interval approach [KW95], starts with the smallest value of $\kappa = 1$ and then adds 1 to κ with each step, until Equation 6.11 delivers a value which is higher than the defined p_{limit}. Then $\frac{1}{2^{(n-1)}}$ is subtracted from κ, where n is the number of iterations which produce a p_{limit} which is higher than the defined value. In the next step the value $\frac{1}{2^{(n)}}$ is added until p_{limit} is higher than the defined value and the approach subtracts $\frac{1}{2^{(n-1)}}$ from κ with $n = 2$. This procedure repeats until the deviation between the determined and the defined value of p_{limit} is smaller than ϵ.

In the following, the application of the mentioned execution time estimators for the calculation of the Weibull distribution function parameters is shown.

[2] $\Gamma(z) = \int_{0}^{\infty} t^{z-1} e^{-t} dt$

The first modification concerns the minimum value of the Weibull density function which is 0. Since the minimum execution time of a task is in general higher than 0, the Weibull density function has to be shifted. Therefore, average and maximal execution time are shifted by the minimum execution time.

$$e'_{\text{avg}} = e_{\text{avg}} - e_{\text{min}} \tag{6.12}$$

$$e'_{\text{max}} = e_{\text{max}} - e_{\text{min}} \tag{6.13}$$

Both values, combined with the probability p_{max} of values exceeding e'_{max}, are inserted in equation 6.11:

$$p_{\text{max}} = \exp\left(-\frac{e'_{\text{max}} \cdot \Gamma\left(1+\frac{1}{\kappa}\right)}{e'_{\text{avg}}}\right)^{\kappa} \tag{6.14}$$

When the parameter κ is calculated, λ can be determined by Equation 6.7. Then, the Weibull probability function of the execution time for statistical estimators can be derived by adding e_{min} to Equation 6.6.

$$WB(x,\lambda,\kappa,e_{\text{min}}) = \begin{cases} \frac{\kappa}{\lambda}\left(\frac{x-e_{\text{min}}}{\lambda}\right)^{(\kappa-1)} e^{-\left(\frac{x-e_{\text{min}}}{\lambda}\right)^{\kappa}} & x \geq e_{\text{min}} \\ 0 & x < e_{\text{min}} \end{cases} \tag{6.15}$$

$$wb(x,\lambda,\kappa,e_{\text{min}}) = 1 - e^{-\left(\frac{x-e_{\text{min}}}{\lambda}\right)^{\kappa}} \tag{6.16}$$

Finally, the problem of the different shapes of Weibull density function is considered, when the mean value e_{avg} is higher or lower than the average value of minimum e_{min} and maximum e_{max}, in the following mentioned as *average of bounds* b_{avg}. When this problem is not obviated, the shape of the Weibull density function differs as shown in Figure 6.5.

When the average value increase beyond b_{avg} the Weibull density function approximates (but is not equal) to a Dirac delta function, which results in a high density at the average value and a low density over the other values. In Figure 6.5 the difference can be seen: although both curves have an average value differing 30 from minimum and maximum bound respectively, the density function with $e_{\text{avg}} = 80$ has fewer values near the average value than the density function with $e_{\text{avg}} = 270$. For higher values of e_{avg} the Weibull density function has nearly all values at e_{avg} which doesn't confirm to the required shape of the distribution function, delivering values over the complete range between e_{min} and e_{max}.

Following mirror approach can be applied, to solve this problem:

1. When $e_{\text{avg}} > (\frac{e_{\text{max}}-e_{\text{min}}}{2})$,
2. set $e'_{\text{avg}} = e_{\text{max}} - e_{\text{avg}}$,
3. calculate Weibull parameters κ and λ by Equation 6.14, and

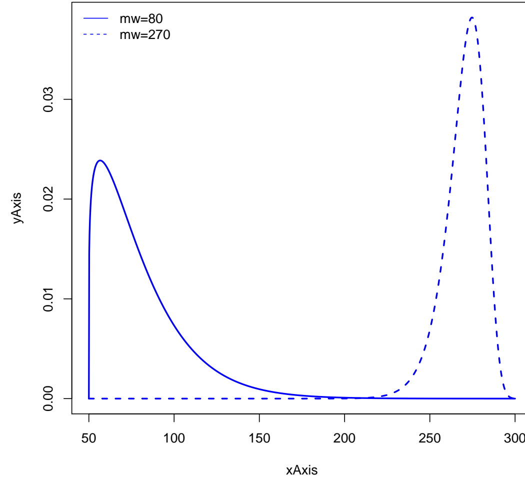

Figure 6.5: Example for different shapes of Weibull density function for equal minimal and maximal value and different average values.

4. determine mirrored values according to:

$$WB(x, \lambda, \kappa, e_{\text{min}}, e_{\text{max}}) = e_{\text{max}} - \begin{cases} \frac{\kappa}{\lambda}\left(\frac{x-e_{\text{min}}}{\lambda}\right)^{(\kappa-1)} e^{-\left(\frac{x-e_{\text{min}}}{\lambda}\right)^{\kappa}} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{6.17}$$

When this approach is applied, a mirrored Weibull density function for equal absolute distances of e_{avg} from average of bounds can be determined, as shown in Figure 6.6.

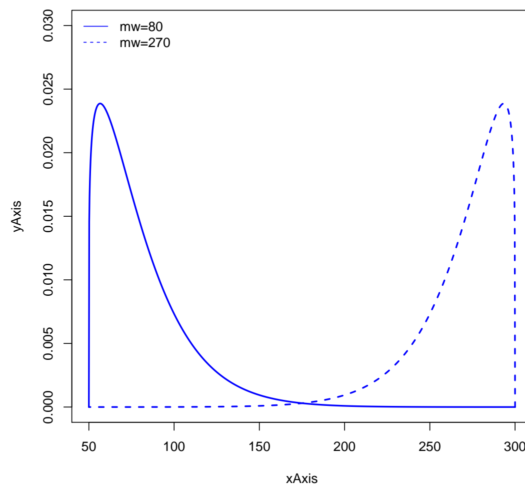

Figure 6.6: Example for corrected shapes of Weibull density function for equal minimal and maximal value and different average values.

Chapter 7

Global Multicore Scheduling

This chapter focuses on global scheduling algorithms for complex task sets with time base triggered arrival pattern and variable execution time of non-preemptive task sections, on multicore processors. For these task sets, especially used in automotive powertrain systems, a non-work-conserving [DMM$^+$10] and a work-conserving [DMMN10] cooperative scheduling algorithm is presented.

7.1 Partly Pfair Approach

This section introduces extensions of Pfair scheduling which consider the following properties of Pfair: (*i*) Pfair is a preemptive scheduling algorithm, (*ii*) Pfair requires that scheduling events and task activation enter at time quanta, (*iii*) Pfair requires tasks to have constant execution times, (*iv*) Pfair requires implicit task deadlines.

7.1.1 Drawback of Pfair scheduling

One widely discussed drawback of Pfair scheduling is the high number of context switches [ZMM03, BCL09, CG06], which occur due to a high number of preemptions (*i*). At any point during task execution a preemption can occur, causing a context switch. Several actions are required at context switching, dependent on processor architecture and software application. Saving process register, stack pointer, and further registers of the current task is required for every software architecture. The duration for these actions can be assumed to be independent of the preemption point. A further action is the recovery of missing cache data for the next running task, where the size of data depends on task data and the current position in task execution. In Pfair scheduled systems, scheduling events occur at each time quantum, defined by the quantum size Q. It seems as if task preemption points are predictable, because the time between two schedule events is constant. However, different code paths and different task input signals result in different program paths. Therefore, the size of required data (and therefore also the possibly missing cache

data) at preemption point is not predictable. Without a prediction of preemption points, there is no way to improve the cache related context changing overhead by reducing required data after a task preemption.

A further disadvantage of Pfair scheduling is that task activation and scheduling events only occur at time quanta (*ii*). Also when Baruah et al. did not mention how to choose the quantum size, certainly the time quanta size has to be a multiple of the instruction processing speed resolution. For short time quanta the high number of scheduling events results in increased context switching overhead, as mentioned above. In contrast, a high quantum size is disadvantageous when using sporadic task sets with original Pfair scheduling, because the time until an activated task becomes scheduled can be as large as Q time units. For tasks with an initial laxity[1] lower than Q this could lead to a deadline miss.

The most critical limitation of Pfair scheduling is a constant task execution time (*iii*), which can not be held in practical systems. Devi and Anderson [DA05] analyzed the effect of this limitation. They assumed that a task quantum can be smaller than one quantum Q and the schedule decision starts after this fraction of a task quantum has been finished. Under this assumption, they proved that task deadlines are violated by at most one quantum Q, therefore Pfair can be applied for soft real-time task sets.

A further limitation of Pfair is the demand for implicit deadlines (*iv*). Task set models with explicit deadlines allow to define more realistic temporal requirements, for example in an automotive brake system the minimum time between braking events may be considerably larger than the required braking-reaction time.

7.1.2 Partly Pfair

In order to overcome the mentioned disadvantages $i-iv$ of Pfair scheduling in the previous section, this work proposes the scheduling model Partly Proportional fair (Partly-Pfair) [DMM$^+$10]. The term *partly* implies that this algorithm does not fulfill the proportional fair bounds (Equation 3.7) but applies a transformation of MTB task set parameters in order to calculate the scheduling policies. Furthermore, the occurrence of scheduling events is modified, which changes the preemptive scheduling of tasks to cooperative scheduling of non-preemptive task sections.

Partly-Pfair is the adaptation of Pfair to MTB task sets, introduced in Section 6.1. In the following, the transformation from MTB task set parameters to Pfair conforming parameters is shown.

[1] difference between deadline and execution time

For MTB task sets, the minimal task inter-arrival time is characterized by $T_i.p$ in time base time. For MTB task sets, arbitrary deadlines are allowed. It is assumed that a job of a task has to be finished at task deadline or at least when the next task activation occurs. $FI(T_i)$ denotes the finalization time, meaning the minimum of the absolute task deadline and the next task activation.

$$FI(T_i) = min(T_i.d, T_i.p) \tag{7.1}$$

At MTB task sets task execution time is split into task sections. For Partly-Pfair, the maximal task section execution time is defined as follows, whereas Q equates the quantization:

$$T_i^k.e \leq Q \; \forall i, k. \tag{7.2}$$

It is assumed that execution time can be variable during runtime. Additionally the exact execution time can not be known prior to execution[2]. Therefore, this work uses the maximal task section execution time $T_i^k.e = Q \; \forall T_i^k \in T_i$ for the calculation of Partly-Pfair scheduling policies. Together with the definition of the finalization time $FI(T_i)$, the modified task weight $w'(T_i)$ calculates by Equation 7.3.

$$\text{wt}'(T_i) = \frac{|\{T_i^k\}|}{\left\lfloor \frac{FI(T_i)}{Q} \right\rfloor} \tag{7.3}$$

The modified weight is a transformation of MTB task set characteristics to Pfair task set restrictions. Since task deadlines $T_i.d$ and minimal inter-arrival times $T_i.p$ are allowed to have arbitrary values, using the exact value of the minimum of both $FI(T_i)$ could result in a task weight which can not be allocated to an integer number of task sections (see Equation 3.9 and 3.10). Therefore, the floor function $\lfloor x \rfloor$ is applied, which returns an integer smaller or equal to x. In order to get a quantized value, the floor function is applied to the quotient of $FI(T_i)$ and Q and the result is multiplied with Q. The floor function guarantees that fluid schedule assigns task execution time to processors in a way that a task job finishes before deadline or next task activation. Furthermore, the execution time $T_i.e$ derives from the number of task sections $|\{T_i^k\}|$ multiplied with Q, in order to achieve a correct weight. The modified weight wt' is used to calculate the modified pseudo-release time $r'(T_i^k)$ and the modified pseudo-deadline $d'(T_i^k)$:

$$r'(T_i^k) = \left\lfloor \frac{k-1}{\text{wt}'(T_i)} \right\rfloor \tag{7.4}$$

$$d'(T_i^k) = \left\lceil \frac{k}{\text{wt}'(T_i)} \right\rceil \tag{7.5}$$

[2] Unless the algorithm has forecasting mechanisms for execution times

All other policies used for the Pfair algorithms can be applied in the same way as with Pfair task set characteristics.

The overlapping bit $b'(T_i^k)$ for Partly-Pfair Scheduling is determined by replacing the task weight $\mathrm{wt}(T_i)$ in Equation 3.12 with the modified task weight $\mathrm{wt}'(T_i)$, as shown in Equation 7.6.

$$b'(T_i^k) = \left\lfloor \frac{k}{\mathrm{wt}'(T_i)} \right\rfloor - \left\lceil \frac{k}{\mathrm{wt}'(T_i)} \right\rceil \tag{7.6}$$

The group deadline $D'(T_i^k)$ derives from replacing $\mathrm{wt}(T_i)$ with $\mathrm{wt}'(T_i)$ in Equation 3.13 and 3.14, as shown in Equation 7.7 and 7.8.

$$D'(T_i^k) = \left\lceil \frac{\left\lceil \left\lceil \frac{l}{\mathrm{wt}'(T_i)} \right\rceil \cdot (1 - \mathrm{wt}'(T_i)) \right\rceil}{1 - \mathrm{wt}'(T_i)} \right\rceil \text{ if } 0.5 \geq \mathrm{wt}'(T_i) < 1 \tag{7.7}$$

$$D'(T_i^k) = 0 \text{ if } 0 < \mathrm{wt}'(T_i) < 0.5 \tag{7.8}$$

To overcome the mentioned restriction to preemptive scheduling (i), Partly-Pfair schedules task sections in a non-preemptable manner. Non-preemptable scheduling has been shown in several work as efficient method to limit context switching overheads [BW97, LLL+98, GMM99, BLV09, YBB10, BBM+10]. Pfair allows to preempt tasks at each time quantum. Since task activation times can occur at any time for MTB task sets, the limitation to synchronized periodic task activation at time quanta is removed:

Partly-Pfair allows to preempt tasks only at task section ends. Whenever a task section has finished the scheduler assigns a ready task sections to the free core.

Furthermore, this also allows sporadic task activation where $T_i.p$ represents the minimal inter-arrival time of a sporadic task. A similar extension, called intra-sporadic release [AS00b], allows task quanta to be released late. An example of such an application is a multimedia system in which packets may sometimes arrive early or late [AS00b].

The definition of variable task section execution times which are smaller than Q is a generalization of the desynchronized variable quanta (DVQ) model, proposed by Devi and Anderson [DA05]. This allows a task to have an execution time, where the last task quantum has an execution requirement less or equal Q.

Example *Partly-Pfair-PD*^{2}

Figure 7.1 shows an example of a *Partly-Pfair-PD*^{2} schedule on a singlecore processor, using slightly modified task properties, compared with the task properties of the *Pfair-PD*^{2} example in Figure 3.5. Tasks T_1 and T_2 have task sections with an execution time $T_1^k.e, T_2^k.e \leq Q \; \forall k$. Task T_1 originates from a desynchronized periodic task set with task

parameter $T_1.o = 0.5$ and $T_1.p = 10$.
At timestamp 0 only task section T_2^1 is pseudo-activated and executes.
Between timestamp 0 and 1, after task section T_2^1 has finished (Figure 7.1, black star), T_1^1 is activated and task section T_2^2 has not reached its pseudo-activation. Therefore, task section T_1^1 is executed.
A scheduler call occurs each time when a task section has finished. When there is no pseudo-activated task section available for execution, an additional scheduler call (Figure 7.1, gray star) is set at the next pseudo-activation time, e.g. at time 3.

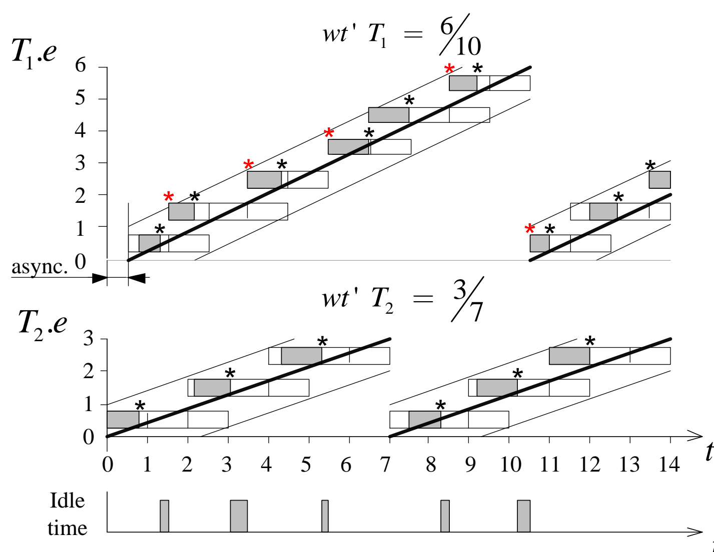

Figure 7.1: Example of a schedule of algorithm *Partly-Pfair-PD*^{2}. In the upper diagrams, the x-axis represents the time and the y-axis the processed execution time of a task. The lower diagram shows when the processor is in the idle state.

7.2 Partly Early Release Fair Extension

Partly-Early Release Fair (*P-ERfair*) [DMMN10] is the application of *ERfair* scheduling to the concept of Partly-Pfair scheduling. The transformations of MTB task set parameters to Pfair scheduling are applied in the same way at *P-ERfair*. The modified weight $wt'(T_i)$ is calculated by Equation 7.3 and the modified pseudo-deadline $d'(T_i^k)$ is calculated by Equation 7.5. However, *P-ERfair* removes the modified pseudo-activation time $r'(T_i^k)$ and allows a task sections to execute whenever the previous task section has finished. This makes *P-ERfair* a work-conserving algorithm. All other policies, e.g. overlapping bit and group deadline for PD^2, are equal to the policies of *Partly-Pfair*.

Example *P-ERfair-PD*^{2}

As an example, the *Partly-Pfair-PD*^{2} example from Figure 7.1 is shown in Figure 7.2 as schedule of *P-ERfair-PD*^{2}. At time stamp 0 only task T_2 is activated and therefore task section T_2^1 is executed. Between time stamp 10 and 11, after task section T_2^3 has finished, T^2 has finished execution and task T_1 is not activated. As there is no task ready for execution, the processor changes to the idle state and the scheduler waits until the next call occurs, which happens when task T_1 is activated the second time. It can be seen that the time when the processor is in the idle state is summarized at *P-ERfair-PD*^{2}, whereas it is distributed over the execution time at *Partly-Pfair-PD*^{2}. In the case study in section 10.4 it will be shown that this effect of *P-ERfair-PD*^{2} is beneficial, when the robustness against perturbations is considered.

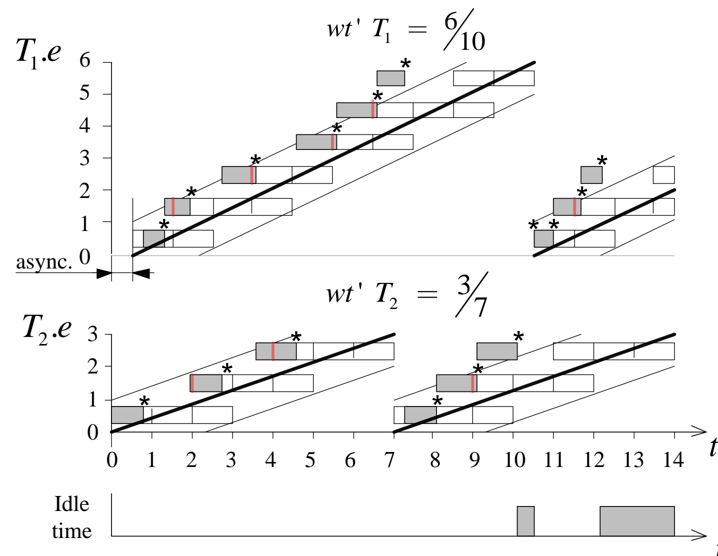

Figure 7.2: Example of a schedule of algorithm *P-ERfair-PD*^{2}. In the upper diagrams, the x-axis represents the time and the y-axis the processed execution time of a task. The lower diagram shows when the processor is in the idle state.

Chapter 8

Simulation-Based Multicore Real-Time Examination

This chapter describes a method for examination of temporal properties of real-time systems by use of discrete-event simulation [Deu08, DSM+10b, DHM+10, DM10]. The simulation model is more detailed and more practically oriented than the abstract real-time system model in Chapter 2.1 in order to study overhead effects and allow an extension of model components and properties, e.g. different task activation patterns. After an introduction to discrete-event simulation, the simulation model is explained in two parts. The first part introduces all components and their relation, mentioned as the *architecture model*. The second part explains how the components interact with each other, mentioned as the *behavioral model*. Afterwards, a metric is introduced which allows a quantitative evaluation of the deadline compliance for a complete task set and the simulative examination of this and further metrics is shown. Finally, some information of the technical implementation is given.

8.1 Discrete-event Simulation

Discrete-event simulation has two characterizing properties according to the simulation model parameters: the discretization of values of a model parameter and the discretization of the time when values change, called event.

The term *discrete* concerns the discretization of acceptable values of parameters. In comparison, continuous systems, e.g. analog electronic components, give a response on inputs which can have arbitrary granularity of parameters. Discrete systems, e.g. digital electronic components, have a defined granularity of parameters.

The term *event* concerns the time of a change in the value of a parameter. In a discrete time system, the event has a time stamp of the time of occurrence, denoted as event time. This time is also discretized and therefore events can take place only at discrete time stamps. The non-discretized approach has immediate changes of values.

The discrete-event model comes very close to an embedded system, where mainly digital components are used. The parameter granularity is defined by the data type resolution, namely the number of bits. The time granularity is defined by the processor frequency.

The model of the discrete event simulation consists of a number of components. Each component model has a state machine to describe its behavior and input and output interfaces, required to interact with other components. In interaction with other components, it is possible to perform state transitions, where the interaction is based on events. An event has an event time, which defines the time of occurrence of an event, a signal which is necessary to differ between multiple interactions between two components, and a value which gives additional information to the signal. The input interface of a component accepts only a limited number of incoming events and the output interface sends only a limited number of outgoing events.

Furthermore, this work extends the state machine model with delayed events. Standard state machines only know one type of events which are processed in order of trigger time. However, for the simulation of real-time systems it is necessary to model the elapse of time, e.g. for the execution of task. For this purpose it is distinguished between immediate events and delayed events.

Immediate events change the state of the receiving component straight after the occurrence of the event. Therefore, the time stamp of the event is equal to the time stamp of the state change. If a state transition triggers another immediate event, there is no delay between the event time of both events.

Delayed events change the state of the receiving component at a defined time in the future. Since a typical state diagram does not allow to model delayed events, in Section 8.3.1 the simulation sequencer is introduced which receives all delayed events and sends an immediate event when delay time has passed. In the meantime, the receiving component stays in the current state and is also able to receive further events. A delayed event can be modified by the component which registers the event at the simulation sequencer, e.g. when a core registers a delayed event for the finalization of a task execution but the task is suspended during execution, then the finalization time of the old task is replaced by the finalization time of the new task.

For the chronological correct triggering of all delayed events, the simulation sequencer has an event queue of all received delayed events and sorts them by their event time. When the global time is equal to an event time, the state transition of the receiving component enters at the component.

8.2 Architectural Model

The architectural model of the simulation includes four subsystems: a hardware subsystem, a software subsystem, a stimulation subsystem, and an operating system subsystem.

These models have a number of components, shown in Figure 8.1. After a short introduction of the different components, the necessity of this fragmentation is discussed in the remainder of this section and in Section 8.3.

The hardware subsystem has a number of cores and quartz oscillators, where one or multiple cores of a processor are assigned to a core cluster. The operating system subsystem models the scheduling of tasks (by a scheduler) and the scheduling of ISRs (by a dispatcher). A scheduler controls a core cluster and is able to allocate tasks, which are assigned to this scheduler, to all cores of the core cluster, as long as the scheduling model allows migration. The dispatcher decides whether the allocated task executes, when there is no waiting ISR, or which ISR executes, when multiple ISRs wait for execution. The software subsystem is extended with ISRs to model interference of available processor capacity on task scheduling, e.g. for robustness considerations. It contains the execution time description of all processes. Tasks are mapped to a scheduler, which allocates task instances to the cluster of cores, ISRs are mapped statically to cores. The stimulation subsystem describes the activation pattern of processes.

8.2.1 Stimulation Subsystem

The triggering of tasks and ISRs is modeled by a stimulation subsystem. A stimulus component has a connection to one or multiple trigger targets. At each trigger of the stimulus, the related trigger target gets a signal. The stimuli are used to model the activation of tasks and ISRs.

8.2.2 Software Subsystem

The software subsystem contains the components ISRs, tasks, and runnables. A runnable abstracts the execution demand of an entity of program code, e.g. a function call, by a number of instructions. Additionally a runnable is able to contain a functional code which is performed after execution of all instructions. The simplification of performing the functional code after all instructions have been executed is a trade off between simulation complexity and temporal accuracy. For a more detailed analysis it would be possible to sub-divide runnables in code fractions. This would allow a higher degree of accuracy, because functional code fractions would define the time granularity. Unfortunately this leads to a much higher number of simulation steps, which is not justified in this work, since the functionality is only considered at scheduler routines.

In practical systems, runnables are sub-parts of a functionality. Therefore, multiple runnables have to be executed in a defined order or a runnable has to be executed at multiple positions of the application[1]. Therefore, runnables are composed[2] in task sec-

[1]E.g. when the runnable contains a basic functionality, which is required at multiple parts of the application.

[2]This work uses composed or composition of objects to indicate that multiple other objects can have a reference on the same objects.

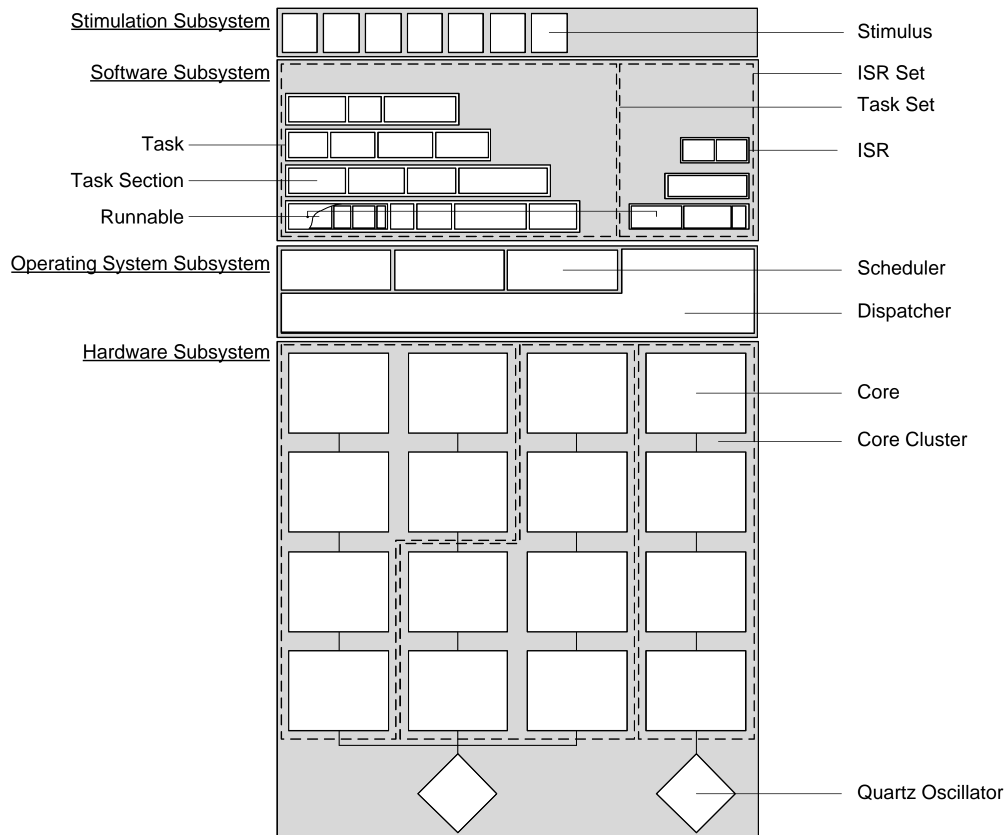

Figure 8.1: Subsystems and components of the discrete-event simulation. The real-time system is divided in the subsystems: stimulation, software, operating system, and hardware. The extension of this model in comparison to singlecore systems [AUT10] are multiple cores, which are arranged to core clusters. Each core cluster has a scheduler which allocates the tasks, managed by this scheduler, to the cores of the core cluster. Quartz oscillators are mapped to cores in order to provide the clock for the processing frequency.

tions in a sequential order. Task sections itself are aggregated[3] to tasks. The subdivision of tasks in task sections allows to build cooperative scheduling, as explained in Chapter 3.1.

In difference to the task model, the ISR model only includes one section. Nevertheless, ISR model allows to allocate a composition of multiple runnables in this section.

8.2.3 Hardware Subsystem

The hardware subsystem consists of a number of quartz oscillators and a number of cores. The quartz oscillators are connected to the cores to provide the frequency for instruction

[3]This work uses aggregated or aggregation of objects to indicate that only one object can have a reference on the same object.

processing speed. The explicit model of quartz oscillators allows to simulate the effect of instruction processing speed variation, how it is used for energy saving purpose.

8.2.4 Operating System Subsystem

The operating system is divided in two parts, one part for tasks and one part for interrupt service routines (ISRs). A scheduler prioritizes tasks and allocates them to a cluster of cores. The interrupt handler prioritizes ISRs at cores, which are statically allocated to cores.

Task have additional properties, required for scheduling decision. These properties are the task deadline, and the algorithm specific properties static priority, task group, and MTA[4] for OSEK scheduling and assumed execution time and assumed inter-arrival time for the group of Pfair and Partly-Pfair algorithms. Further scheduling policies can be derived from these values. It is important to distinguish between the real and the assumed execution time and inter-arrival time. At scheduling decision, some parameters are not known ahead like the execution time, because it depends e.g. on the executed path or on cache misses. Furthermore, the inter-arrival time can be indeterministic due to external influences. Therefore, assumptions, i.e. forecasts, of these parameters are required. These assumed parameters can either be static or variable, e.g. when using on-line forecast or interpolation mechanism.

ISRs have the priority as additional parameter. It is assumed that all ISRs are non-migrating with the only exception of the ISR of a global scheduling algorithm, which is able to allocate its instance at scheduling request to any core. Further information is introduced in Section 8.3.5.

8.3 Behavioral Model

For the purpose of the explanation of the simulation behavioral model, an own definition of state diagrams is used to describe the internal behavior and a symbolic model describes the component interaction. The state diagrams contain states which can be changed by a transition. A *regular transition* is triggered by an event, a *conditioned transition* is triggered when a certain condition is true, or an *unconditioned transition* is triggered after the operations of the previous state finished.

All components start in the initial state (INIT). If and only if a component exists for a fraction of the simulated time, it also has a final state. For example, this applies to process instances, which are initialized at the activation and destroyed at the finalization.

[4]Multiple Task Activation (MTA) limits the number of concurrently existing task instances of one task and is used in many practical systems for bounding of overhead.

The behavior of the component is described by state diagrams and the following notation:

$$A \xrightarrow{X} B$$

denotes an event X, which is sent from component A to component B at performing a state transition;

$$B \underset{X}{\longrightarrow} A$$

denotes an event X, which is received from component B by component A and probably results in a state transition.

Figure 8.2 shows an example of the graphical description of the behavior of two components A and B. Component A sends event E_1 to component B, which can be seen in the description above the state diagrams. Additionally component A sends event E_2 to component C and receives event E_3. Component B receives event E_1 from component A.

When component A is assumed to be in *STATE A* and event E_3 enters, the event E_1 is sent to component B and component A changes into *STATE B*. The entry of event E_1 cause component B to change from *STATE A* to *STATE B*.

Component A has a non-conditioned state transition *STATE B* into *STATE C* (signalized by a dot at leaving state), therefore event E_2 is triggered immediately. A transition which is conditioned on an internal property of the component is represented by a condition σ, e.g. at component B the transition from *STATE A* into *STATE B*.

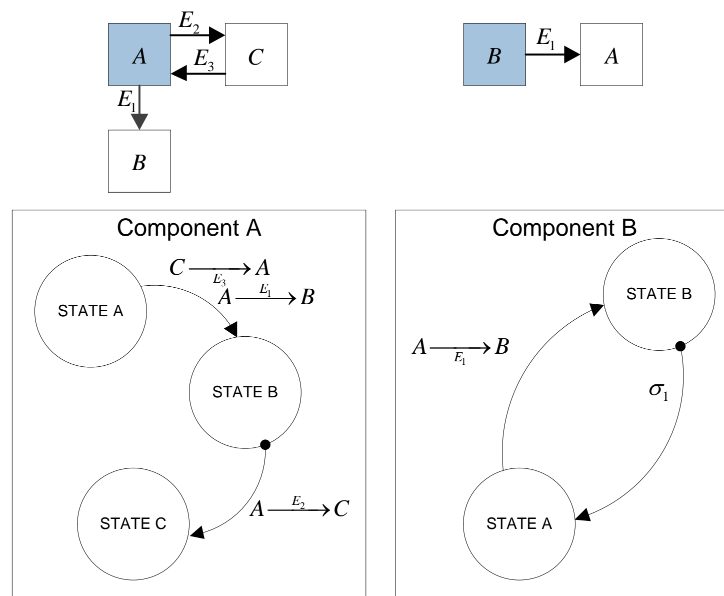

Figure 8.2: Example of the description of a behavioral model.

The following sections describe the behavior of the simulation components with this notation. Before introducing the different components of the simulation, the mechanism of the simulation sequencer is shown, which is necessary to arrange all delayed events in the correct chronological order.

8.3.1 Simulation Sequencer

As already mentioned in Section 8.1, the discrete-event simulation has immediate and delayed events. The objective of the simulation sequencer is to trigger all delayed events in the correct order. Whenever a simulation component sends a delayed event, this event is registered at the simulation sequencer. The simulation sequencer sorts all delayed events in increasing order, sets the global time to the time of the next delayed event, and triggers the event. A component is able to simultaneously register only one delayed event. When a component registers a new delayed event, although it already has registered an event, the old trigger time is replaced by the new trigger time.

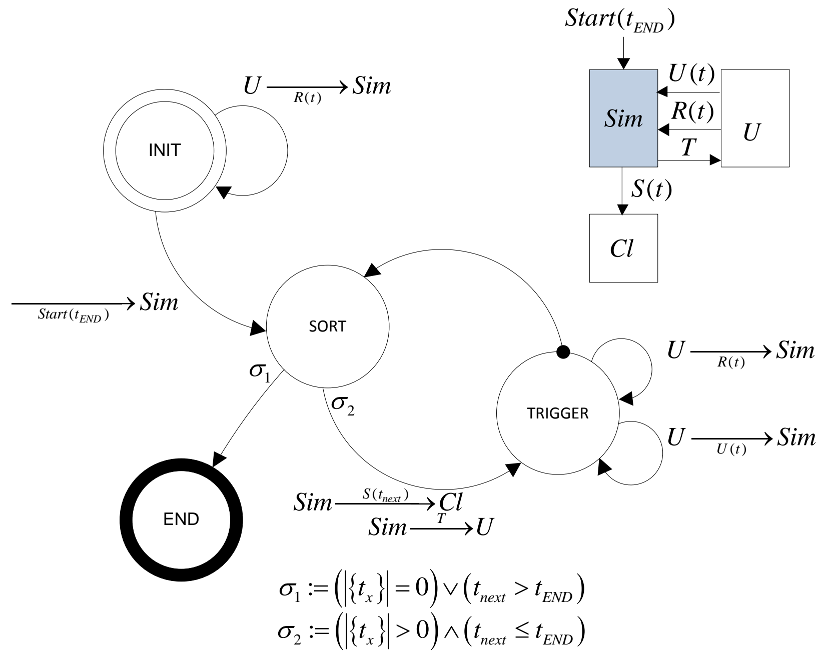

Figure 8.3: State diagram of simulation sequencer.

Figure 8.3 shows the state diagram and the interacting components of the simulation sequencer. The simulation sequencer SIM interacts with *user components* U by the events *register event* $R(t)$, *unregister event* $U(t)$, and *trigger* T, with the *clock* component Cl by the event *set time* $S(t)$, and with the simulation environment by the event *start* $Start(t_{\mathrm{END}})$, which sets the maximal simulation time t_{END}.

A *user component* registers delayed events at the simulation sequencer. The event $R(t)$ sets the event time t of the next delayed event for the registering user component. The event T executes the registered delayed event and sets global time to time t. The clock component provides the global time to all simulation components, but only the simulation sequencer is able to increase the global time by $S(t)$.

In *INIT* state, user components U are able to register a delayed event at the simulation sequencer *Sim*. For example, a stimulation component registers its next trigger time. When the simulation starts (signal $Start(t_{\text{END}})$), the simulation sequencer changes in the state *SORT*. Whenever entering *SORT*, all registered delayed events $\{t\}$ are sorted by increasing event time t. When two delayed events have the same event time, they are causally sorted according to the order of registration. After finishing sorting, it is distinguished between the condition σ_1 and σ_2. The simulation sequencer change to state *END*, when condition σ_1 holds. It means that there is no registered event or the next event time t_{next} exceeds t_{END}. The simulation sequencer changes to state *TRIGGER* when σ_2 holds. This means that there is at least one delayed event and the event time t_{next} of this event is before or equal to t_{END}. When changing to state *TRIGGER*, the sequencer changes the time at *Cl* to t_{next}. Afterwards the delayed event is triggered at the related user. In state *TRIGGER*, users U are able to register new delayed events $R(t)$ or unregister U(t) existing delayed events. A simulation user component, getting triggered by a delayed event, is able to send immediate events with the same time stamp also to other components, which itself are able to register or unregister delayed events at *Sim*. When all immediate events are triggered and all delayed events are registered at *Sim*, the simulation sequencer changes again to state *SORT*.

8.3.2 Stimulation Subsystem

The stimulation component S is connected to the simulation sequencer *Sim* and one or multiple trigger targets TT. S receives the *trigger* event T from the simulation sequencer and sends the *register* event $R(t)$ to register a delayed event. The trigger target receives the *fire* event F.

Figure 8.4 shows the behavioral model of the stimulation component. The stimulation component models the recurring firing events F through a recurring registration at *Sim*. When the stimulation component is initialized, it sends the registering event $R(t)$ for the firing time $t = t_0$ (t_0 equates the first firing time). Afterwards S changes to state *WAIT*. When the stimulation component receives the trigger event T from simulation sequencer (the global time has progressed to firing time t_0), all trigger targets receive the fire event F, and the stimulation component changes to state *TRIGGER*. Afterwards the stimulation component registers the next firing event. When t_j is denoted as the jth time of the firing event of a stimulus component, then the time of the next firing event t_{j+1} is registered at SIM by the event $R(t_{j+1})$ and the stimulation component changes to state *WAIT*. This process repeats for the complete simulated time.

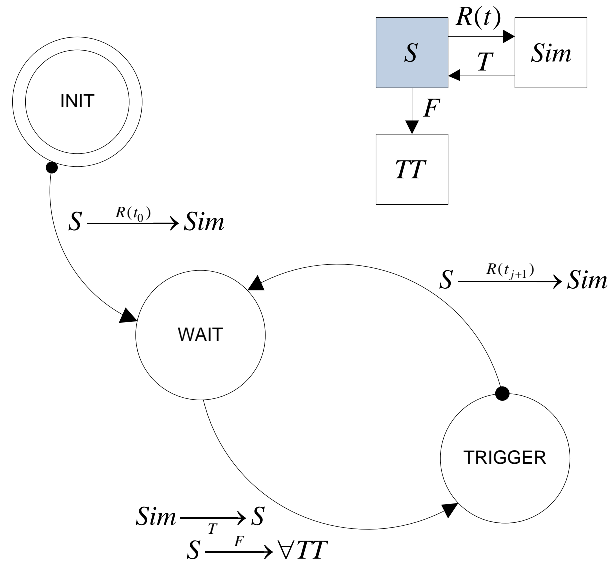

Figure 8.4: Behavioral model of stimulation component.

Since this work considers in the simulation model tasks and ISRs, the stimulation component, implementing the introduced task execution time demand models of Chapter 2.3, is extended to ISRs.

The time between t_j and t_{j+1} depends on the task execution demand model. Since all introduced models from Chapter 2.3 describe the task execution demand pattern in a way that this time can be derived, all mentioned models can be simulated. However, for this study, only the periodic model is used, with probability function extension, to model stochastic behavior, and multiple time base extension, to model task activation dependencies.

A fire event denotes the action when a stimulus triggers its target. The time, when the jth trigger event[5] enters is denoted as t_j, and the time interval between two trigger events $j-1$ and j is denoted as inter-arrival time $\delta_j = t_j - t_{j-1}$.

Periodic Model:

The simplest stimulation model is the periodic request pattern. All fire events have the same inter-arrival time δ, equal to the inter-arrival time p, except the first one, which fires after offset o has passed, measured from simulation start.

$$\delta_j^{Periodic} = \begin{cases} o & j = 1 \\ p & \forall j > 1 \end{cases} \tag{8.1}$$

[5]The index j is used for multiple components of the simulation model, whenever the instance of an object or the entry of an event is described.

Probabilistic Jitter Extension:

The inter-arrival time results from a periodic model and a probability function $p(x)$.

$$\delta_j^P = \begin{cases} o + p(x) & j = 1 \\ p + p(x) & \forall j > 1 \end{cases} \tag{8.2}$$

This work uses the probabilistic distribution functions: uniform $p_u(x)$, Weibull $p_{wb}(x)$, and a discrete probability function $p_d(x)$. A pseudo-random number generator produces numbers x according to the probabilistic distribution functions. The generated sequence of numbers has to fulfill a number of quality criteria, e.g. the spectral test to prove the independence of the generated numbers. Furthermore, the period time when a sequence of numbers repeats should be as long as possible for long simulation times. Therefore, the Mersenne twister is used, introduced in [MN98], which has a prime period of $2^{19937} - 1$ and fulfills necessary statistical test.

Multiple Time Base Extension:

Section 6.1 introduced the multiple time base (MTB) extension which allows to model a modification of task activation patterns for a subgroup of tasks in a task set. The extension can be applied to any of the introduced arrival models from section 2.3.1 by application of the following approach. The inter-arrival time which origins the arrival model X is denoted as δ^X.

The MTB extension defines a frequency multiplier $f^v(t)$ as a function of the global time t. The variation of $f^v(t)$ is modeled as a list of 2-tuples (t_i^v, f_i^v). Whenever time t_i^v enters, all stimuli which are related to the time base b^v apply the following rule to determine the new inter-arrival time $\delta_j^{X'}$ by use of the previous activation time t_{j-1}:

$$\delta_j^{X'}(t) = (t - t_{j-1}) + ((t_{j-1} + \delta_j^X) - t) * f^v(t). \tag{8.3}$$

Afterward $\delta_j^X(t)$ is set to $\delta_j^{X'}(t)$. When a modification of the time base enters until next activation, Equation 8.3 can be applied in the same way.

The unmodified inter-arrival time δ^X derives from the inter-arrival model X as explained in Section 6.1.

8.3.3 Software Subsystem

This section defines the software model of the real-time system, which is deduced from the task set model of section 2.2.1.

In many real-time systems, timing requirements, i.e. deadlines, for the separate parts of the software are distributed over a high range, e.g. in automotive powertrain systems in an interval of $[10^{-3}, 10^3]$ milliseconds [MNW10]. In order to allow tight response times, especially for the software parts with deadlines near to the lower limit, there is a multi-level scheduling of processors. The parts of the software with short timing requirements

are realized as Interrupt Service Routines (ISRs), having a reduced context-switching overhead and a limited access on time intensive operating system functions. All other parts of the software are realized as tasks. ISRs are mostly statically allocated to cores and a static priority is assigned for scheduling decisions (ISR scheduling is managed by the dispatcher) in order to minimize the delay of online scheduling decisions, whereas tasks can be scheduled by any scheduling algorithm (task scheduling), allowing a better usage of the available processing capacity. At each time of scheduling decision, a waiting ISR is preferred over any task, in order to allow tight ISR response times. This type of scheduling, which prefers decisions of one scheduler (for ISRs) over the decisions of another scheduler (for tasks), is a subgroup of hierarchical scheduling [DB05].

Since ISRs and tasks have the same stimulation and processing model but different scheduling policies, this work generalizes ISRs and tasks to a *processes model* and extends this model by a set of individual parameters for scheduling decision of ISRs and tasks. Additionally, tasks can have multiple task sections for non-preemptive scheduling of parts of the execution time (see Section 2.2.1), whereas ISRs have only one section due to full-preemptive scheduling. Generalizing ISRs and tasks to processes has the benefit of a variable assignment of software parts to the dispatcher of ISRs or the task scheduler by adding the corresponding schedule parameters. Furthermore, the stimulation model and the determination of real-time metrics (see Section 4.1.2) can be applied in the same way.

When a stimulus sends a fire event to a process, i.e. activates a process, a *process instance* is generated. The instantiation of a process is required due to a concurrent execution of multiple instances of the same processes. This enters when the inter-arrival time of a stimulus is smaller than the response time of the related process, which especially is the case in event driven activation, when multiple requests for execution of one process could exist at the same time. In uniprocessor systems, the common approach is to execute all process instances of one process in a FIFO manner, because only one process instance can be executed at the same time. The first process instance executes until termination, all other process instances of the process remain in the state activated. For the case of too many activations, there is a pragmatic approach for handling this scenario in practical systems, e.g. by the operating system OSEK/VDX [Int05]. A counter for multiple task activation (MTA) determines the number of coexistent process instances of one process. If the MTA counter exceeds its configured limit, all further trigger events for that process are ignored. In multiprocessor systems, i.e. when using a global scheduling algorithm, an alternative approach is applicable. For example, new arriving process instances can be allocated to other processors and therefore process instances are able to be executed concurrently. This is a high benefit, especially in queued processing systems where data items can be processed concurrently, however requires a new process model for execution. In uniprocessors systems, it is sufficient to have a global data section where the process

instances of one process have access[6]. Other process instances use this data section only if the previous process instance finished (justified due to FIFO execution). Therefore, the interference of process data is prevented. In multicore systems, when process instances are able to be executed concurrently, data inconsistency can occur due to interfering access of multiple process instances on a process data section. Furthermore, when a scheduling algorithm executes schedule decisions, task states and properties are required, which can also be modified.

Therefore, this work defines a process instance model, which is introduced after the process model.

Process

Figure 8.5 shows the behavioral model of the process component.

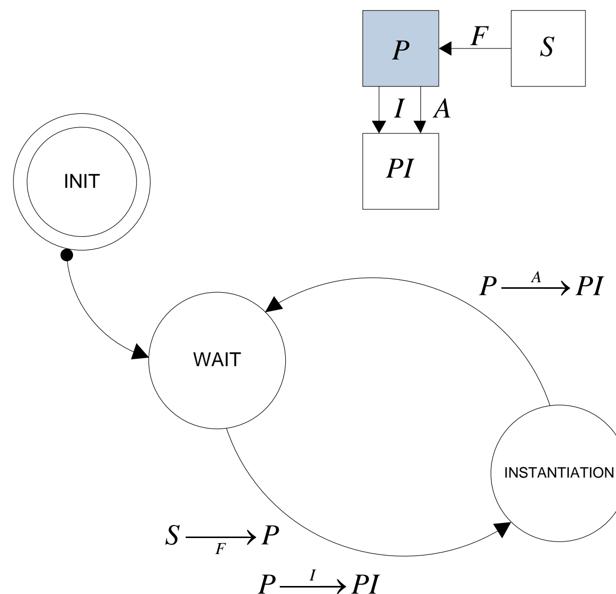

Figure 8.5: Behavioral model of process component.

A process P interacts with the stimulation component S and the process instance component PI. The stimulation component sends the fire event F and the process instance receives the events initialize I and activate A.

After initialization, a process component immediately changes to state *WAIT*, and waits until a *fire* event occurs. When this happens, a process instance is getting instanced and the process changes in state *INSTANTATION*. In this state, further related components get initialized and component parameters are calculated (see component process instance). When initialization was successful, the process sends the event activate A to

[6]Sometimes also architectures with complete shared memory are used, which is dangerous because of unauthorized access on process memory.

the process instance PI and changes into state $WAIT$.

Process Instance

The process instance PI interacts with the components process P, runnable instance RI, core C, and operating system OS, whereby the operating system can be either a scheduler or an interrupt management unit.

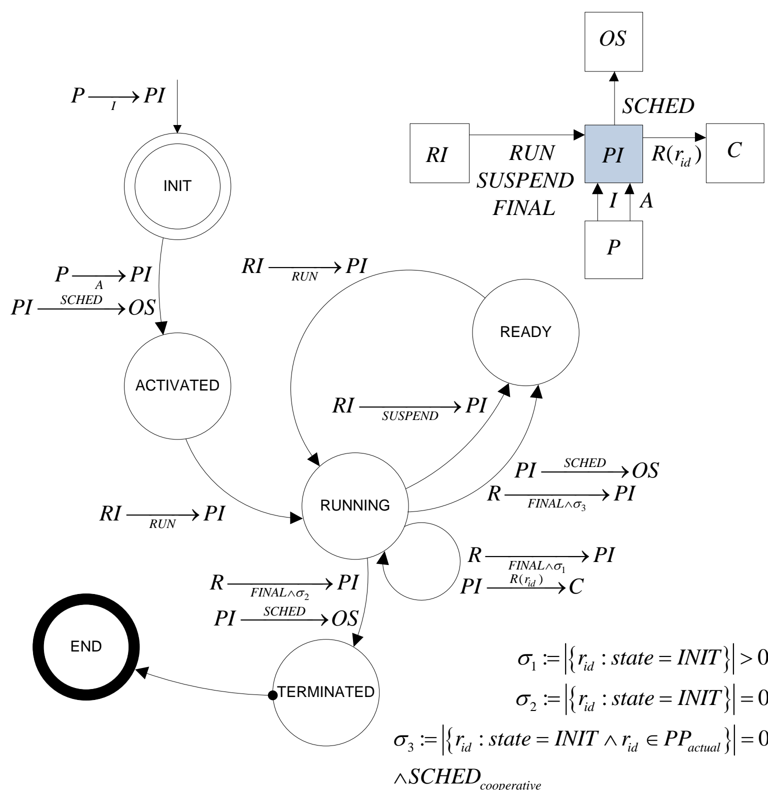

Figure 8.6: Behavioral model of process instance component.

In comparison with other components, a process instance component PI gets initialized whenever the process component sends the signal initialize I, but is not initialized at start of simulation. Additionally, whenever the signal I enters, a new process instance is generated, which allows that multiple process instances of the same process exist in parallel and can also be executed independently.

A process instance has a number of process instance sections, which itself contain a number of runnable instances. All these components get initialized, when event I enters. A process instance additionally has a reference on the runnable instance which should be executed next. At beginning, this reference is assigned to the first runnable instance of the first process instance. The operating system uses this reference, to assign this

runnable instance to a core.

After initialization and when event *activate A* has entered, a process instance sends the event *schedule SCHED* to the operating system, and waits in the state *ACTIVATED*. The operating system assigns the first runnable instance to one of the managed cores. When the runnable instance executes, it sends the signal *RUN* to the process instance, which changes to the state *running*. The cascaded change from *ACTIVATED* to *RUNNING* (from process instance over runnable instance to core) allows to model further behavior at each component, e.g. precedence constraints at runnable instance and prioritization at the core component. When the runnable instance is interrupted at the core, the runnable instance sends the event *SUSPEND* and process instance changes to the state *READY*. When the runnable instance gets resumed, the signal *RUN* enters again, and the process instance changes again into state *RUNNING*. When a runnable instance has finished its execution, it sends the signal *FINAL*. Now the process instance differs between the conditions σ_1 and σ_2.

σ_1 enters when the finalized runnable instance was not the last runnable instance in the actual process section or the actual process section instance is not the last process section instance, i.e. the number of runnable instances in the state *INIT* is higher than 0. Then, the next runnable instance is registered by the event register $R(r_{id})$ at the same core, which executes the previous runnable instance. If the runnable instance is the last runnable of a process instance and there is a cooperative scheduler, the schedule event *SCHED* is sent to the operating system *OS*.

σ_2 enters when the finalized runnable instance is the last runnable instance of the process instance. Then, the process instance sends the event *SCHED* to the operating system in order to notify that the core of the last running runnable instance is available. Afterwards, the process instance changes into the state *TERMINATED*. After termination operations the process instance changes into the final state *END*.

σ_3 enters when the finalized runnable is the last runnable of its process phase and the process is scheduled by a cooperative scheduling algorithm. Then, the process changes to *READY* and the scheduling algorithm is executed.

Runnable Instance

Figure 8.7 shows the behavioral model of the runnable instance.

The runnable instance interacts with the core *C* and the process instance *PI* component. It routes the events *RUN, SUSPEND,* and *FINAL* from the core to the process instance and changes its state according to the entering signals in the same way as the process instance model. The duplication of states is necessary because the core component is only able to interact with runnable instances.

When a core has executed all instructions of a runnable instance, the core sends the event *FINAL* and the runnable instance change into the state *PROCESSING FUNC-*

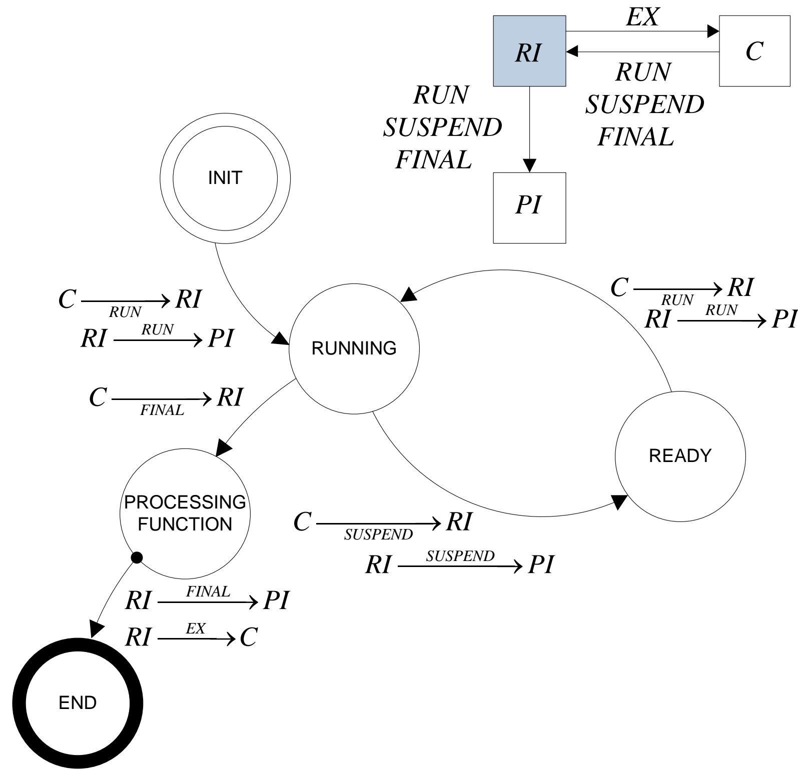

Figure 8.7: Behavioral model of runnable instance component.

TION. In this state, functional code or any dummy code is processed. This is used to model logical functions. However, in this work this property is used only for the model of the scheduling algorithm. When the functional code is processed, the runnable instance sends the event *FINAL* to the process instance. As already mentioned, the process instance is now able to register the next runnable instance at the core. Afterwards the *exit* event *EX* is sent to the executing core, which is afterwards able to start the next runnable instance. Furthermore, a runnable instance includes parameters for saving remaining instructions and the last executed core.

Instruction Model

The instruction model describes the variation of the instructions of a runnable.

An explicit modeling of instructions is applied to allow on the one hand a simulation of heterogeneous hardware architectures, where the number of required cycles for processing one instruction can be configured by a multiplier. Furthermore, cores can have different processor frequencies which results in different execution times. On the other hand an instruction-accurate simulation allows a higher accuracy of the preemption behavior, meaning when a preemption of a runnable instance enters it can be derived when an

instruction is processed. The transformation from instructions to execution time is shown in Section 8.3.4.

When a runnable is triggered, it generates a runnable instance and the instruction model determines the number of instructions, depending on the type of instruction model. There are following types of instruction model:

- Constant Instruction model:

$$I_j = I \ \forall j \tag{8.4}$$

  Each runnable instance has the same number of instructions for all runnable instances j. This model is used for worst-case response time determination in the case of local scheduling.

- Probabilistic Instruction model:

$$P(I_j = I_n) = p_n \tag{8.5}$$

  The number of instructions I of the $j - th$ instance has with a probability p_n the value I_n, whereas the probability follows a distribution function $P()$ as defined in section 6.2.

For both models, the execution time of the task set model has to be transformed into instructions by a normed conversion value respectively the processor frequency.

8.3.4 Hardware Subsystem

The introduced scheduling algorithms in Chapter 7 are developed for symmetric multiprocessing systems, but the following behavioral model of the hardware subsystem allows to model SMP and many heterogeneous multicore processor systems, too. Since a quartz oscillator can be connected to an arbitrary number of cores, it is possible to model processor architectures with cores of different processing speed. This is required for example for the analysis of co-processor architectures, where tasks which use Input/Output ports are allocated to a core with a low processing speed and the tasks which include computation intensive algorithms are allocated to a core with a higher processing speed. Furthermore, the processing frequency of cores can be dynamically changed during simulation time, in a predefined manner or in a dynamic manner by a control-interface for the operating system[7]. In the following, the required behavioral models are introduced.

[7] For example for energy saving purpose, when the processing frequency can be reduced due to a dynamic workload in a low utilization state.

Quartz Oscillator

The quartz oscillator component models the processor frequency variations of the core component. These variations are extracted in a separate model in order to allow synchronous changes of processor frequencies at multiple core components.

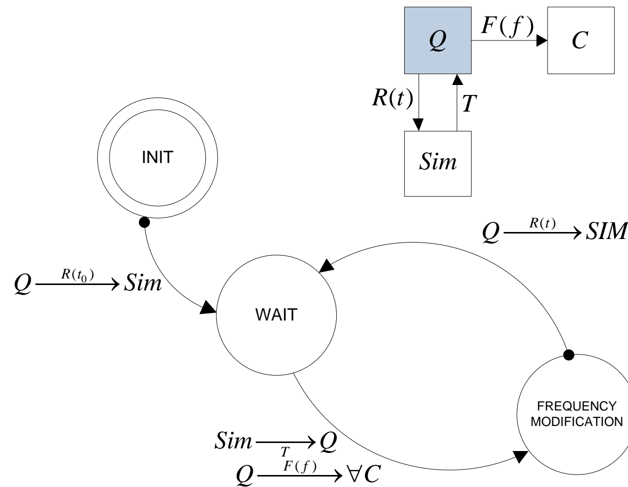

Figure 8.8: Behavioral model of quartz oscillator component.

The processor frequency changes of quartz oscillator components can be modeled in different ways. All components have in common that they register the time when the processor frequency changes the next time at the simulation component.

For this purpose a quartz oscillator component is a *user* component of the simulation sequencer *Sim* and sends register events $R(t)$ and receives trigger events T. Furthermore, it interacts with core components C in order to change processor frequencies by event $F(f)$.

After initialization, the quartz oscillator component Q sends the event register $R(t_0 = 0)$ to simulation sequencer *Sim* and changes in state *WAITING*. The registration at time $t = 0$ is necessary to notify all related cores about the processor frequency. Otherwise, the core has a processor frequency of 0 and no instructions would be executed. In order to limit the number of events, which are sent between core and quartz oscillator components, the quartz oscillator notifies the core only when processor frequency variation occurs, instead of notifying about each processing tick.

When trigger event T enters, the quartz oscillator provides the frequency to all registered core components. When a synchronized processor frequency modification of multiple core components is necessary, it is possible to register these cores to one quartz oscillator component. Then all frequency modifications occur simultaneously. When en-

tering into the state *FREQUENCY MODIFICATION*, the next frequency modification value and its entry time is determined.

There are several models of frequency modifications possible, e.g. a static list of frequency modification or an interface to the scheduler component, which is able to adjust processor frequency dependent of workload on processor.

When the time of the next frequency modification is determined, the next trigger time t is registered by register event $R(t)$ at the simulation sequencer and the quartz oscillator component Q changes again to state *WAITING*.

Core

The core component models the execution of runnable instances on a processor. The execution of runnable instance works in the following way: When a runnable instance starts execution, the core calculates the finishing time of the executing runnable instance by use of the remaining instructions. The core registers a delayed event with this time at the simulation sequencer in the role of a *user* component. When there is no preemption of the runnable instance, the core is triggered when runnable instance has finished. Whenever a runnable instance with a higher priority than the processing runnable instance is registered at the core, the remaining instruction of the executed runnable instance are calculated, the executing runnable instance is suspended, and the next runnable instance is started in the same way.

Runnable instances are prioritized according following schema: All runnable instances, executed in the context of a task, have a priority equal to 0. The scheduler component of the operating system guarantees that only one runnable instance of a task is assigned to a core at the same time.

All runnable instances, executed in the context of an ISR, inherit the priority of the ISR priority level, which is higher than 0.

The core component sorts all runnable instances in decreasing order and executes the runnable instance with the highest priority. When there are multiple runnable instances with the same priority, they are executed in a FIFO manner.

This model is used because it is sufficient for modeling the simple static priority based prioritization of ISRs but also allows to model the complex scheduling of tasks. In analogy, this approach can be compared with a hierarchical scheduling approach, where the low priority scheduler of task instances gives a scheduling decision to a higher priority dispatcher, which only executes the scheduler's decision, when there is no ISR instance allocated to the processing resource.

Figure 8.8 shows the behavior of the core component. The core component C receives from the quartz oscillator component Q the frequency modification event $F(f)$, which sets the core frequency. The operating system component OS sends register events $R(r_{id})$ which request a runnable instance for execution, and it sends unregister events $U(r_{id})$ to remove a registered runnable instance from the core. The core component sends the

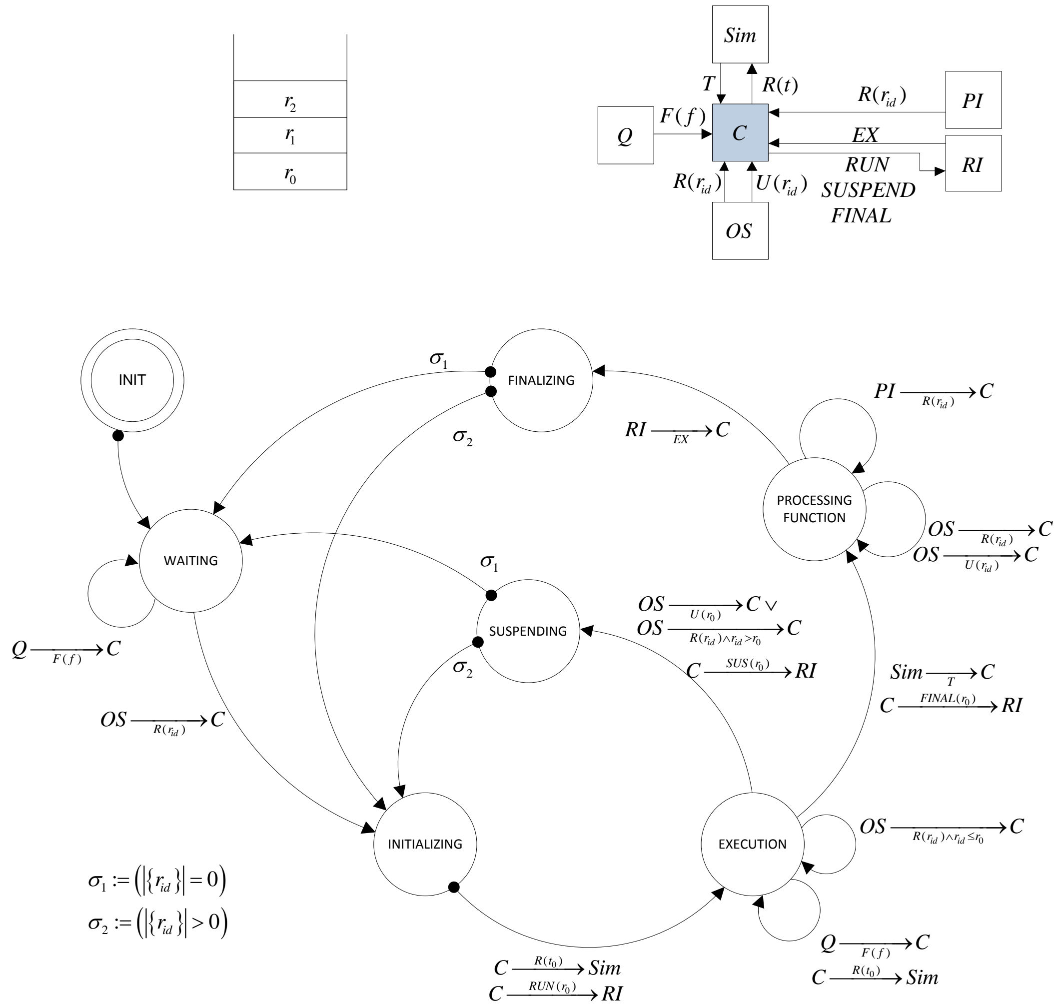

Figure 8.9: Behavioral model of core component.

events *RUN*, *SUSPEND*, and *FINAL* to the runnable instance component to notify about status of execution. The core component receives from the runnable instance component the exit event EX to notify about finishing function processing of runnable instance. The process instance component is able to register runnable instances by registering the event $R(r_{id})$. From the simulation sequencer component *Sim*, the core component receives the trigger event T and sends the register event $R(t)$ in order to determine correct execution times.

Additionally the core includes an *instructions per processor tick* (IPT) parameter. The execution time e of a runnable instance can be calculated by Equation 8.6, in dependence of *IPT*, the actual processor frequency f_a, and the remaining instructions I.

$$e = \frac{\mathrm{I}}{IPT} \cdot f_a^{-1} \tag{8.6}$$

In the following, the behavior of the core component is described. After initialization at start of simulation, the core component C changes without transition condition to state *WAITING*. When the operating system component *OS* registers a runnable instance by event $R(r_{id})$, the core component changes to state *INITIALIZING*. Now, all registered runnable instances are sorted according to the priority of the related process instance, i.e. priority 0 for task instances and the corresponding priority level of the ISR instance.

For the runnable instance with the highest priority level, the remaining instructions are determined and the execution time is calculated by Equation 8.6. Then, finishing time is registered at simulation sequencer, the runnable instance is notified about runnable execution by event $RUN(r_0)$, and the core change in state *EXECUTION*. Now the Transitions 1–4 are possible.

Transition 1: When a quartz oscillator component sends the event $F(f)$, the processor frequency changes and execution time has to be calculated again. When f is denoted as actual processor frequency, t_f as actual finishing time, $f^{'}$ as new processor frequency, and $t_f^{'}$ as new finishing time, than by Equation 8.7 the new finishing time can be determined.

$$t_f^{'} = t + (t_f - t) \cdot \frac{f^{'}}{f} \tag{8.7}$$

Afterwards, this time is registered at the simulation sequencer *Sim* by the event $R(t^{'})$.

Transition 2: Another transition enters when the operating system *OS* registers a new runnable instance r_{id} by the event $R(r_{id})$. When the priority of the actual runnable instance r_0 is lower or equal than the r_{id}, the new runnable instance is pushed to the runnable instance queue of this core component. Contrary, when the new runnable instance has a higher priority than the executing runnable instance or the operating system *OS* unregisters the executing running task instance by the event $U(r_0)$, the executing runnable instance stops execution. Then, the remaining instructions has to be calculated, in order to be available when this runnable instance executes next time. This can be done by rearanging Equation 8.6 to instructions I, whereby e equates the remaining execution time. Furthermore, when the *OS* unregisters the runnable instance, it is removed from core queue of runnable instances. In both cases, the core component sends the signal $SUS(r_0)$ to the runnable instance, and changes into state *SUSPENDING*.

From state *SUSPENDING* the core component changes automatically in the state *WAITING* or *INITIALIZING*. In the case of condition σ_1, meaning the number of remaining runnable instances is zero, core component changes in state *WAITING*. In the case of condition σ_2, meaning the number of remaining runnable instances is greater zero, the core component changes again to state *INITIALIZING* and starts execution of next runnable instance.

Transition 3: During the execution of a runnable instance, it is possible that a quartz oscillator component Q sends a frequency event $F(f)$. The modification of execution

time can be determined in the same way as in state *WAITING* (by Equation 8.7), but additionally the new time when execution finishes is registered at *Sim* by event $R(t^{'})$.

Transition 4: When the core component gets the trigger event T from the simulation sequencer *Sim*, the runnable instance gets the event $FINAL(r_0)$, starts with processing of the functional code, and the core component changes in the state *PROCESSING FUNCTION*.

In the state *PROCESSING FUNCTION*, it is possible that the functional code of the executing runnable instance includes a function which registers new runnables. In this work, only the operating system component (i.e. the scheduler function) has a functional code and is able to send event $R(r_{id})$. Alternatively, the process instance of the finished runnable instance registers the next runnable instance.

When the processing of the functional code is finished, the executed runnable instance component sends the exit event *EX* to the core component, which changes in state *FINALIZING*. When there are no runnable instances registered at core component (condition σ_1), the core component changes again in the state *WAITING*. When there are runnable instances registered (condition σ_2), the core component changes into state *INITIALIZING*.

8.3.5 Operating System Subsystem

Task instances are assigned by a scheduler to core components and ISR instances are assigned by an interrupt management unit to core components.

For simplification of interrupt management, ISR instances are mapped statically to cores. Furthermore, the prioritization of ISR instances is done by the core component, as described in the previous section. Nested interrupts[8] are allowed and ISRs with the same priority are executed by FIFO manner. Multiple ISR activation is allowed, where ISR instances of the same ISR are prioritized by activation time.

For the assignment of task instances, the scheduling algorithm routine is modeled by an ISR, whereby the scheduler ISR has the highest priority of all ISRs. In dependence of the scheduling algorithm, the scheduler manages a number of cores of the multicore processor, mentioned as core cluster. The core cluster can include one core (Table 3.2, group *AI*), a subset of cores (Table 3.2, group *AII*), or all cores of a multicore processor (Table 3.2, group *AIII*). When a scheduler call[9] enters, the scheduler determines from a queue of ready task instances and running task instances with a scheduling policy the x

[8]Nested interrupt mentions the case, when an ISR instance executes and a higher priority ISR instance is activated, then the executing ISR instance is suspended, the higher priority ISR instance executes, and when the higher priority ISR has finished, the suspended ISR resumes at the suspended position.

[9]In general, the scheduler is called at task activation, task termination, cooperative schedule point, or scheduler specific points in time.

highest priority task instances, whereby x equates the number of cores in the core cluster. The scheduler always assigns only one task instance to a core at the same time. The actual runnable instance of this task instance immediately executes, as long as there is no runnable instance of an ISR instance allocated to the core, because runnable instances of ISRs instances always have a higher priority than runnable instances of task instances.

Algorithm 1 Pseudo-code of generic scheduler.

```
Require: {a} (core cluster), {f} (available cores), {j} (ready tasks), {i} (running tasks),
  {k} (nominee tasks), {s} (sorted tasks)
  { Step 1: Check non-blocked cores}
  for all {i} do
    {f} ← BLOCKING(i)
  end for
  { Step 2: Find nominated tasks }
  for all {i} do
    for all {f} do
      {k^f} ← NOMINEE(i,f)
    end for
  end for
  for all {j} do
    for all {f} do
      {k^f} ← NOMINEE(j,f)
    end for
  end for
  { Step 3: Sort nominated tasks}
  for all {f} do
    {s^f} ← SORT({k^f})
  end for{ Step 4: Dispatching Schedule}
  for all {i} do
    if Rank(i,{s^1, ..., s^f}) ≤ |{f}| or i ∉ {f} then
      REMOVE(i)
    end if
  end for
  for all {s} do
    if Rank(s) < |{f}| then
      SUSPEND(i)
    end if
  end for
  for all {s} do
    if Rank(s) ≥ |{f}| then
      RESUME(s)
    end if
  end for
```

Algorithm 1 shows the pseudo code of the generic multicore scheduling routine. This routine implements the scheduling model from Section 3.1.3 and can be specialized by all scheduling algorithms from Table 3.2. The first step determines whether a core is blocked

by a running tasks. A core can be blocked in the case of a non-preemptive (group DI) or purely cooperative scheduling algorithm (group DII), e.g. the algorithms *P-ERfair-PD*^{2} or *Pfair-PD*^{2}. Therefore, all running task instances in i are checked whether they block its related core in x. The result is stored in f, which finally includes a list of available cores.

The next step determines by the function *NOMINEE* whether a task is nominated for execution on a core. Constraints can prevent the task from execution on a certain core. This can be a task instance with non fulfilled precedence constraints[10], e.g. in case of non-work-conserving algorithms (group *WII*) or when a scheduling algorithm works in a bounded-migration manner[11] (group *MI*,*MII*,*MIII*). The result is a two dimensional array which has a list of task instances for each core which can be executed on the core. Note that a task either is nominated for one core (in the case of group *MI*, *MII*, and *AI*), or for all cores (in the case of all other groups).

When the task nominee lists have been created, the task instances are sorted according to scheduling algorithm policies. When a task instance is able to execute on multiple cores it is only considered once in the sorted list. Therefore, the list of sorted task instances s has the same number of elements as lists $i + j$. The function *RANK(s)* returns the rank of a task instance s in the sorted list s, whereby a task instance with rank 1 has the highest scheduling policy value.

The final step dispatches the task instances to the cores in the following manner. Task instances which execute at the moment and have a rank in the nominee list less or equal f or block a core are removed from nominee list and stay in the state *RUNNING* on the executing core. Task instances which execute at the moment and have a rank in nominee list higher than f and don't block a core are suspended by sending the unregister event $U(r_{id})$ with the runnable instance r_{id} of the executing task instance to the *Sim* component. All other task instances, which are on the list of ready tasks and have a rank in nominee list less or equal f are started for execution by sending the register event $R(r_{id})$ to the *Sim* component, with the last running runnable instance r_{id} of the task instance.

The listings of the functions *BLOCKING* and *SORT* are dependent from the scheduling algorithm. They can be found for the analyzed algorithms in Section A.

8.4 A Metric for Real-Time Examination

For the examination of deadline compliance of a single task, in general the response time metric is used. However, for evaluation of a complete task set, there are often only true-false statements whether all tasks meet their deadline or don't. In the following, a metric

[10] Then the task instance is not able to execute on any core

[11] Then a task instance at a core which is once started is not allowed to execute on any other core

is introduced which allows a more precise examination of real-time properties on task set level, denoted as *maximal Normed Lateness* (*mNL*) .

The lateness $L_{i,j}$ (Equation 4.2) of a task job $T_{i,j}$ equates the exceeding of its deadline. When the lateness is negative, the deadline is met. In order to calculate the *mNL* metric, the lateness $L_{i,j}$ is standardized by the task deadline $T_i.d$, representing a percentaged delay. From the normed lateness value of all jobs, the maximum can be determined which equates the worst handled job of all tasks related to the percentaged delay. Since the deadline is assumed to be constant for all jobs of a task, the calculation can be simplified as shown in Equation 8.8.

$$mNL(\tau) = \max_{T_i \in \tau} \left(\frac{\max\limits_{T_{i,j} \in T_i} (L_{i,j})}{T_{i,j}.d} \right) \tag{8.8}$$

This metric is proposed, because of the need for an one-dimensional value, which allows a comparison of different system models, i.e. an evaluation of the deadline compliances for different task sets. Furthermore a metric which allows a quantitative evaluation of the deadline compliances of all tasks in a task set is more representative than a simple true/false statement about the deadline compliance. Therefore, it is assumed that not the absolute distance from task finalization to deadline is important but the distance, relatively to the deadline. This is valid, as long the assignment of processing time for a task scales with the inverse of the task deadline, which in turn is valid for all deadline driven scheduling algorithms or scheduling algorithms which derive a static task priority from the task deadline. Furthermore, the lateness is chosen instead of the response time. This is not necessarily required but allows to set the bound of deadline compliance to 0 which is intuitive instead of a bound of 1 in the case of the response time.

The *mNL* metric is used in the following for schedulability examination of a complete task set.

8.5 Approximation of Bounds for Schedulability

The target of a *schedulability analysis* is to determine the worst-case scenario for a task, where its response time is maximized or grows unboundedly. In a *schedulability examination*, the target is to modify system model parameters in a way that the worst-case (and best-case) scenario is approximated. The response time is affected by system model parameters in the following way. As long as the scheduling algorithm has no memorization of passed events, it simply assigns priorities to task instances in a deterministic manner. Therefore, there is no property, which can be modified in order to change task response time. Similarly the processor has a constant processor frequency and a constant number of cores in this work. Therefore, these components don't have to be considered for the approximation. The only property which results in different schedule decisions is the workload of tasks, namely the execution time and the inter-arrival time. For schedu-

lability examination of local scheduling algorithms, the assumption of the worst-case execution time is sufficient. As shown in Section 4.3 the worst-case response time has its maximum at maximal interval of occupied processor time. Therefore, a lower execution time of any task results in a lower response time, this property is also denoted as sustainability. Furthermore, the worst-case response time for local scheduling algorithms occurs when all tasks are released at the beginning and all further task arise with the minimal inter-arrival time, this property is also denoted as critical instant.

For examination of schedulability for global scheduling algorithms, the critical instant and the worst-case execution time don't necessarily result in the worst-case response time, as shown in the examples of Section 4.3. Therefore, it is necessary to determine the worst-case response time by modifying both, task execution time and task inter-arrival time.

For the execution time variation, execution times,i.e. instructions, are randomly generated by the definition of the probability function.

For the inter-arrival time variation, periodic offset based task sets with MTB extension are assumed. Therefore, the only parameters which can be modified is the time base parameter. It is also possible to model a sporadic task activation with the MTB task set by assigning only one sporadic task to a time base.

At start of simulation, all parameters of the time bases have the minimal value, meaning $f^v(t = 0) = 1$. Then, a successive variation approach is applied. The approach changes activation patterns between the initial time base parameter and a fraction of the maximal time base parameter. In order to prevent a repeating sequence of activation patterns, time base parameter are varied at prime numbers which guarantees that no inter-arrival time of any task is a multiple of the interval of time base parameter variation.

Therefore, the task with the highest inter-arrival time is selected and the next prime number which is at least 5-times higher[12]. This value is denoted as δ^M. At $t = \delta^M$, the frequency multiplier $f^1(t)$ of the first time base b^1 is modified by a fraction delta $+\delta$. At $t = 2 \cdot \delta^M$, the frequency multiplier $f^1(t)$ is again modified by $+\delta$. This modification repeats until a fraction of maximal frequency multiplier $f^1(t) = \delta^1_{\max}$ has been achieved. Afterwards, after each time interval of length δ^M, the frequency multiplier will be decreased until reaching the initial value. Then, the frequency multiplier of the second timebase is modified by $+\delta$. Afterwards the approach of time base variation of the first time base starts from the beginning. When the first time base reaches again the initial value, the second frequency multiplier changes by $+\delta$ and the process starts from ramping time base one. This repeats until the second time base reaches a certain multiplier value, and afterwards the frequency multiplier is ramped downwards in the same way as the first time base. When the frequency multiplier $f^2(t)$ of time base b^2 reaches again the initial value, the third time base b^3 is modified and so on. When the last time

[12]This value was determined empirically as good trade-off between low simulation duration and good worst-case response time approximation.

base has ramped upwards and downwards, the process starts from beginning with an increased $+\delta$ and an increased limit of the frequency multiplier $f^v(t)$. The approximation stops when the maximal determined response-time of all tasks does not change for one complete ramping process for all time bases.

8.6 Technical Implementation

The discrete event-based simulation is realized as a C++ application. For simulation performance, the simulation model and discrete-event simulation core were implemented from the scratch in order to prevent runtime overhead from any framework. Therefore, it was possible to achieve a very low experiment execution duration. For the models, comparable to the models of the case studies, a simulation time can be achieved which is four times lower than the simulated time. This is a performance benefit of factor 1,000 in comparison with existing simulation tool solutions. The GNU scientific library [GNU10] was used for uniform and Weibull distribution functions and the pseudo-random number generator Mersenne-Twister [MN98]. For parallelization of the simulation of multiple models, a cluster computing cloud was built and managed with the Condor High Throughput Computing framework [FTF$^+$02].

Chapter 9

Sensitivity Analysis of Probabilistic System Models

This chapter describes an approach for the evaluation of scheduling algorithms for multicore processors. The application of the approach is split. On the one hand, a quantitative comparison of the fitness of scheduling algorithms for application in a specific real-time system is possible, where changes of task set parameter over the lifetime of a real-time systems are considered. On the other hand, a sensitivity analysis of the effect of system characteristics like task set utilization on system metrics like deadline compliance can be evaluated.

9.1 Task Set Parameter Variations

The real-time examination methods of Section 4.2 and the simulation-based real-time examination approach determine metrics, e.g. the response time, for a single real-time system model. But, in order to compare existing and proposed scheduling algorithms for multicore systems, the evaluation of a single model is not representative. Changes in task set configuration produce different results in system evaluation. These changes occur multiple times in the development and lifetime of embedded systems.

During the development of the embedded system, task execution times change due to varying content of executed functions entities, e.g. some parts of the program are not implemented or implementations are improved. Furthermore, task activation patterns change, e.g. through changes in hardware, peripherals, or connected embedded systems.

During the lifetime of the embedded system, task set parameters also change. For example software updates are applied, including fault repairs[1], software upgrades with additional functions, or software changes due to modified hardware components. Furthermore, aged hardware might change execution time, e.g. due to changed input data

[1] E.g. software updates are applied during automotive service inspections.

which increase runtime of algorithms.

Since a determined metric of a real-time system strongly depends on the task set, strictly speaking all possible task set configurations of the development and lifetime of the embedded system have to be considered when benchmarking scheduling algorithms.

Though, in typical embedded systems the dimension of problem space of possible task sets is in a range where an exhaustive evaluation is not applicable due to required computation effort. Furthermore, an exhaustive evaluation does not consider the probability of task set parameters and therefore the frequency of metric values, e.g. response time, is not known, too.

Sensitivity analysis approaches are efficient mechanisms to support the embedded system designer at these variations. For example, sensitivity analysis has been introduced for bounding upper limits on task deadlines at EDF scheduling for singlecore processor systems [ZBB10a], minimal possible task periods [ZBB10b], or several other use-cases [ZBB09, DRRG10].

The Sensitivity Analysis of Probabilistic Systems (SA-PS) approach [DSM+10b] determines metrics in dependence of task set parameter variations.

The variation of task set parameters is expressed by a probabilistic system model. By use of this probabilistic system model, the SA-PS approach uses a Monte Carlo sampling technique [MU49]. Task set samples are generated, according to a system model with probability density functions of the task set parameters. Then, these task set samples in combination with the remaining model components (processor and scheduler model) are analyzed by a single model examination approach and metrics are determined. Since the Monte Carlo approach only samples a subpart of all possible task sets, originating the probabilistic description, a consideration of the confidence of determined metrics is necessary. For this purpose, the SA-PS approach assigns the system models to clusters regarding a certain system model characteristic, e.g. the task set utilization $U_{sum}(\tau)$, the average task utilization, or any other characteristic which can be determined from system model parameters. For each cluster, statistical estimators on metrics are calculated in order to describe e.g. the range, the variability, or the median of values. These statistical estimators could be used for a sensitivity analysis, however the confidence of the statistical estimators is missing. This is important because Monte-Carlo randomization is only a sampling method. Therefore, the bootstrapping approach [ET93] is applied, which determines the confidence interval of the statistical estimators.

The following sections describe the steps of the SA-PS approach.

9.2 Probabilistic System Model

The sensitivity analysis requires a probabilistic description of the task set, defining the range and the distribution of task set parameters, in order to generate task sets rep-

resenting possible stages in development or lifetime of the embedded system. For this purpose, the task set model τ (see Section 2.2) with the MTB extension (see Chapter 6) are extended by a probabilistic description. A task set τ includes a number of tasks $\{T_i\}$, where a task T_i has the task parameters $T_i = (p, e, d, b^v)$. When $T_i.z$ describes any property $z \in \{p, e, d, b^v\}$ of a task T_i, then the probabilistic description of the model parameter v origins from following definition:

$$P(T_i.z = z_n) = x_n, \tag{9.1}$$

whereby $z = f(x)$ is the probability density function. The probability of values Z, falling into an interval $[z_1, z_2]$, can be derived by [Das10]:

$$P[z_1 \leq Z \leq z_2] = \int_{z_1}^{z_2} f(x)\,\mathrm{d}x.$$

The kind of probability density function depends on the variation of task set parameters. In general, measurements of execution time $T_i.e$ are available for different embedded system projects or different stages of the development and lifetime of an embedded system. Furthermore, variation of inter-arrival times $T_i.p$, deadlines $T_i.d$, or time bases $T_i.b^v$ are recorded. This information is used to determine the probability density function, e.g. a discrete probability function or a Weibull probability function as introduced in Section 6.2.

Depending on the kind of variation, this work distinguishes between the following two types of probabilistic system models.

9.2.1 Probability of Task Set Parameters

The model for probabilities of task set parameters has for all task sets $\{\tau\}$ of the probabilistic task set τ^P a constant number of tasks in a task set.

Therefore, for each task $T_i \in \tau^p$ and each task parameter $T_i.z$ $z \in (p, e, d, b^v)$, a probability density function according to Equation 9.1 is defined. In the case of a constant value for a task parameter, there is a constant value instead of a probability density function.

A typical application of this kind of probability model is the case when the number of tasks in a task set is constant, but the execution time varies e.g. due to the changing functional code.

9.2.2 Probability of Task Quantity

For the case that the quantity of tasks changes, the probabilistic model description has to be extended by a probabilistic description of task quantity regarding Equation 9.2 with the probability density function $z = f(x)$.

$$P(|\{T_i\}| = z_n) = x_n \tag{9.2}$$

When the task quantity is based on a probability density function, task parameters can not be specified in the probabilistic model description for a certain task of the task set. Therefore, the probabilistic model description includes one probability density function $T_i.z = f(x)\ \forall T_i \in \tau^P$ for each task parameter $z \in \{p, e, d, b^v\}$.

A real-time system model with these kinds of probabilistic task set τ^P is denoted as probabilistic system model S^P. It includes the components: probabilistic tasks set τ^P, scheduler model ξ, and processor model Π.

9.3 Monte-Carlo Randomization

For evaluation of metrics for the probabilistic system model S^P, the introduced real-time system examination approach from Chapter 8 is not applicable directly because the probability functions doesn't define variations of task set parameters during the execution of the real-time system but they define variations between the different stages of a real-time system. Therefore, an approach for creating real-time system models S originating the probabilistic system model S^P is necessary.

In order to generate a representative selection of real-time system models, the SA-PS approach applies *Monte-Carlo randomization*, presented by Metropolis and Ulam [MU49].

Monte-Carlo randomization was initially introduced as a methodology to determine numerical solutions of mathematical problems. An application example of Monte-Carlo randomization which can be compared to the application of the Monte-Carlo randomization in this work is the determination of the mathematical constant π, shown in Figure 9.1. A random number generator produces pairs of two numbers in the range of $[0, 1]$. For each pair, representing x and y coordinates, the approach determines whether the point of the coordinates is inside the upper right quarter of a circle with a radius of 1 or it is outside the circle. The ratio between points inside of the circle and all generated pairs represents a quarter of the area of a circle with an arc radius of 1, which is equal to $\frac{\pi}{4}$.

The objective of the SA-PS approach is to determine the effect of system characteristics variation on a system metric, e.g. the maximal normalized lateness. The probabilistic system model is a description of a multi-dimensional exploration space. For analysis of this exploration space it is not significant how a change of a single value of a model parameter of one component, e.g. the execution time of a task, effects a system metric. Detecting this kind of effects does not allow to make decisions in software development, because at changing parameter of another component, the system metric could change, too. Therefore, a more generalizing characterization is necessary, which allows to make

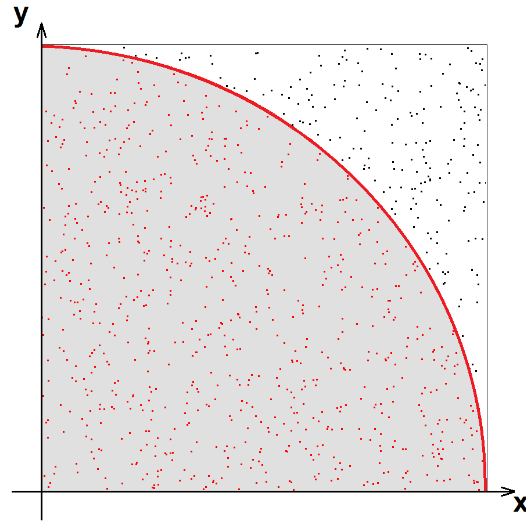

Figure 9.1: Application of Monte-Carlo randomization in order to determine the value of π.

decisions at the software development, independently of a certain model parameter but in dependence of a certain system characteristic, e.g. the system utilization $U_{sum}(\tau)$.

In order to determine such a bound for a system characteristic from a probabilistic model, an exhaustive evaluation of the exploration space would lead to an exact bound. However an exhaustive evaluation is not possible due to required calculation time.

Therefore, the SA-PS approach applies the Monte-Carlo approach. Randomly generated task sets represent a sample of the probabilistic system and allow to determine the effect on a system characteristic. When plotting the determined metric values as a function of the system characteristic values, the exploration space is successively analyzed. The space, clamping by the analyzed values, can be analyzed afterwards by clustering of system characteristic values and determining statistical estimators for the cluster. Since the analyzed system models represent only a sample of the system, confidence intervals of the determined estimators have to be calculated in order to compensate the missing analyzed models.

A system model S is generated from the probabilistic system model S^P in the following way. Since model parameters are given as probability density function, the following transformation has to be applied for the generation of parameter values, whose distribution approximates the probability density function. A pseudo-random number generator produces uniformly distributed values in the range $[0, 1]$ representing a probability p. Additionally, from the probability density function $f(x)$ a cumulative distribution function $F(x)$ is determined, with $F : \mathbb{R} \rightarrow [0, 1]$. By passing the pseudo randomly generated numbers in the inverse function of $F^{-1}(p)$, values according to the probability density

function $f(x)$ can be determined. This method is also known as inverse transformation [SD88].

For generation of random uniform values it is important to generate repeatable series of random values to be able to replicate examinations of characterization metrics. Therefore, a pseudo-random number generation is necessary. The pseudo random-number generator (P-RNG) has to fulfill a number statistical tests, guaranteeing a long period and having a high quality of randomness. The SA-PS approach uses the Mersenne-Twister, introduced by Matsumoto and Nishimura [MN98]. The Mersenne-Twister has a long period of 2^{19937} and fulfills numerous tests of randomness [MN98], e.g. the DIEHARD battery of statistical tests [Mar96].

In order to improve evaluation duration, system models including a task set which is not feasible are rejected. The SA-PS approach applies the following feasibility checks on the generated task sets:

- **Generalized density check**: Whenever the maximal generalized density

$$\lambda_{\max}(\tau) > 1, \tag{9.3}$$

  there is a task $T_f \in \tau$, which has a task deadline $T_f.d$ or a task inter-arrival time $T_f.p$ shorter than the task execution time $T_f.e$. In the first case, this task can not fulfill its deadline, because $T_{f,j}.A + T_f.e > T_{f,j}.D\ \forall j$. In the second case, this task is overloaded, meaning in any case the next task instance is started before the previous task instance has finished. For the case of a global scheduling algorithm, this scenario is feasible when successive task instances are allocated to different cores (at least the $\lambda_{\max}(\tau) - 1$ successive instances, when there is no other task allocated to the core). However, for the case of local scheduling algorithms, the delay of task instances constantly increases at least by the value $T_f.e - T_f.p$ which necessarily results in a violation of task deadlines in finite time. Since the SA-PS approach compares local and global scheduling algorithms, task sets violating Equation 9.3 are rejected.

- **Task set utilization check**: Whenever the system utilization

$$U_{sum}(\tau) > m, \tag{9.4}$$

  the task set τ is not feasible, because the task set execution requirement exceeds the available computation capacity of the real-time system. Therefore, a task set which violates Equation 9.4 is rejected.

9.4 Examination of Characterization Metrics

In the next step, the SA-PS approach determines a metric, e.g. the response time, of the system model. It depends on the properties of system model S, namely task set τ, scheduler ξ, and processor Φ, and the metric itself, which type of real-time system examination method can be applied.

Because introduced schedulability examination approaches from Section 4.2 are not able to analyze the presented scheduling algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2}, the simulation-based multicore real-time examination approach has to be applied to these algorithms.

For the case of local multicore scheduling algorithms, real-time system examination approaches from section 4.2 can be applied.

9.5 Classification of System Models

The previous step determines a vector of metrics $\{M(S)\}$ for all generated system models $\{S\} \in S^P$. The target of the sensitivity analysis is to determine the interdependency of $\{M(S)\}$ with system characteristics $\{C(S)\}$, for example $C(S) = mNL(S)$, and to compare the dependencies of different scheduling algorithms.

The interdependency can be analyzed by correlation evaluation of both vectors, plotted in a scatter diagram (as shown in the case studies). The cloud represents the interdependency of a certain metric value with a certain system characteristic and bounds of achieving a certain metric value can be derived.

For the quantitative comparison of scheduling algorithms a correlation evaluation is not sufficient. When plotting multiple clouds in one scatter diagram, overlapping sections could occur and it is not possible to determine which algorithm is more adequate to achieve a certain metric value. Therefore, the range of system characteristics

$$[\min_{S \in S^P}(C(S)), \max_{S \in S^P}(C(S))]$$

is divided in a number of equal sized clusters $\{Z_k\}$ $k \in \mathbb{N}^*$ with the cluster size z. All system models $\{S\}$ are assigned to one of these clusters according the following equation:

$$Z_k = \{S|(k-1) \cdot z \leq C(S) < k \cdot z\}. \tag{9.5}$$

9.6 Statistical Evaluation

After this classification, two scheduling algorithms ξ_a and ξ_b can be compared according to the following approach.

For each cluster Z_k, estimators $\sigma(Z_k)$ of the metric values $M(S)$ in this cluster are determined. For the comparison of ξ_a and ξ_b, the SA-PS approach determines the $Q_{.01}$

and $Q_{.99}$ quantile to estimate the range of cluster values and the $Q_{.5}$ quantile to determine the median value of cluster values. Further statistical estimators are possible.

As already mentioned in previous sections, Monte-Carlo randomization only covers a subset of all possible system models, where the degree of coverage depends on the number of generated system models. Therefore, it is necessary to determine confidence intervals of the determined estimators. For this purpose, the SA-PS approach applies the bootstrapping approach, introduced by Efron and Tibshirani [ET93].

Bootstrapping is a method to determine the confidence interval of statistical estimators of a number of random variables by resampling the random variables. From a set of random variables $(x_1, ..., x_n)$ bootstrap samples $X_b = (x_1^*, ..., x_n^*)$ with $b = 1, ..., B$ are randomly selected with replacement. For each bootstrap sample X_b the statistical estimator $\sigma_b(x_1^*, ..., x_n^*)$ is determined. Finally the distribution function $F(\sigma_1, ..., \sigma_B)$ is determined and confidence intervals can be calculated, e.g. through quantiles.

For the comparison of scheduling algorithms, in the first step the statistical estimators are calculated and in the second step the confidence bounds are determined by the bootstrapping approach.

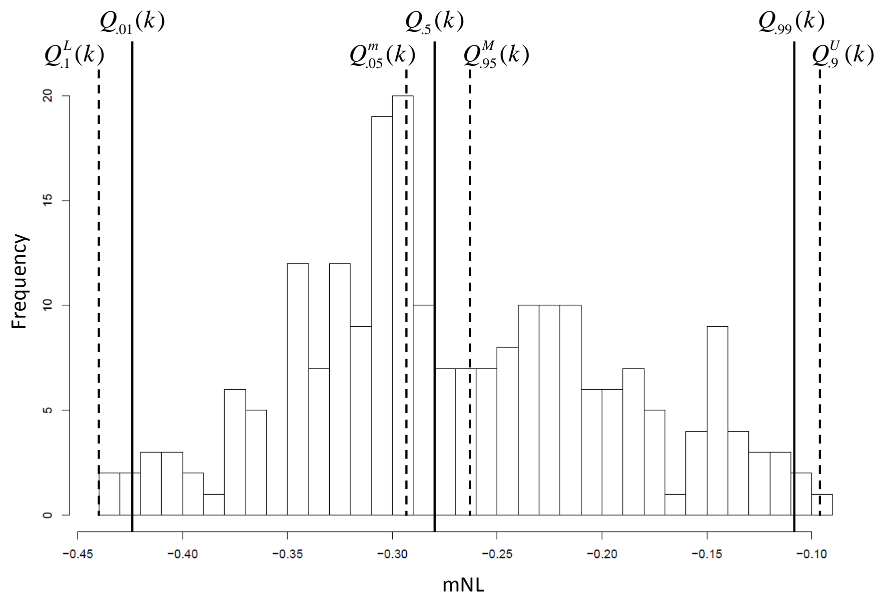

Figure 9.2: Statistical estimators and bootstrapped confidence bounds of a distribution of metric values in a cluster Z_k.

Figure 9.2 shows the distribution of *mNL* metrics values in a cluster k. For this distribution, the statistical estimators $Q_{.01}(k)$, representing the 1 percent quantile, $Q_{.99}(k)$, representing the 99 percent quantile, and $Q_{.5}(k)$, representing the median, are calculated (shown as continuous line in Figure 9.2). In the next step, the confidence bounds (shown as dotted line) are calculated for these estimators. $Q^L_{.1}(k)$ equates the 10 percent quan-

tile of the bootstrapped statistical estimator $Q_{.01}(k)$. Informally, this value defines the confidence bound that with a probability of 10 percent one percent of the metric values have a lower value. In a similar way, $Q^U_{.9}(k)$ is determined for the statistical estimator $Q_{.99}(k)$. $Q^U_{.9}(k)$ equates the 90 percent confidence bound of the 99 percent quantile of metric values.

$Q^M_{.05}(k)$ and $Q^M_{.95}(k)$ represents the 90% confidence range of the median of the distribution of metric values in cluster k, meaning the median of the distribution of metric values has a value which is with a probability of 90 percent between $Q^M_{.05}(k)$ and $Q^M_{.95}(k)$. With these estimators it is possible to compare a cluster Z_k for two scheduling algorithms.

For the evaluation of scheduling algorithms, the SA-PS approach compares the upper and lower confidence bounds, $Q^U_{.9}(k)$ and $Q^L_{.1}(k)$ respectively, and the confidence range $(Q^U_{.9}(k) - Q^L_{.1}(k))$ for a certain cluster k. Depending on the relation of these values for a cluster, two scheduling algorithms ξ_b and ξ_b can be compared according the following definitions.

The first comparison is the predictability of a metric, meaning the statistical spread of metric values in a cluster. The predictability is defined in the following way:

Definition 9.1
Predictability. *A metric of a scheduling algorithm ξ_a has a higher predictability than the metric of a scheduling algorithm ξ_b, when $(Q^U_{.9}(k)_a - Q^L_{.1}(k)_a) < (Q^U_{.9}(k)_b - Q^L_{.1}(k)_b)$, meaning the absolute value of the range between the upper and lower confidence bounds of scheduling algorithm ξ_a is lower than the absolute value of the range between the upper and lower confidence bounds of scheduling algorithm ξ_b.*

A scheduling algorithm, which has a higher predictability in comparison with another scheduling algorithm is less sensible for task set variations as long as the task set utilization variations don't exceed the cluster boundaries. This is beneficial for the design process of embedded systems, because metric values can be estimated with a higher confidence.

The second evaluation is a comparison of the metric values, whereas it is assumed that a high metric value is better than a lower one. It is differed between two cases. The first case is that scheduling algorithm ξ_a is sometimes better but never worse than scheduling algorithm ξ_b:

Definition 9.2
Weak Predominance (Maximum Value). *When the target of a benchmark is to maximize a metric, a scheduling algorithm ξ_a is sometimes better but never worse than a scheduling algorithm ξ_b, when $Q^U_{.9}(k)_a > Q^U_{.9}(k)_b$ and $Q^L_{.1}(k)_a \geq Q^L_{.1}(k)_b$, meaning the upper confidence bound of scheduling algorithm ξ_a is higher than the upper confidence bound of scheduling algorithm ξ_b and the lower confidence bound of scheduling algorithm ξ_a is higher than or equal to the lower confidence bound of scheduling algorithm ξ_b.*

This formulation is used because it is often the case that the range between the upper and lower confidence bounds of two scheduling algorithms overlap. If they don't overlap, a tight statement is possible:

Definition 9.3
Strong Predominance (Maximum Value). *When the target of a benchmark is to maximize a metric, a scheduling algorithm ξ_a predominates a scheduling algorithm ξ_b in relation to a metric, when $Q^L_{.1}(k)_a > Q^U_{.9}(k)_b$, meaning the lower confidence bound of scheduling algorithm ξ_a is higher than the upper confidence bound of scheduling algorithm ξ_b.*

For the case that a low metric value is better, in Definition 9.2 the comparators have to be inversed:

Definition 9.4
Weak Predominance (Minimum Value). *When the target of a benchmark is to minimize a metric, a scheduling algorithm ξ_a is sometimes better but never worse than a scheduling algorithm ξ_b, when $Q^U_{.9}(k)_a < Q^U_{.9}(k)_b$ and $Q^L_{.1}(k)_a \leq Q^L_{.1}(k)_b$.*

Furthermore, in Definition 9.3 comparators have to be inversed and confidence bounds have to be transposed:

Definition 9.5
Strong Predominance (Minimum Value). *When the target of a benchmark is to minimize a metric, a scheduling algorithm ξ_a predominates a scheduling algorithm ξ_b in relation to a metric, when $Q^U_{.1}(k)_a < Q^L_{.9}(k)_b$.*

With these definitions it is possible to compare different scheduling algorithms for a probabilistic system model, as shown in the case studies in the next part.

Part III

Case Studies

Chapter 10

Execution of Case Studies

This chapter describes three case studies, applying the contribution of this work to the domain of automotive powertrain systems. The first case study compares the presented global scheduling algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2} regarding the application in automotive systems, including multiple task trigger sources e.g. periodic activation, CAN field bus, or FlexRay bus. The second case study compares *Partly-Pfair-PD*^{2}, *P-ERfair-PD*^{2}, and the local scheduling algorithm EDF with bin-packing heuristic Worst Fit Decreasing (WFD) for a group of automotive powertrain system, ported from a singlecore to a dualcore processor. The third case study is a robustness evaluation of the algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2}.

10.1 Automotive Systems

This section gives a brief overview of the architecture of automotive systems and how the system model is derived.

In a typical automotive system, there are multiple Electronic Control Units (ECUs), including functions of different parts of the system, e.g. entertainment, car body, or powertrain category. In today's upper class automotive systems, there are up to 100 ECUs. For sharing of information between these ECUs, data-frames on bus systems activate tasks on ECUs. Depending on the transmission type of the bus system, data-frames arrive in defined time slices or data-frames arrive sporadically due to prioritization. This is one kind of task activation. Another kind of activation is an internal timer, which periodically activates task in a constant manner. Furthermore, in the case of a powertrain system, tasks are activated in dependence of the position of the crankshaft. This complex activation behavior can be modeled by the presented models for demand of execution time and the MTB extension. The execution time is modeled by probabilistic functions.

In order to give a better understanding of the complexity of automotive powertrain systems, figure 10.1 shows all sensors and actors which are controlled by an engine management system (sensors and actors which are accessed via other ECUs are not shown).

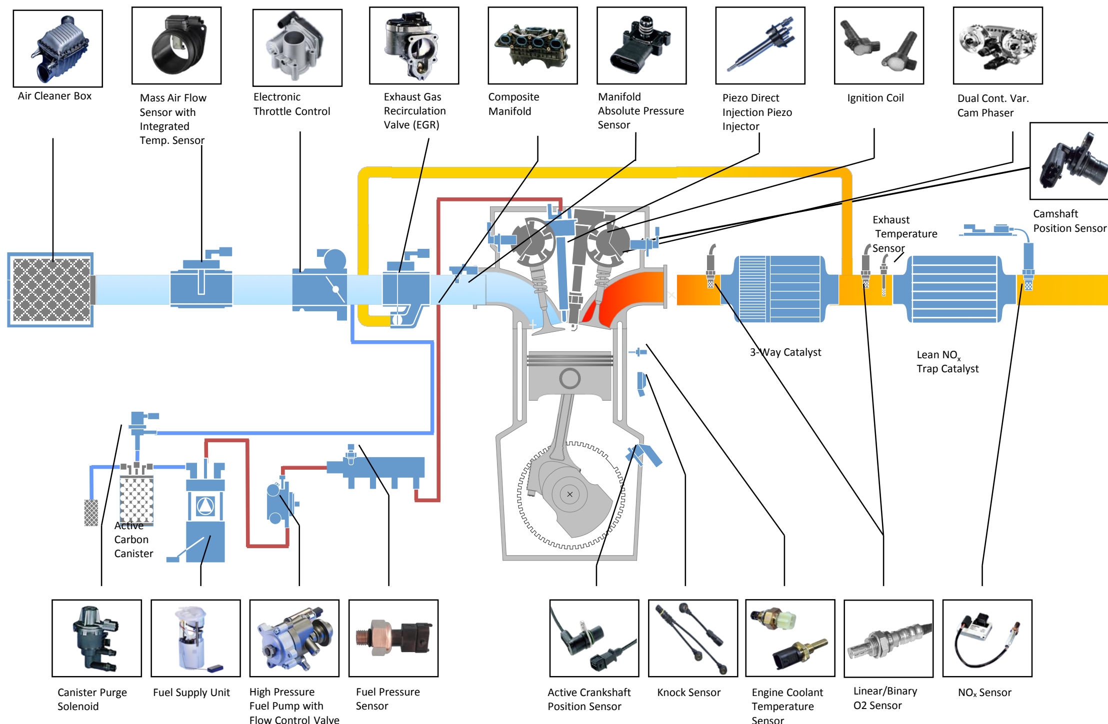

Figure 10.1: Sensors and actors of an automotive powertrain system (By courtesy of Continental Automotive GmbH).

The engine management unit receives beside the angle of the fuel pedal a lot of additional data e.g. from the mass air flow sensor, NO_x sensor, or fuel pressure sensor. This data is processed by algorithms to determine the firing angle and the amount of air in the cylinder, which allows to make the combustion process more efficient in order to fulfill exhaust emission norms and reduce fuel consumption, but also to provide the demanded turning moment. These calculations have to be partitioned on tasks with different temporal requirements, dependent on the required updating rate. Otherwise it is not possible to process the algorithms and trigger the injection with minimal interarrival times of up to 1.2 ms.

In the automotive domain, software or software parts are typically used in different projects of the same type. However, through modifications of hardware, peripherals, or functional amount, the software differs between these projects. The objective of the following case studies is to evaluate scheduling algorithms for these automotive systems by use of a probabilistic system model.

The task set data, used to generate the probabilistic system model of the case studies, was provided from the Continental Automotive GmbH. Task execution times were determined by profiling methods and code analysis approaches. Task deadlines and the inter-arrival times of periodic tasks and engine speed dependent inter-arrival times are used from design documents.

10.2 Case Study I: A Quadcore System

In this case study, a probabilistic task set description, representing different automotive systems, is used to compare the scheduling algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2}.

Since in this case study also the number of tasks in a task set changes, the probability of task quantity model from Section 9.2.2 is applied in the following way. The index z denotes a randomly generated task set τ_z from the probabilistic system model S^P.

In the first step, the quantity of tasks n_z of task set τ_z is drawn from a discrete uniform distribution ($n_{\min} = 20, n_{\max} = 30$). Afterwards, for each task $T_i \in \tau$ the inter-arrival time is drawn from an equally distributed list of inter-arrival times $\{2.5, 5.0, 7.5, 10.0, 20.0, 50.0\}$ (these and all further times are in ms) and the task utilization $\text{wt}(T_i)$ is drawn from a Weibull distribution ($\text{wt}_{\min} = 0.05, \overline{\text{wt}} = 0.15, \text{wt}_{\max} = 0.51, p_{\max} = 10\%$), choosen in order to model a generalized function of the minimum, average and maximum task utilization. For all tasks, the deadline is equal to the inter-arrival time. The quantum Q is set to 0.25. For the determination of the number of task sections of a task, execution times are randomly generated from a Weibull distribution ($e_{\min} = 0.125$, $e_{\text{avg}} = 0.24$, $e_{\max} = 0.25$, $p_{\max} = 10\%$) which are assigned to the related task T_i as long as condition

$$\sum_{k=0}^{q} T_i^k.e \leq T_i.p \ \cdot \ \text{wt}(T_i)$$

is fulfilled. The task offset o is drawn from a uniform distribution ($o_{\min} = 0.0, o_{\max} = 0.05$). Finally, each task is assigned to a time base b^v according to a uniform distribution $\{1, 2, 3, 4\}$ with frequency multiplier values $f^v(t)$ in the range $[1, 1.05]$ $\forall v$.

Figure 10.2 shows the result of 500.000 randomly generated and simulated task sets, falling in a system utilization interval $U_{sum}(\tau_z) \in [1.68, 4]$.

One point represents the maximum normed lateness $mNL(\tau_z)$ of a task set τ_z in dependence of the system utilization $U_{sum}(\tau_z)$.

For the algorithm *Partly-Pfair-PD*^{2} the only deadline violation occurs at a system utilization of 3.995. The *mNL* value of this task set is +0.00005, which equates a deadline violation of 0.005 %. The *mNL* value results from a task with 50 *ms* inter-arrival time. At *P-ERfair-PD*^{2}, no deadline violation occurs for any generated and simulated task sets. Additionally, it can be clearly seen that*P-ERfair-PD*^{2} predominates *Partly-Pfair-PD*^{2} up

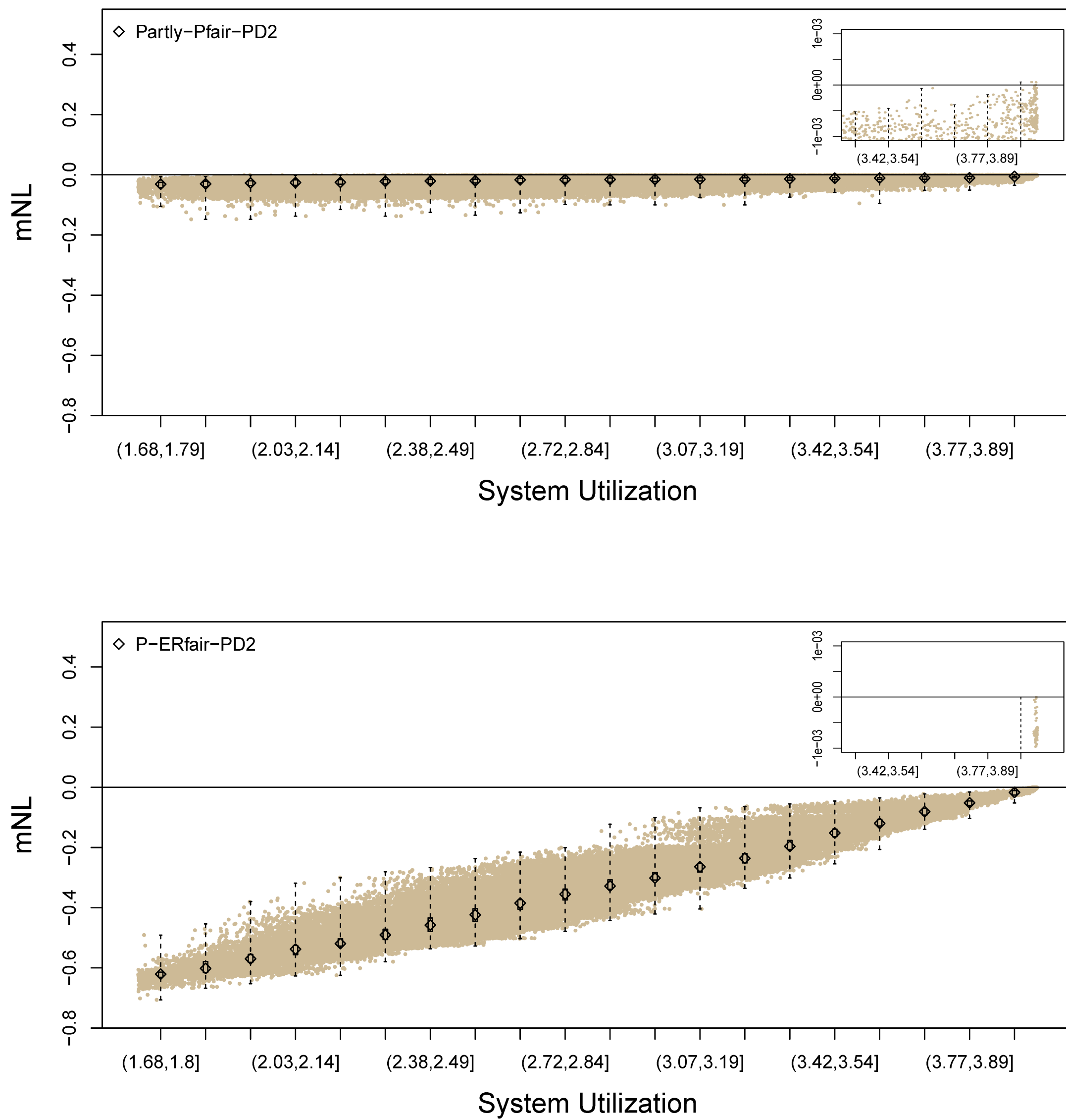

Figure 10.2: Scatter diagrams of *mNL* (y-axis) as function of system utilization (x-axis) for the algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2}. The diagram shows 500.000 systems, generated by a probabilistic automotive system model and simulated by use of computer cloud.

to a utilization of 3.89 and afterwards weakly predominates *Partly-Pfair-PD*^{2}. However *Partly-Pfair-PD*^{2} has a higher predictability than *P-ERfair-PD*^{2}.

10.3 Case Study II: Porting from Singlecore to Dualcore Processors

In this case study, the porting of a number of singlecore processor automotive powertrain systems on a dualcore processor platform is analyzed. The singlecore processor of the original system is replaced by a dualcore processor with equal processing capacity. For comparison reason, the processing frequency of the dualcore processor is the half of the singlecore processor.

The target of the case study is to find the scheduling algorithm which allows the highest system utilization while still fulfilling all deadline requirements. Therefore, the case study compares the scheduling algorithms *Partly-Pfair-PD*^{2}, *P-ERfair-PD*^{2}, and *WFD-EDF* (Worst-Fit-Decreasing).

For automotive powertrain systems, the task execution times differ due to variations in software functionality. Further task set parameters are identical at the different systems due to equal environment conditions. Therefore, the SA-PS approach is applied with the probability of task set parameter model (see Section 9.2.1). The probabilistic system S^P is used to generate task sets $\{\tau_z\}$ with a non probabilistic periodic offset based demand model, defining a minimal inter-arrival time and a task offset. Furthermore, the task set parameters task deadline, number of task sections, and time base reference are constant. These values are listed in Table 10.1.

Tasks are assigned to two types of time bases. The first time base b^1 is the crankshaft trigger which activates tasks in the case of a constant engine rotation speed with different but constant inter-arrival times. However, when rotation speed changes, the inter-arrival time of all task is changed by a common factor. The frequency multiplier $f^1(t)$ models the ratio between maximal rotation speed $r_{\max}$ and actual rotation speed. For this reason, $f^1(t)$ can never be smaller than 1. The lower bound of $f^1(t)$ derives by the maximal rotation speed, namely $\min_t(f^1(t)) = \frac{r_{\max}}{r_{\max}} = 1$. The upper bound of $f^1(t)$ derives by the engine idle speed $r_{\min}$, namely $\max_t(f^1(t)) = \frac{r_{\max}}{r_{\min}} = 6.8$. During the simulation time, $f^1(t)$ is varied between these bounds according to the methodology in section 8.5. The second time base b^2 is a periodic trigger which activates tasks with constant inter-arrival times and task offsets, originating the periodic offset based task set. Therefore, the related time base b^2 has a constant multiplier $f^2(t) = 1$.

The execution time of the task sections is modeled by a Weibull probability distribution function

$$P(T_i^k.e = x_n) = p_n$$

and

$$p_n = wb(x, \lambda, \kappa, e_{\min}) \;\forall T_i^k \in T_i,\; \forall T_i \in \tau_z.$$

All task sections have an identical variation of task section execution time. Using the approach of Section 6.2.2, the Weibull parameter shape κ and scale λ can be calculated from

Table 10.1: Task set parameters of porting case study.

Task	**Inter-arrival time**	**Offset**	**Deadline**	**Number of task-sections**	**Time base**
T00_RPM	2.5	0	2.5	6	b^1
T01_RPM	8.8	2.5	5.0	1	b^1
T02_RPM	2.9	0	1.3	1	b^1
T03_RPM	2.9	0.6	1.3	1	b^1
T04_RPM	2.9	1.2	1.3	1	b^1
T05_RPM	2.9	1.8	1.3	1	b^1
T06_1MS	1.0	0.5	0.6	1	b^2
T07_5MS	5.0	0	5.0	1	b^2
T08_5MS	5.0	7.5	10.0	1	b^2
T09_10MS	10.0	5.0	2.5	3	b^2
T10_10MS	10.0	0	10.0	3	b^2
T11_10MS	10.0	5.0	10.0	2	b^2
T12_20MS	20.0	0	20.0	3	b^2
T13_40MS	40.0	0	80.0	1	b^2
T14_100MS	100.0	0	100.0	10	b^2
T15_1000MS	1000.0	0	500.0	6	b^2

the statistical estimators on execution times: minimal execution time $T_i^k.e_{\min} = 0.01$, average execution time $T_i^k.e_{avg} = 0.22$, and maximal execution time $T_i^k.e_{\max} = 0.3$. The probability of an execution time higher than the maximal is $p_{\max} = 0.0001$. The execution time of the scheduling algorithm ISR, executed at each schedule decision, is considered with $2\mu s$.

From this probabilistic system model, the SA-PS approach generates 25.000 system models for the evaluation and the sensitivity analysis. A simulation duration of 20s was determined empirically to be sufficient for approximation of worst-case response time. In order to compare the algorithms regarding maximal system utilization, the metric *mNL* is determined and analyzed as function of the system utilization $U_{sum}(\tau_z)$. The scatter diagram of the result is shown for all algorithms in Figure 10.3. The $x-$axis represents the system utilization $U_{sum}(\tau_z)$, the $y-$axis represents the *mNL* metric.

The experiment shows, that the algorithm *Partly-Pfair-PD*^{2} has a low variation of *mNL* for each cluster and has the highest predictability of all analyzed algorithms. But, already at a low system utilization, *mNL* exceeds 0 and the system violates task deadlines. This effect occurs because *Partly-Pfair-PD*^{2} works in a non-work-conserving manner. It schedules task instances in a way that they finish short before deadline. However, due to

an execution of the scheduling algorithm ISR at the start of each task section, in some cases the delay of execution results in a violation of task deadlines.

The algorithm *P-ERfair-PD*^{2} shows a high variation of *mNL* per cluster. The median of *mNL* in a cluster increases with the system utilization. Even at high system utilizations there are only a few deadline violations per cluster (see the enlarged detail of *P-ERfair-PD*^{2} in Figure 10.3). The first deadline violation occurs at a system utilization of 1.81 (which corresponds to 91%). These violations result from the additional time for the scheduling algorithm. Migration overhead was not considered in this case study due to the assumptions of no delay for a migration at task section boundary, as argued in section 2.2.2. For processor architectures which don't fulfill this assumption, in [SPDM09] the migration overhead for comparable task sets was estimated by simulation studies and it was derived, that an additional system utilization of $5-10\%$ has to be assumed for this kind of task sets. In comparison with the algorithm *Partly-Pfair-PD*^{2}, *P-ERfair-PD*^{2} has a weak predominance.

The algorithm *WFD-EDF* shows the highest variations of *mNL* and first deadline violations at a system utilization of 1.633. Furthermore, it is obvious that the area divides in two sections (to be discussed below). A disadvantage of the bin-packing approach in general is that there are task sets, which can not be partitioned by a certain heuristic, e.g. WFD, even though there is a valid partition. In this case, further partitioning heuristics have to be applied. Since in each case study the same task sets have been examined by all scheduling algorithms, task sets which could not be partitioned by a heuristic were rejected for all scheduling algorithms.

In Figure 10.4, the range of system utilization $[1.56, 2]$ is divided in 20 equal sized clusters. For each cluster, the confidence bounds of the statistical estimators from Section 9.6 are calculated by the bootstrapping approach with a bootstrap sample size of 500.

The statistical estimators are represented by a box plot. The upper and lower whisker represent the upper and lower confidence interval of the upper and lower quantile respectively, the box represents the confidence interval of the median, and the symbol represents the median itself.

When these statistical estimators are compared for different scheduling algorithms of one cluster, it can be seen that in any case the algorithm *P-ERfair-PD*^{2} has a lower variance than the algorithm *WFD-EDF*. The algorithm *Partly-Pfair-PD*^{2} has the lowest variance of all scheduling algorithms regarding median and range of *mNL* but the algorithm is never better than any other scheduling algorithm. Due to the low variation, *Partly-Pfair-PD*^{2} has a higher predictability of *mNL* than the other algorithms which makes this algorithm appropriate for systems where the task finalization should be independent from other task execution times. However, due to first deadline violation at a low system utilization, this algorithm is not suitable for automotive powertrain systems with hard real-time requirements. Equally, *WFD-EDF* is not applicable because the number of deadline violations increases rapidly for system utilizations higher than 1.633.

Therefore, the only algorithm which fulfills deadline requirements also for a high system utilization is *P-ERfair-PD*^{2}.

A detailed consolidation of the task partitioning of the WFD heuristic shows that all these task sets have a $mNL < 0$ when only models are considered where task *T00_RPM* and *T09_10MS* execute on a different core (see Figure 10.5). Unfortunately, this very good result can not be compared with the experiment results of the scheduling algorithms *P-ERfair-PD*^{2} and *Partly-Pfair-PD*^{2} because of the filtered set of analyzed models.

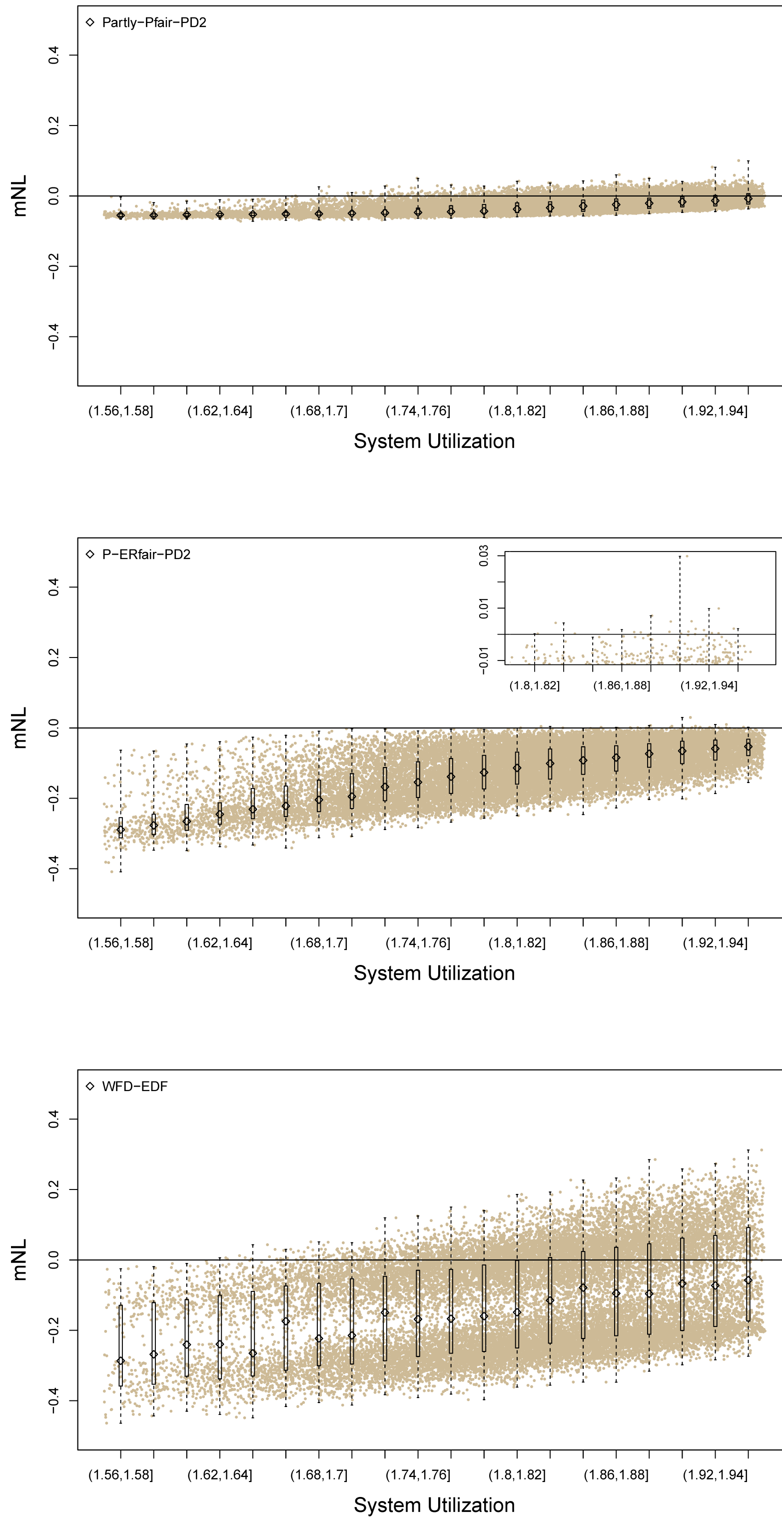

Figure 10.3: Scatter diagrams of *mNL* (y-axis) as a function of the system utilization (x-axis) for the algorithms *Partly-Pfair-PD*^{2}, *P-ERfair-PD*^{2}, and *WFD-EDF*. The diagram shows 25.000 system models originating from a probabilistic automotive powertrain system model.

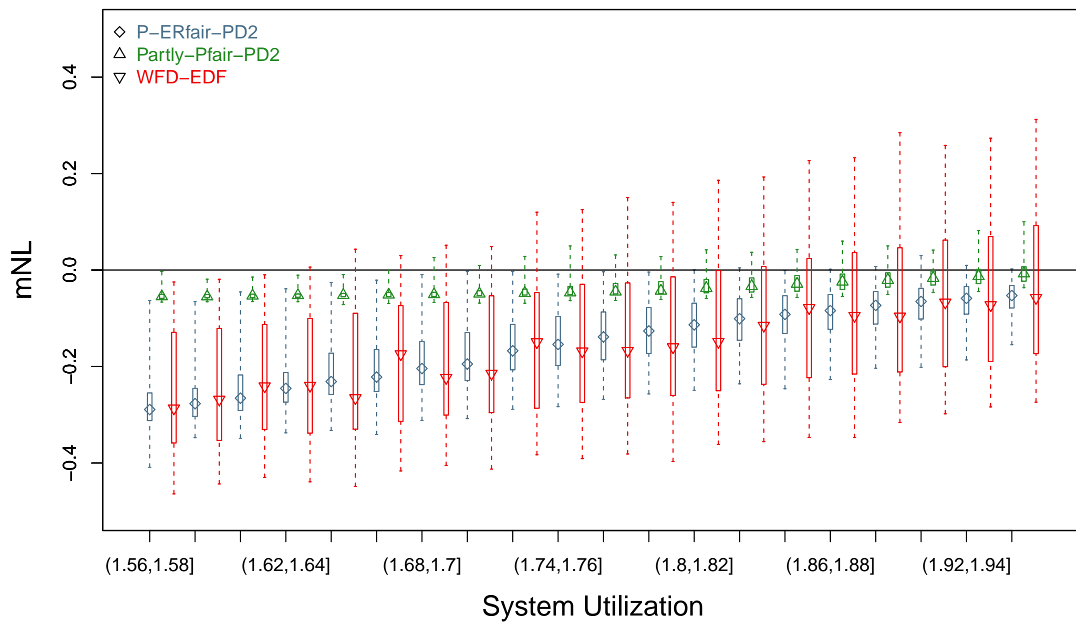

Figure 10.4: Comparison of the scheduling algorithms *Partly-Pfair-PD*^{2}, *P-ERfair-PD*^{2}, and WFD-EDF by statistical estimators on *mNL* (y-axis) as a function of the system utilization (x-axis).

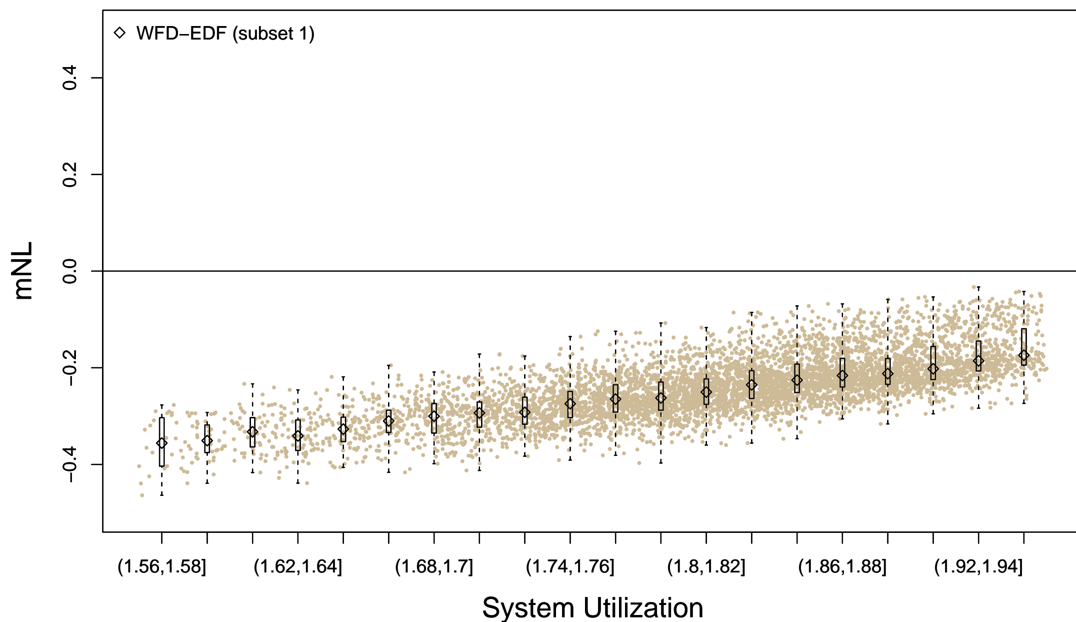

Figure 10.5: Subset of examined models from porting case study (Figure 10.3) and WFD-EDF scheduling, where only models are considered with task *T00_RPM* and *T09_10MS* on different cores.

10.4 Case Study III: Robustness Analysis

This case study analyzes the robustness of the algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2} by inserting temporal perturbations, i.e. execution time changes of task sections. For this case study, the same probabilistic system description as in Section 10.2 is used.

Additionally, each task section is challenged with a random variation of the task section execution time via a Weibull distribution in the range of $T_i^k.e = [0.9 \cdot Q, ..., 2 \cdot Q] \;\; \forall T_i^k \in T_i, \forall T_i \in \tau_z$, with $T_i^k.e \in \mathbb{R}$, and $T_{i,j}^k.e_{\text{avg}} = Q$. Again, Q is set to 250 μs. In real systems, such task execution time variation could occur due to varying runtime of algorithms, preemptions caused by interrupt service routines, different control flows depending on the input, etc.

For the evaluation, 100.000 task sets were generated and simulated.

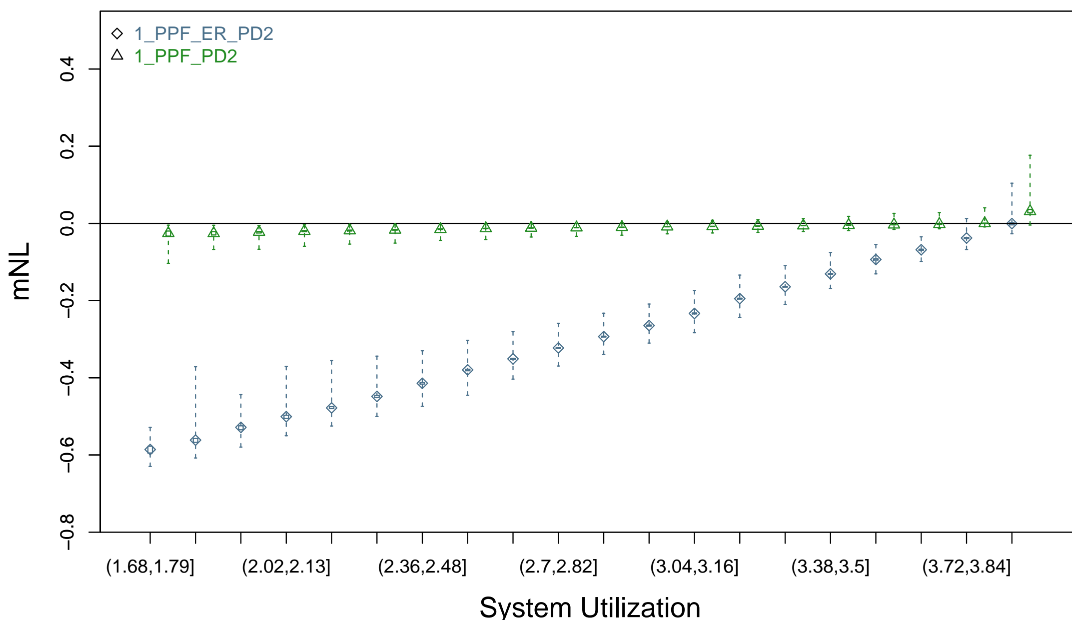

Figure 10.6: Robustness consideration of *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2} through random variation of the task section execution time via a Weibull distribution. The task section execution time can be twice as long as allowed.

Figure 10.6 shows the examination results for both algorithms on a quadcore processor ($m = 4$). Each box plot represents statistical estimators of *mNL* for a group of task sets, clustered by system utilization in 20 equal sized clusters. The box plot of a cluster consists of a symbol, two whiskers, and a box. The symbol represents the median, the box around the symbol represents the bootstrapped 99% confidence intervall of the median, and the whiskers represent the upper and the lower bootstrapped 99% confidence intervall of the inter-quantile distance (99%) boundaries respectively.

Due to the non-work-conserving behavior, *Partly-Pfair-PD*^{2} leads to a *mNL* typically close to zero. For the case of a system without task set perturbations, *Partly-Pfair-PD*^{2} fulfills all deadlines up to a system utilization of 3.995 (which corresponds to $99,9\%$) (see Figure 10.2). However, for the case of a system with perturbations, *Partly-Pfair-PD*^{2} has the first deadline violation at a system utilization of 2.2 (which corresponds to 55%). Furthermore, in contrast to case study I, *Partly-Pfair-PD*^{2} has a lower predictability in the last utilization cluster.

The algorithm *P-ERfair-PD*^{2} is work-conserving, which leads to the fact that tasks typically finish earlier and have lower *mNL* value for low system utilizations. Consequently, the perturbed system is able to meet all deadlines up to a rather high system utilization of 3.7 (which corresponds to 92.5%).
This shows that *P-ERfair-PD*^{2} is able to handle variations of the task section execution time quite well, while *Partly-Pfair-PD*^{2} fails completely above 57.5% (i.e. upper box plot whisker is above $mNL = 0$ limit). *P-ERfair-PD*^{2} strongly predominates *Partly-Pfair-PD*^{2} up to a utilization of 3.72.

Chapter 11

Discussion

This chapter discusses the results of the presented case studies, compares them with other studies, and summarizes the advances in knowledge.

11.1 Local Scheduling with Bin-Packing Partitioning

In case study II, the bin-packing approach WFD was selected for analysis because it allows the highest system utilization of all bin-packing algorithms of Section 3.3.1. This is justified due to a load balancing effect at partitioning, because WFD selects the core with the lowest utilization for the assignment of the next task.

The examined schedulability bound of the bin-packing partitioning algorithm WFD is equal to schedulability bound in [LDG04b]. Carpenter et al. [CFH$^+$04] derived for local scheduling with job-fix priority assignment a system utilization bound of $U_{sum}(\tau_z) = \frac{M+1}{2}$. Lopez et al. [LDG04b] proved for bin-packing algorithms using a decreasing task sorting that the maximal system utilization can be increased when the maximal task utilization $U_{\max}(\tau_z)$ is decreased. Using equation 3.2, $U_{sum}(\tau_z) = 1.63$ is the schedulability bound for task sets with $U_{\max}(\tau_z) = 0.58$ (in Case Study II, $U_{\max}(\tau_z) = 0.58$ derives from task *T00_RPM*). In the case study, the task set with the lowest utilization, which has a $mNL > 0$, has a system utilization of $U_{sum}(\tau_z) = 1.633$. Therefore, the schedulability bound of the case study corresponds with the general schedulability bound of theoretical considerations. However, the case study also shows that many task partitionings produce a much better *mNL* as the schedulability bound. For task sets originating the probabilistic task set description of Case Study II, it was shown that the application of a simple rule, which defines that certain tasks are not allowed to execute on the same core, extends the maximal system utilization bound. Certainly, this rule is only valid for the probabilistic task set description of the automotive powertrain systems.

As expected, in comparison with the global scheduling algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2}, WFD partitioning with EDF scheduling has more deadline violations and never predominates these algorithms. Furthermore, the predictability of *WFD-EDF*

is much lower than at *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2}. According to Question 1 of this work, *WFD-EDF* (and all other bin-packing algorithms) doesn't fulfill the requirement of efficiency, which makes the algorithm inappropriate for embedded multicore systems.

11.2 Global Scheduling Algorithms

The algorithm *Partly-Pfair-PD*^{2} works in a non-work-conserving manner, meaning the processor may be idle even though tasks are ready to execute. The algorithm *P-ERfair-PD*^{2} works in a work-conserving manner and executes ready tasks whenever there are free processors. The experiments show that the non-work-conserving behavior of *Partly-Pfair-PD*^{2} leads to a lower variability of maximal task response times at changing task set parameters but similar system utilization, because the variation of mNL in a system utilization cluster is very low in comparison with the algorithm *P-ERfair-PD*^{2}. This is particularly beneficial for soft real-time systems which require a production of calculation results in a periodic manner, e.g. at control of robotic actuators by feedback controlled systems or at multimedia applications with a constant sending interval for video frames.

Devi and Anderson [DA05] showed that the original algorithm *Pfair-PD*^{2} leads to deadline violations of at most one time quantum Q, when using desynchronized scheduling calls. Desynchronized scheduling calls means that the scheduler is called and is allowed to assign a waiting time quanta to a core, when the execution time of a time quanta is smaller than Q. Case Study I showed for task sets, common in automotive systems, the modified algorithm *Partly-Pfair-PD*^{2} has deadline violations only at maximal system utilization and only for the tasks in the task set, which have the highest inter-arrival time (or rather task deadline). Also when a direct comparison is not possible because the algorithms are proposed for different kind of task sets, it is observed that the algorithm *Partly-Pfair-PD*^{2} performs for the analyzed task sets better than the original *Pfair-PD*^{2} algorithm for sporadic task sets. According predictability, *Partly-Pfair-PD*^{2} outperforms both algorithms, *P-ERfair-PD*^{2} and *WFD-EDF*. Therefore *Partly-Pfair-PD*^{2} is appropriate for efficient soft real-time systems, which furthermore require a high steadiness of response times at task set perturbations.

The benefit of the work-conserving behavior of *P-ERfair-PD*^{2} is a higher robustness against task execution time perturbations. Immediately execution of ready task results in lower response times, i.e. higher laxities at task termination, which allows the scheduler to compensate the higher execution times of the task sections. Furthermore, global scheduling is able to migrate waiting task sections to another core when a core is blocked by another task. In each case study *P-ERfair-PD*^{2} strongly or at least weakly predominates *Partly-Pfair-PD*^{2}. This qualifies *P-ERfair-PD*^{2} for highly efficient hard real-time systems with a high robustness of deadline compliance, as long as the predictability of

response times is not significant.

11.3 Simulation-Based Multicore Real-Time Examination

The simulation-based examination approach was used for schedulability considerations because mentioned examination approaches from Section 4.2 are not able to analyze global multicore scheduling algorithms with a task section fix priority assignment for task sets with inter-arrival patterns which differ from periodic activation.

For the case of local scheduling algorithms with a job-fix priority assignment, Case Study II shows that the schedulability bound of the simulation-based examination approach corresponds to the proved schedulability bounds. This indicates that the mechanisms of the simulation for approximation of worst-case response times are adequate for the examination of local scheduling algorithms.

When task set parameter fulfill the restrictions of synchronized periodic task sets with implicit deadline and quantized execution time, the algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2} behave like the original algorithms *Pfair-PD*^{2} and *P-ERfair-PD*^{2} (see Equation 7.3). In this case, the determined response time corresponds to the maximal value which was proved by Baruah et al. [BCPV96] and Anderson [AS00a] as schedulability bound. Therefore, also for global scheduling algorithms, the simulation-based examination approach adequately approximates the worst-case response time. According to Question 2 of this work, the simulation-based real-time examination approach can be stated as appropriate for the examination of response times of the presented algorithms.

Part IV

Conclusion and Future Work

Chapter 12

Summary

This work has investigated real-time scheduling of practical task sets on embedded multicore processor systems as well as the schedulability examination of these systems.

The presented global scheduling algorithm *Partly-Pfair-PD*^{2} works in a non-work-conserving manner. For practical task sets with multiple time base extension, defining clock dependencies for activation pattern of tasks with variable execution times and cooperative scheduling points, *Partly-Pfair-PD*^{2} has the benefit of a high predictability of task response times and a moderate violation of task deadlines, also at high task set utilizations. Therefore, this algorithm is particularly suitable for a soft real-time system, e.g. embedded systems with multimedia function, where tasks have to finish execution regularly but moderate deadline violations are permitted [But05c].

The introduced work-conserving variant, denoted as *P-ERfair-PD*^{2}, turned out to be very efficient regarding the maximal system utilization because it allows to utilize the complete processor capacity and fulfills all task deadlines in several case studies. Furthermore, *P-ERfair-PD*^{2} shows a high robustness regarding task set parameter perturbation, e.g. in a case study, the algorithm was able to schedule task sets on a quadcore processor up to 93% of task set utilization despite of perturbations provoking up to the double of task execution times. Therefore, this algorithm is especially interesting for high performance and hard real-time systems, e.g. automotive powertrain systems.

For the group of global scheduling algorithms with dynamic priority assignment, only algorithm specific proofs of schedulability were introduced up to now. The introduced simulation-based schedulability examination methodology does not prove compliance of task deadlines, but examines schedulability by an approximation of response-times. This approach can be used for many kinds of scheduling algorithm, e.g. local static priority scheduling algorithms, cooperative scheduling algorithms, and global dynamic priority scheduling algorithms. Furthermore, this approach also allows to consider scheduling overheads.

For a sensitivity analysis of the effect of changing task set parameters on schedulability, the SA-PS methodology was introduced. Using a probabilistic system model, the

SA-PS approach allows to determine precisely whether a system parameter effects any metric of system. Furthermore, this approach can be used for comparison of scheduling algorithms.

Several case studies, located in the automotive domain, evaluated the applicability of the presented algorithms and approaches.

Chapter 13

Further Work

In this work, scheduling algorithms for independent tasks and sequentially dependent task sections have been proposed and evaluated. However, through the introduced task set model, task response times can not be reduced. This is important when the processor frequency remains constant in next generation processor architectures. With the advent of highly parallelized systems, like many core processor systems, identifying independent sections in a task and the parallel execution of these sections could overcome this drawback. At certain positions during task execution, a task instance is parallelized for execution on multiple cores and results return afterwards, also known as fork-join mechanisms. For this kind of task sets, additional temporal requirements have to be assigned to parallelized task sections in order to complete these task sections before task deadline is exceeded. Scheduling of those task sets and the problem of a high sorting time for the long queue of waiting task sections for a many core processor system remains for further work.

The SA-PS approach allows a sensitivity analysis of probabilistic system models. For this purpose, the deadline compliance was analyzed as a function of system model characteristics. However, through this reduction of system model characteristics to one dimension, a lot of information is neglected. By application of clustering approaches, groups of system model parameters could be extracted from the SA-PS approach which influence the deadline compliance. Employing this information in the development process of embedded systems could lead to a higher quality of deadline compliance.

Bibliography

[ABD08] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208, 2008.

[ABR+93] N. Audsley, A. Burns, M. F. Richardson, K.W. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[ABRW91] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: the deadline-monotonic approach. In *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[ABS06] K. Albers, F. Bodmann, and F. Slomka. Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 97–106, 2006.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[ALM10] K. Altisen, Y. Liu, and M. Moy. Performance Evaluation of Components Using a Granularity-based Interface Between Real-Time Calculus and Timed Automata. *Electronic Proceedings in Theoretical Computer Science*, 28:16–33, 2010.

[AS00a] J. H. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems 2000*, pages 35–43, 2000.

[AS00b] J. H. Anderson and A. Srinivasan. Pfair Scheduling: Beyond Periodic Task Systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, 2000.

[AS01] J. H. Anderson and A. Srinivasan. Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 76–85, 2001.

[AUT10] AUTOSAR. AUTomotive Open System ARchitecture Release 4.0, 2010.

[Bak07] T.P. Baker. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*, pages 1–28. Chapman & Hall/CRC, Boca Raton, 2007.

[Bar98] P. Barford. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the ACM SIGMETRICS*, pages 151–160, June 1998.

[Bar05] S. K. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 137–144, 2005.

[Bar07] S. K. Baruah. Techniques for Multiprocessor Global Schedulability Analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128, December 2007.

[BB03] E. Bini and G. C. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, July 2003.

[BB06] S. K. Baruah and A. Burns. Sustainable Scheduling Analysis. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 159–168, 2006.

[BBM+10] M. Bertogna, G. C. Buttazzo, M. Marinoni, G. Yao, F. Esposito, S. Superiore, S. Anna, and M. Caccamo. Preemption points placement for sporadic task sets. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 251–260, 2010.

[BCL09] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems on Parallel and Distributed Systems*, 20(4):553–566, 2009.

[BCOQ92] F. L. Baccelli, G. Cohen, G. J. Olsder, and J. P. Quadrat. *Synchronization and linearity: An Algebra for Discrete Event Systems*. Wiley, New York, 1992.

[BCPV96] S. K. Baruah, N. Cohen, G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[Ber07] M. Bertogna. *Real-time scheduling analysis for multiprocessor platforms.* PhD thesis, Universtiy Scuola Superiore Sant'Anna, Pisa, 2007.

[BG04] S. K. Baruah and J. Goossens. Scheduling Real-Time Tasks: Algorithms and Complexity. In *Handbook of Scheduling*. Chapman & Hall/CRC, Boca Raton, 2004.

[BGP95] S. K. Baruah, J.E. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of 9th International Parallel Processing Symposium*, pages 280–288, 1995.

[BGST04] J. Beirlant, Y. Goegebeur, J. Segers, and J. Teugels. *Statistics of extremes: theory and applications.* Wiley series in probability and statistics. Wiley, Weinheim, 2004.

[BLV09] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems*, 42(1-3):63–119, 2009.

[BMR90] S. K. Baruah, A. K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[BRH90] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, November 1990.

[Bur95] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. pages 225–248. Prentice Hall, New Jersey, 1995.

[But05a] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer, New York, 2005.

[But05b] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, 29(1):5–26, 2005.

[But05c] G.C. Buttazzo. *Soft real-time systems: predictability vs. efficiency*. Springer, New York, 2005.

[BW97] A. Burns and A. J. Wellings. Restricted tasking models. *ACM SIGAda Ada Letters*, 17(5):27–32, 1997.

[BW09] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley, Reading Massachusetts, 2009.

[CD73] E. G. Coffman and P. J. Denning. *Operating systems theory.* Prentice Hall, Upper Saddle River, New Jersey, 1973.

[CEBA02] A. Cervin, J. Eker, B. Bernhardsson, and E. Arzen. Feedback-Feedforward Scheduling of Control Tasks. *Real-Time Systems*, 23(1):25–53, 2002.

[CFH+04] J. Carpenter, J. H. Funk, P. L. Holman, A. Srinivasan, J. H. Anderson, and S. K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook of Scheduling*, pages 30.1–30.19. Chapman & Hall/CRC, Boca Raton, 2004.

[CG06] L. Cucu and J. Goossens. Feasibility Intervals for Fixed-Priority Real-Time Scheduling on Uniform Multiprocessors. In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, pages 397–404, 2006.

[CJGJ78] E.G. Coffman Jr, M.R. Garey, and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[CKT03] S. Chakraborty, S. Kuenzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the Design, Automation, and Test in Europe Conference*, pages 190–195, 2003.

[Cor94] J. C. Corbett. Modeling and analysis of real-time Ada tasking programs. In *Proceedings Real-Time Systems Symposium*, pages 132–141, 1994.

[CRJ06] H. Cho, B. Ravindran, and E. Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 101–110, 2006.

[Cru02] R. L. Cruz. A Calculus for Network Delay, Part I: Network Elements in Isolation. *IEEE Transactions On Information Technology*, 37(1):114–131, 2002.

[DA05] U. C. Devi and J. H. Anderson. Desynchronized pfair scheduling on multiprocessors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[Das10] A. Dasgupta. *Fundamentals of Probability: A First Course.* Springer, New York, 2010.

[DB05] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proccedings of the 26th IEEE Real-Time Systems Symposium*, pages 389–398, 2005.

[DB09] R. I. Davis and A. Burns. A Survey of Hard Real-Time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems. *Technical Report RTS Group, University of York*, 2009.

[Der74] M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74:807–813, 1974.

[Deu08] M. Deubzer. Method of Performance Analysis for Embedded System Architectures. Master's thesis, University of Applied Sciences Regensburg, 2008.

[DH01] L. De Alfaro and T. Henzinger. Interface theories for component-based design. In *Proceedings of the Embedded Software Conference*, pages 148–165, 2001.

[Dha77] S. K. Dhall. *Scheduling periodic-time-critical jobs on single processor and multiprocessor computing systems.* PhD thesis, University of Illinois at Urbana-Champaign, 1977.

[DHM+10] M. Deubzer, M. Hobelsberger, J. Mottok, F. Schiller, R. Dumke, M. Siegle, U. Margull, M. Niemetz, and G. Wirrer. Modeling and Simulation of Embedded Real-Time Multi-Core Systems. In *Proceedings of the 3rd Embedded Software Engineering Congress*, pages 228–241, 2010.

[DL78] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[DM89] M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software*, 15(12):1497–1506, 1989.

[DM10] M. Deubzer and J. Mottok. Dependability von Systemen mit dynamischen Multicore-Schedulingalgorithmen. In *safetronic.2010*, November 2010.

[DMM+10] M. Deubzer, U. Margull, J. Mottok, M. Niemetz, and G. Wirrer. Partly Proportionate Fair Multiprocessor Scheduling of Heterogeneous Task Systems. In *Proceedings 5th Embedded Real Time Software and Systems Conference*, 2010.

[DMMN10] M. Deubzer, J. Mottok, U. Margull, and M. Niemetz. Efficient Scheduling of Reliable Automotive Multi-core Systems with PD2 by Weakening ERfair Task System Requirements. In *Proceedings of the Automotive Safety and Security*, pages 60–67, 2010.

[DRRG10] F. Dorin, P. Richard, M. Richard, and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46(3):305–331, 2010.

[DRS06] S.B. Dhia, M. Ramdani, and É. Sicard. *Electromagnetic compatibility of integrated circuits: techniques for low emission and susceptibility.* Springer, New York, 2006.

[DSM+10a] M. Deubzer, F. Schiller, J. Mottok, M. Niemetz, and U. Margull. Effizientes Multicore-Scheduling in Eingebetteten Systemen - Teil 1: Algorithmen für zuverlässige Echtzeitsysteme. *atp, Automatisierungstechnische Praxis*, 52(9):60–67, 2010.

[DSM+10b] M. Deubzer, F. Schiller, J. Mottok, M. Niemetz, and U. Margull. Effizientes Multicore-Scheduling in Eingebetteten Systemen - Teil 2: Ein simulationsbasierter Ansatz zum Vergleich von Scheduling-Algorithmen. *atp, Automatisierungstechnische Praxis*, 52(10):54–63, 2010.

[DTB93] R. I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 222–231, 1993.

[ET93] B. Efron and R.J. Tibshirani. *An introduction to the bootstrap.* Chapman & Hall/CRC, Boca Raton, 1993.

[FGB10] N. W. Fisher, J. Goossens, and S. K. Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71, 2010.

[FN09] J. H. Funk and V. Nadadur. LRE-TL : An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets. In *Proceedings of the 17th International Conference on Real-Time and Network Systems*, pages 26–27, 2009.

[Fre09] Freescale Semiconductors. Embedded Multicore: An Introduction, Whitepaper, 2009.

[FTF+02] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[GFB03] J. Goossens, J. H. Funk, and S. K. Baruah. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Real-Time Systems*, 25(2):187–205, 2003.

[GGD+07] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. In *Proceedings of the 5th International Conference on Software Technologies for Embedded and Ubiquitous Systems*, number 2006, pages 263–272, 2007.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability.* Freeman, San Francisco, 1979.

[GMM99] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.

[GNU10] GNU Scientific Library. http://www.gnu.org/software/gsl/, December 2010.

[Gra71] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. *Proceedings of the AFIPS Joint Computer Conferences*, pages 205–217, 1971.

[Gum04] E.J. Gumbel. *Statistics of extremes.* Dover Publications, Mineola, New York, 2004.

[HHJ+04] A. Hamann, R. Henia, M. Jersak, R. Racu, K. Richter, and R. Ernst. SymTA/S-Symbolic Timing Analysis for Systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 17–20, 2004.

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis-the SymTA/S approach. *IEE-Proceedings Computers and Digital Techniques*, 152(2):148–166, 2005.

[HV06] M. Hendriks and M. Verhoef. Timed Automata Based Analysis of Embedded System Architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2006.

[Int05] International Organizations for Standardization. ISO 17356-3 Road vehicles - Open interface for embedded automotive applications - Part 3: OSEK/VDX Operating System (OS), 2005.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[KM91] T. W. Kuo and A. K. Mok. Load adjustment in adaptive real-time systems. *Proceedings of the 12th Real-Time Systems Symposium*, pages 160–170, 1991.

[Kop08] H. Kopetz. Wrong Assumptions and Neglected Areas in Embedded Systems Research. In *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, page 360, May 2008.

[Kue06] S. Kuenzli. *Efficient Design Space Exploration for Embedded Systems.* PhD thesis, Swiss Federal Institute of Technology, Zurich, 2006.

[KW95] K. Knopp and W. Walter. *Theorie und Anwendung der unendlichen Reihen.* Springer, Berlin, 1995.

[LA10] C. Liu and J.H. Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32, 2010.

[LBT01] J. Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet.* Springer, Berlin, 2001.

[LCA09] H. Leontyev, S. Chakraborty, and J. H. Anderson. Multiprocessor extensions to real-time calculus. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 410–421, 2009.

[LDG04a] J. M. López, J. L. Díaz, and D. F. García. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *IEEE Transactions on Parallel and*, 15(7):642–653, 2004.

[LDG04b] J. M. Lopez, J. L. Díaz, and D. F. García. Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems*, 28(1):39–68, 2004.

[Leh90] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, 1990.

[LGDG00] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. García. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 25–33, 2000.

[LH94] J. W. S. Liu and R. Ha. Efficient Mehtods of Validating timing constraints in multiprocessor and distributed real-time systems. In *Proccedings of the 14th International Conference on Distributed Computing Systems*, pages 162–171, 1994.

[LHS+98] C. Lee, J. Hahn, Y. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[Liu69] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, 37:60, 1969.

[LL73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LLL+98] S. Lee, C. Lee, M. Lee, S. L. Min, and C. S. Kim. Limited preemptible scheduling to embrace cache memory in real-time systems. In *Proceedings of the Languages, Compilers, and Tools for Embedded Systems Conference*, pages 51–64, 1998.

[LNKN10] Y. Lu, T. Nolte, J. Kraft, and C. Norstrom. A statistical approach to response-time analysis of complex embedded real-time systems. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 153–160, 2010.

[LNR99] G. Lechner, H. Naunheimer, and J. Ryborz. *Automotive transmissions: fundamentals, selection, design and application.* Springer, Berlin, 1999.

[LS02] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, 2002.

[LW82] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[Mar96] G. Marsaglia. DIEHARD: a battery of tests of randomness. *See http://stat. fsu. edu/∼ geo/diehard. html*, 1996.

[May09] D. May. The XMOS XS1 Architecture. *XMOS Ltd*, 2009.

[MHK+08] J. Madsen, M. R. Hansen, K. S. Knudsen, J. E. Nielsen, and A. W. Brekling. System-level verification of multi-core embedded systems using timed-automata. In *Proceedings of the 17th World Congress International Federation of Automatic Control*, pages 9302–9307, 2008.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[MNW10] U. Margull, M. Niemetz, and G. Wirrer. Quirks and Challenges in the Design and Verification of Efficient, High-Load Real-Time Software Systems. In *5th Embedded Real Time Software and Systems Conference*, 2010.

[Mok83] A. K. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment.* PhD thesis, Massachusetts Institute of Technology, 1983.

[MTN08] Jukka Mäki-Turja and Mikael Nolin. Efficient implementation of tight response-times for tasks with offsets. *Real-Time Systems*, 40(1):77–116, 2008.

[MU49] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[Ope10] Open System C Initiative. http://www.systemc.org, December 2010.

[PH98] J. C. Palencia and G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 26–37. Citeseer, 1998.

[PLW90] M. Pecht, P. Lall, and S. J. Whelan. Temperature dependence of microelectronic device failures. *Quality and Reliability Engineering*, 6:275–284, 1990.

[PS01] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *ACM SIGOPS Operating Systems Review*, 35(5):89, 2001.

[PTCT07] L. T. X. Phan, L. Thiele, S. Chakraborty, and P. S. Thiagarajan. Composing Functional and State-based Performance Models for Analyzing Heterogeneous Real-Time Systems Systems. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 343–352, 2007.

[PWTH09] S. Perathoner, E. Wandeler, L. Thiele, and A. Hamann. Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Design Automation for Embedded Systems*, 13(1):27–49, 2009.

[RE02] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proceedings of the conference on design, automation and test in Europe*, pages 506–513, 2002.

[RJE03] K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *IEEE Computer*, 36(4):60–67, 2003.

[RK08] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo method.* Wiley, Hoboken, 2008.

[RM06] H. Ramaprasad and F. Mueller. Tightening the Bounds on Feasible Preemption Points. In *Proceedings 27th IEEE International Real-Time Systems Symposium*, pages 212–224, December 2006.

[RRE03] K. Richter, R. Racu, and R. Ernst. Scheduling analysis integration for heterogeneous multiprocessor SoC. In *Proceedings of the 24th International Real-Time Systems Symposium*, pages 236–245, 2003.

[RWTW06] J. Reineke, B. Wachter, S. Thesing, and R. Wilhelm. A definition and classification of timing anomalies. In *Proceedings 6th International Workshop on Worst-Case Execution Time Analysis*, pages 1–6, 2006.

[RZJE02] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceedings 39th Annual Design Automation Conference*, pages 287–292, 2002.

[SAC+04] L. Sha, T. Abdelzaher, A. Cervin, T. P. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory : A Historical Perspective. *Real-Time Systems*, 28:101–155, 2004.

[SB96] M. Spuri and G. C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*, 10(2):179–210, 1996.

[SD88] B. Schmeiser and L. Devroye. Non-Uniform Random Variate Generation. *Journal of the American Statistical Association*, 83(403):906–945, 1988.

[SPDM09] G. Stamatescu, D. Popescu, M. Deubzer, and J. Mottok. Migration Overhead in Multiprocessor Scheduling. *Proceedings of the 2nd Embedded Software Engineering Congress*, pages 645–653, 2009.

[SREP08] S. Samii, S. Rafiliu, P. Eles, and Z. Peng. A simulation methodology for worst-case response time estimation of distributed real-time systems. In *Proceedings of the Design, Automation and Test in Europe Conference 2008*, pages 556–561, 2008.

[SSE08] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 41–48, 2008.

[Sta98] J. A. Stankovic. *Deadline scheduling for real-time systems: EDF and related algorithms*. Kluwer Academic Publisher, Norwell, 1998.

[TCGK02] L. Thiele, S. Chakraborty, M. Gries, and S. Kuenzli. *Design space exploration of network processor architectures*, volume 1, pages 55–89. 2002.

[Tin92] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical report, Technical Report RTS Group, University of York, 1992.

[VDI96] VDI Richtlinie 3633. Simulation von Logistik-, Materialfluß- und Produktionssystemen - Begriffsdefinitionen, 1996.

[WMM+08] R. Wilhelm, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, and R. Heckmann. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[YBB09] G. Yao, G. C. Buttazzo, and M. Bertogna. Bounding the Maximum Length of Non-Preemptive Regions Under Fixed Priority Scheduling. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 351–360, 2009.

[YBB10] G. Yao, G. C. Buttazzo, and M. Bertogna. Comparative evaluation of limited preemptive methods. In *Proceedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation*, 2010.

[ZBB09] F. Zhang, A. Burns, and S. Baruah. Sensitivity Analysis for Real-Time Systems. Technical report, University of York, Department of Computer Science, Technical Report YCS-2009-438, 2009.

[ZBB10a] F. Zhang, A. Burns, and S. Baruah. Sensitivity analysis of relative deadline for EDF scheduled real-time systems. In *Proceedings of the 2nd International Conference on Mechanical and Electronics Engineering*, 2010.

[ZBB10b] F. Zhang, A. Burns, and S. Baruah. Sensitivity analysis of task period for EDF scheduled arbitrary deadline real-time systems. In *Proceedings on the 3rd IEEE International Conference on Computer Science and Information Technology*, pages 23–28, 2010.

[ZMM03] D. Zhu, D. Mosse, and R. Melhem. Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary? In *Proceedings of the 24th IEEE Real-Time Systems Symposium, 2003*, pages 142–151, 2003.

Index

Activation
    Asynchronous, 24
    Synchronous, 24
Architectural Model, 84

Behavioral Model, 84

Cascade, 50
Constrained System Density, 29
Cooperative Scheduling, 39
Critical Instant, 61

Deadline
    Arbitrary, 28
    Explicit, 28
    Implicit, 28
Deferred Preemption, 39
Discrete Distribution, 74
Discrete Event Simulation, 84
Dispatcher, 31
DVFS, 16

Efficiency, 1
Embedded System, 9
Embedded Systems, 9

Feasibility, 52, 53
Finalization Time, 55

Generalized System Density, 29
Global Scheduling, 19
Group-Deadline, 49

Harmonic Base, 37

Instructions Per Tick Parameter, 102
Interruption, 33

Lateness, 55
Laxity, 39
Local Scheduling, 19

Maximal Normed Lateness, 107
Maximal Task Utilization, 29
Maximal Task Weight, 29
Migration
    Bounded, 34
    Full, 34
    No, 34
    Task Section, 34
Monte-Carlo Randomization, 113
Multicore System, 16
Multiple Time Base Extension, 69

Non-Sustainability, 60
Non-work-conserving, 35

Offset, 24
Overlapping-Bit, 48

Partitioned Scheduling, 19
Performance, 64
Pfair Scheduling, 46
Phasing, 67
Policies, 31
Predictability, 60, 118
Predominance, 118
Preemption, 33
Priority
    Dynamic, 34
    Job-fix, 34
    Section-fix, 34
    Static, 34

- Task-fix, 34
- Processor, 16
- Pseudo-Activation, 47
- Pseudo-Deadline, 47
- Pseudo-Release, 47

- Rank, 31
- Real-time, 1
- Real-Time Calculus, 58
- Real-Time System Simulation, 84
- Response Time, 55
- Robustness, 1
- Runnable, 15

- SA-PS, 111
- Schedulability, 52, 53
- Schedulability Analysis, 107
- Schedulability Bound, 32
- Schedulability Examination, 54, 107
- Schedule, 31
- Scheduler, 19
- Scheduling
  - Clustered, 33
  - Cooperative, 34
  - Global, 33
  - Local, 33
  - Preemptive, 34
- Scheduling Algorithms
  - DM, 37
  - EDF, 38
  - LLF, 39
  - LLREF, 45
  - Pfair
    - BF, 51
    - PD, 49
    - PD^2, 49
    - PF, 49
  - RM, 37
- Singlecore System, 16
- Sustainability, 60
- Symmetric Multicore Processor, 4
- Symmetric Multiprocessing, SMP, 16
- System Utilization, 29

- Task, 13
- Task Section, 13
- Task Set, 12
- Task Utilization, 29
- Task Weight, 29
- Temporal Requirements, 10
- Temporal Robustness, 64

- Utilization, 64

- Weight, 29
- Work-Conserving, 34

List of Figures

2.1 Graphical definition of a real-time system. 11

2.2 A task T_i has temporal properties, including inter-arrival time p_i, execution time e_i, and deadline d_i, and logical properties, including input signals $\Phi^I(i)$ and output signals $\Phi^O(i)$. 12

2.3 Graphical definition of a task section. 14

2.4 Example of a task with task sections. 15

2.5 Memory architecture of a multicore processor. 18

2.6 Signal storage and distribution in the task section based communication approach. A task stores signals during calculation in the local cache (C), which is synchronized with the private memory (PM) (Step 1). Collectively shared signals can be loaded from the shared memory (SM) (Step 2). A synchronization routine (sync) gathers all signals from the private memories and provides them to the shared memory (Step 3). Afterwards, the actual signal value can be loaded from the shared memory (Step 4). . 19

2.7 Notations of timestamps during the execution of job $T_{i,j}$ of task T_i. 20

2.8 Example of a recurrent task activation. The input interface signal $\Phi^I_{IF}(t)$ is sampled and a task activation is triggered in a periodic manner with an inter-arrival time p_i. 21

2.9 Example of an arrival curve task activation. A change in the interface signal $\Phi^I_{IF}(t)$ is immediately processed by an activated task. The upper and lower number of activations is expressed by an upper α^u and lower α^l arrival curve respectively as a function of any time interval with the size Δ. 22

2.10 Example of a hierarchical task activation. A task T_a (activated in a periodic manner with inter-arrival time p_a) activates another task T_b at end of execution or at any other point during execution. The activation time of T_b depends on the delay of task T_a. 23

2.11 Arrival curve of periodic task activations with $T_i.p = 7$. The upper α^u_i and lower α^l_i number of task activations are specified in dependence of the size of any time interval Δ. 26

2.12 Comparison of WCET and BCET bounds, detected by different approaches [WMM+08]. The maximum of the measured execution time approximates the exact WCET from the left side by systematically executing measurement scenarios. Analytically determined maximum execution time approximates the exact WCET from the right side by reducing the pessimism of the prediction. The BCET determination works analogous. 28

3.1 Arrival of two periodic task (upper curves) and related fluid scheduling graph (lower curve). 44
3.2 LLREF: Determination of T-L plane [CRJ06]. 45
3.3 LLREF example of token movement through job execution in a T-L plane [CRJ06]. 46
3.4 Approximation of *Pfair* to the fluid schedule. The fluid schedule defines that the execution time $T_1.e = 3$ of a task is allocated to a processor between two successive activations (arrows, $T_1.p = 5$) in a way that the remaining execution time constantly decreases (dotted line). *Pfair* defines a minimum and maximum lag (dashed line) from the fluid schedule in continuous time. In discretized time, task quanta have to execute in windows (continuous line, $\Upsilon_1^1, \Upsilon_1^2, \Upsilon_1^3$), derived from lag boundary. 48
3.5 Example of a schedule of algorithm Pfair-PD^2. The x-axis represents the time and the y-axis defines the processed execution time. Task quanta (grey) are processed between the upper and lower lag boundary (slim line) which results from the fluid schedule graph (thick line). 50

4.1 Examination methodologies of worst-case response time. The x-axis represents the number of iterations (for iterative approaches) or the calculation time (e.g. for simulation approaches) and the y-axis represents the estimation of the quality criteria. A pessimistic approach approximates the worst-case response time from the upper side (overestimated) and the optimistic approach from the lower side (underestimated). An exact approach determines the worst-case response time directly. 54
4.2 Task job metric response-time, measured between the activation $T_{i,j}.A$ and the finalization $T_{i,j}.F$ of a job $T_{i,j}$. 55
4.3 Task job metric lateness, measured between the deadline $T_{i,j}.D$ and the finalization $T_{i,j}.F$ of a job $T_{i,j}$. 56
4.4 Task job metric End-to-End, measured between the finalization $T_{i,j}.F$ of two subsequent jobs $T_{i,j}$ and $T_{i,j+1}$. 56
4.5 Task job metric Start-to-Start, measured between the start $T_{i,j}.S$ of two subsequent jobs $T_{i,j}$ and $T_{i,j+1}$. 57
4.6 Example 4.1: Multiprocessor Anomalies - sporadic activation. Dualcore processor with three tasks. 61

4.7 Example 4.2: Multiprocessor Anomalies - synchronized periodic activation. Dualcore processor with four tasks. 62

6.1 Example of task activations from different time bases. Tasks are activated in a periodic manner with an inter-arrival time p and an offset o. Task T_1 and T_2 are activated from time base b^1 and task T_3 and T_4 are activated from time base b^2. All task instances execute on the same processor. . . . 68
6.2 Schedule sequence of worst-case response time of task T_4 for different assumed task set models. The sporadic task set model (A) assumes all tasks to be activated simultaneously at critical instant, which results in an overestimated worst-case response time $R_4 = 16$ for task T_4. The periodic task set model with offsets (B) results in an underestimated worst-case response time $R_4 = 10$. Only the consideration of time bases, which defines that the first transactions T_1 and T_2 and the second transaction T_3 and T_4 are able to be shifted against each other, gives the correct worst-case response-time of $R_4 = 11$. 69
6.3 Relation between time t^v of time base b^v and global time t. 70
6.4 Global time t as a function of t^v (upper diagram) and derivative of t over t^v. 71
6.5 Example for different shapes of Weibull density function for equal minimal and maximal value and different average values. 77
6.6 Example for corrected shapes of Weibull density function for equal minimal and maximal value and different average values. 77

7.1 Example of a schedule of algorithm *Partly-Pfair-PD*^{2}. In the upper diagrams, the x-axis represents the time and the y-axis the processed execution time of a task. The lower diagram shows when the processor is in the idle state. 82
7.2 Example of a schedule of algorithm *P-ERfair-PD*^{2}. In the upper diagrams, the x-axis represents the time and the y-axis the processed execution time of a task. The lower diagram shows when the processor is in the idle state. 83

8.1 Subsystems and components of the discrete-event simulation. The real-time system is divided in the subsystems: stimulation, software, operating system, and hardware. The extension of this model in comparison to singlecore systems [AUT10] are multiple cores, which are arranged to core clusters. Each core cluster has a scheduler which allocates the tasks, managed by this scheduler, to the cores of the core cluster. Quartz oscillators are mapped to cores in order to provide the clock for the processing frequency. 87
8.2 Example of the description of a behavioral model. 89
8.3 State diagram of simulation sequencer. 90

8.4 Behavioral model of stimulation component. 92
8.5 Behavioral model of process component. 95
8.6 Behavioral model of process instance component. 96
8.7 Behavioral model of runnable instance component. 98
8.8 Behavioral model of quartz oscillator component. 100
8.9 Behavioral model of core component. 102

9.1 Application of Monte-Carlo randomization in order to determine the value of π. 114
9.2 Statistical estimators and bootstrapped confidence bounds of a distribution of metric values in a cluster Z_k. 117

10.1 Sensors and actors of an automotive powertrain system (By courtesy of Continental Automotive GmbH). 122
10.2 Scatter diagrams of *mNL* (y-axis) as function of system utilization (x-axis) for the algorithms *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2}. The diagram shows 500.000 systems, generated by a probabilistic automotive system model and simulated by use of computer cloud. 124
10.3 Scatter diagrams of *mNL* (y-axis) as a function of the system utilization (x-axis) for the algorithms *Partly-Pfair-PD*^{2}, *P-ERfair-PD*^{2}, and *WFD-EDF*. The diagram shows 25.000 system models originating from a probabilistic automotive powertrain system model. 130
10.4 Comparison of the scheduling algorithms *Partly-Pfair-PD*^{2}, *P-ERfair-PD*^{2}, and WFD-EDF by statistical estimators on *mNL* (y-axis) as a function of the system utilization (x-axis). 131
10.5 Subset of examined models from porting case study (Figure 10.3) and WFD-EDF scheduling, where only models are considered with task *T00_RPM* and *T09_10MS* on different cores. 131
10.6 Robustness consideration of *Partly-Pfair-PD*^{2} and *P-ERfair-PD*^{2} through random variation of the task section execution time via a Weibull distribution. The task section execution time can be twice as long as allowed. . 132

List of Tables

2.1 List of all task symbols. 13
2.2 Task section symbols. 14
2.3 Job time intervals and timestamps. 20

3.1 Classification of general utilization based schedulability bounds on multicore scheduling [CFH+04]. The maximal system utilization U depends on the number of cores m and for some groups of scheduling algorithms on the maximal task utilization $\alpha = U_{max}(\tau)$. 32
3.2 Classification of multicore scheduling algorithms. 33

10.1 Task set parameters of porting case study. 127

Part V

Appendix

Appendix A

Pseudo-Code of Scheduling Algorithms

A.1 Algorithm EDF

Algorithm 2 Pseudo-code of EDF *BLOCKING* function. Returns 0 when task T_i does not block the core.

Require: T_i (task)
  return $\leftarrow 0$

Algorithm 3 Pseudo-code of EDF *NOMINEE* function. Returns 1 when task T_i is nominated for execution.

Require: T_i (task), f (core)
  return $\leftarrow 1$

Algorithm 4 Pseudo-code of EDF *Compare* function. Returns from two tasks the task with the higher rank, as long there is one, and returns task T_a when both tasks have the same rank. The symbol $T_x.D$ denotes the absolute deadline of task T_x.

Require: T_a (task a), T_b (task b)
  if $T_a.D \leq T_b.D$ **then**
    return $\leftarrow T_a$
  else
    return $\leftarrow T_b$
  end if

A.2 Algorithm *Partly-Pfair-PD*^{2}

Algorithm 5 Pseudo-code of *Partly-Pfair-PD*^{2} *BLOCKING* function. Returns 1 when task T_i blocks a core and returns 0 when task T_i does not block the core. The function *TaskState*T_x returns the actual state of a task (see Figure 8.6 for state definition).

Require: T_i (task)
  if TaskState (T_i) = RUNNING **then**
    return ← 1
  else
    return ← 0
  end if

Algorithm 6 Pseudo-code of *Partly-Pfair-PD*^{2} *NOMINEE* function. Returns 1 when task T_i is nominated for execution and returns 0 when task T_i is not nominated for execution. The symbol t denotes the actual time and $r'(T_x^k)$ returns the pseudo-release time of the actual task section T_x^k of task T_x.

Require: T_i (task), f (core)
  if $t \geq r'(T_i^k)$ **then**
    return ← 1
  else
    return ← 0
  end if

Algorithm 7 Pseudo-code of *Partly-Pfair-PD*^{2} *Compare* function. Returns from two tasks the task with the higher rank, as long there is one, and returns task T_a when both tasks have the same rank. The symbol $d'(T_x^k)$ denotes the pseudo-deadline, $b'(T_x^k)$ the overlapping-bit, $D'(T_x^k)$ the group deadline, of the actual task section T_x^k of task T_x respectively.

```
Require: T_a (task a), T_b (task b)
  if d'(T_a^k) < d'(T_b^k) then
    return ← T_a
  else if d'(T_a^k) = d'(T_b^k) then
    if b'(T_a^k) = 1 ∧ b'(T_b^k) = 0 then
      return ← T_a
    else if b'(T_a^k) = b'(T_b^k) then
      if D'(T_a^k) ≥ D'(T_b^k) then
        return ← T_a
      else
        return ← T_b
      end if
    else
      return ← T_b
    end if
  else
    return ← T_b
  end if
```

A.3 Algorithm *P-ERfair-PD*^{2}

Algorithm 8 Pseudo-code of *P-ERfair-PD*^{2} *BLOCKING* function. Returns 1 when task T_i blocks a core and returns 0 when task T_i does not block the core. The function *TaskState*T_x returns the actual state of a task (see Figure 8.6 for state definition).

```
Require: T_i (task)
  if TaskState (T_i) = RUNNING then
    return ← 1
  else
    return ← 0
  end if
```

Algorithm 9 Pseudo-code of *P-ERfair-PD*^{2} *NOMINEE* function. Returns 1 when task T_i is nominated for execution and returns 0 when task T_i is not nominated for execution.

Require: T_i (task), f (core)
  return $\leftarrow 1$

Algorithm 10 Pseudo-code of *P-ERfair-PD*^{2} *Compare* function. Returns from two tasks the task with the higher rank, as long there is one, and returns task T_a when both tasks have the same rank. The symbol $d'(T_x^k)$ denotes the pseudo-deadline, $b'(T_x^k)$ the overlapping-bit, $D'(T_x^k)$ the group deadline, of the actual task section T_x^k of task T_x respectively.

```
Require: T_a (task a), T_b (task b)
  if d'(T_a^k) < d'(T_b^k) then
    return ← T_a
  else if d'(T_a^k) = d'(T_b^k) then
    if b'(T_a^k) = 1 ∧ b'(T_b^k) = 0 then
      return ← T_a
    else if b'(T_a^k) = b'(T_b^k) then
      if D'(T_a^k) ≥ D'(T_b^k) then
        return ← T_a
      else
        return ← T_b
      end if
    else
      return ← T_b
    end if
  else
    return ← T_b
  end if
```