technische universität dortmund

fakultät für informatik

computer science 12

Master Thesis

Multicore Systems with Dynamic Real-Time Guarantees

Lea Schönberger

May 30, 2017

Supervisors:
Prof. Dr. Jian-Jia Chen
Dipl.-Inf. Georg von der Brüggen

TU Dortmund University
Faculty of Computer Science
Computer Science Chair 12 (Embedded Systems)
http://ls12-www.cs.tu-dortmund.de
Design Automation for Embedded Systems Group

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Structure of the Thesis . . . 3

2. Real-Time Systems **5**
2.1. Theoretical Fundamentals of Real-Time Systems . . . 5

3. Mixed-Criticality Systems **9**
3.1. The Theoretical Mixed-Criticality Model . . . 9
3.2. Fixed-Priority Scheduling on Single Processor Systems . . . 10
3.3. Mixed-Criticality Systems in Practice . . . 12

4. Systems with Dynamic Real-Time Guarantees **15**
4.1. Task Model . . . 16
4.2. System Model . . . 17
4.3. Schedulability Test . . . 18
4.4. Optimal Priority Assignment . . . 19

5. Multicore Fixed-Priority Scheduling **21**
5.1. Partitioned Scheduling . . . 21
5.2. Semi-Partitioned Scheduling . . . 23

6. Multicore Systems with Dynamic Real-Time Guarantees **27**
6.1. System Model . . . 27
6.2. Partitioned Approach . . . 29
6.2.1. Schedulability Test . . . 29
6.2.2. Priority Order . . . 30
6.2.3. Classical Bin-Packing Heuristics . . . 31
6.2.4. Enhanced Bin-Packing Heuristics . . . 36
6.2.5. Limits of Partitioned Scheduling . . . 41
6.3. Semi-Partitioned Approach . . . 41
6.3.1. Task Splitting . . . 42
6.3.2. Highest-Priority Task Splitting . . . 46

6.3.3. Complete Task Migration 49
6.3.4. Timing Tolerable Task Migration 51

7. Evaluation **55**
7.1. Experiment Setup 55
7.2. Partitioned Scheduling Strategies 56
7.3. Task Splitting and Highest-Priority Task Splitting 60
7.4. l-Failure-Proof Multicore Systems with Dynamic Real-Time Guarantees . . 61
7.5. Timing Tolerable Task Migration 63
7.6. Conclusion 65

A. Appendix **67**

List of Figures **80**

List of Algorithms **81**

Bibliography **85**

Affidavit (Eidesstattliche Versicherung) **86**

1. Introduction

1.1. Motivation

With respect to computing hardware, several reliability threats can be identified such as the so-called transient faults. These, in contrast to permanent faults, occur as bursts provoked, for instance, by power supply jitters or electromagnetic interference due to atmospheric effects [15]. Such transient faults can have an impact on the contemplated system's execution behavior and may lead to incorrect computation results or even to a system failure. Especially referring to safety-critical applications as employed in the avionics or automotive sector, it is indispensable to take fault prevention as well as fault recovery strategies into account throughout the system design process. Thus, manifold techniques are applied by the industry to deal with faulty execution behavior, e.g., spatial isolation of certain components, hardware redundancy, remapping of logical system functionalities onto a flawless subset of hardware resources, excessive monitoring, and, particularly emphasized in this thesis, re-execution of erroneous software jobs [13].

An application's task shall be considered, after whose execution a fault detection routine is conducted which determines if the delivered result is correct or faulty. In the former case, each deadline is met, as depicted in Figure 1.1 by means of tasks τ_1, τ_2, and τ_3. In the latter case, the faulty task is re-executed in order to ensure a valid output, whereas the number of potential re-executions should be limited depending on the system characteristics. However, by reason of fault recovery, a task's worst-case execution time can be prolongated significantly, so that, in consequence, it may engender another task's deadline miss. This situation is exemplarily illustrated in Figure 1.2, where a job of τ_3 misses its deadline due to a twofold re-execution of a task instance of τ_1.

When a fault occurs, a system is said to perform a mode change, which can be depicted as a switch from the normal execution scenario to a special state of alert. In the field of

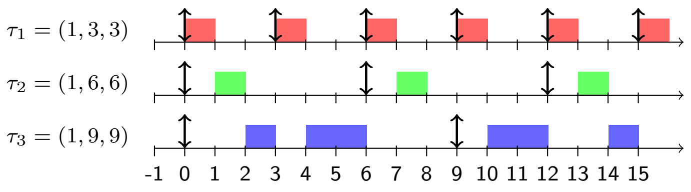

Figure 1.1.: A feasible schedule according to the rate-monotonic policy without error occurrence.

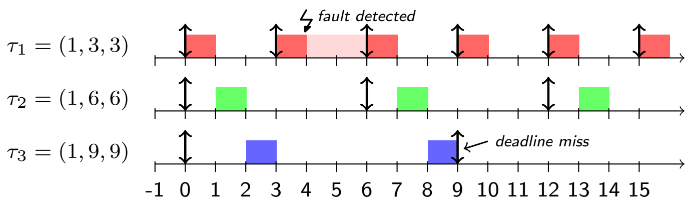

Figure 1.2.: Task τ_3 misses its deadline due to multiple re-executions of τ_1 by reason of fault recovery (light red).

so-called Mixed-Criticality Systems, it is commonly necessary to carry out online adaption strategies in such cases as, e.g., neglecting or dropping low-criticality tasks for the benefit of higher-criticality tasks as well as recomputing and adjusting deadlines to ensure that tasks with high criticality and thus of high or utmost importance meet their deadlines so that the system's proper functioning can be guaranteed.

Von der Brüggen et al. [24] suggested a diverging approach, denoted as *System with Dynamic Real-Time Guarantees* model, according to which, instead of dealing with low-criticality tasks inequitably, it is taken into consideration that a low-criticality task's result could still be useful despite its lateness. Hence, in the Systems with Dynamic Real-Time Guarantees model introduced in [24], low-criticality tasks are neither neglected nor aborted but kept running even in case of a system mode change due to the occurrence of a fault, as long as the high-criticality tasks meet their deadlines. Following this concept, von der Brüggen et al. [24] introduced the notion of *full timing guarantees*, i.e., adherence to each task's timing requirements, as well as of *limited timing guarantees*, i.e., compliance with all high-criticality tasks' timing properties as well as bounded tardiness with respect to low-criticality tasks. Devoid of any online adaption, full timing guarantees can be assured in the normal system mode and limited timing guarantees, in contrast, in case of fault occurrence.

Since the revealed concept has been designed for single-core systems, in this master thesis shall be examined if and how the Systems with Dynamic Real-Time Guarantees model can be provided for multicore platforms likewise.

1.2. Structure of the Thesis

In this thesis, initially, a brief introduction into the fundamental concepts of Real-Time Systems (cf. 2) as well as into those of Mixed-Criticality Systems (cf. 3) will be provided. Thereon, in chapter 4, the *System with Dynamic Real-Time Guarantees* model proposed by von der Brüggen et al. [24] will be introduced in detail, including the deployed task model, the concrete system model, an appropriate schedulability test for establishing a System with Dynamic Real-Time Guarantees as well as the respective method for generating an optimal priority assignment.

The following chapter (cf. 5) covers the most common approaches to multiprocessor scheduling that are applicable to the specific problem definition addressed in this thesis, namely, the partitioned and the semi-partitioned scheduling paradigm. Subsequently (cf. 6), the attempt is made to transfer the System with Dynamic Real-Time Guarantees model by von der Brüggen et al. [24] onto multiprocessor systems, for which reason in the first place the system model of a *Multicore System with Dynamic Real-Time Guarantees* is defined. Building on this, several partitioned approaches aiming at the formation of systems satisfying the characteristics of a Multicore System with Dynamic Real-Time Guarantees are discussed (cf. 6.2), where, regarding each strategy, the devised algorithms are illustrated in pseudo-code. Moreover, two semi-partitioned concepts, the task splitting (cf. 6.3.1) as well as the highest-priority task splitting paradigm (cf. 6.3.2), are introduced in order to achieve a Multicore System with Dynamic Real-Time Guarantees in those cases, in which partitioned approaches reach their limits.

Pursuing another objective, more precisely, an increased reliability of the particular Multicore System with Dynamic Real-Time Guarantees, additionally the *l-Failure-Proof Multicore System with Dynamic Real-Time Guarantees* model is presented, which, by means of complete task migration (cf. 6.3.3), ensures the adherence to the characteristics of a Multicore System with Dynamic Real-Time Guarantees even if l processors break down. Furthermore, the concept of timing tolerable task migration is suggested (cf. 6.3.4) for the purpose of reducing the tardiness of a specified group of timing tolerable tasks under the circumstances that, e.g. due to electromagnetic radiation, each task on one or more processors exhibits faulty execution behavior.

Finally, the proposed strategies are analyzed and evaluated in chapter 7 by means of the results of comprehensive experiments.

2. Real-Time Systems

Henceforth, an introduction into the topic of real-time systems is given following the work of Buttazzo [7] (especially pp. 1-51).

A real-time system is characterized by the fact that it mandatorily needs to satisfy temporal correctness, i.e., timeliness with respect to specified constraints, as well as functional correctness, i.e., the delivery of valid results. These days, real-time systems can be found in miscellaneous fields of application including the automotive and avionics industry, the medical sector as well as the telecommunication and consumer electronics sector besides many more. Depending on the application area, disparate categories of real-time systems can be distinguished with respect to the consequences of a malfunction concerning the particular system, namely, soft real-time systems, e.g. multimedia applications where late results are not very useful, firm real-time systems, e.g. telecommunication systems where late results may be useless but not harmful, and hard real-time systems, in terms of which a deadline miss' impact is hazardous, e.g. in the avionics sector or with regard to a nuclear power plant.

Notwithstanding the broad range of application, the design of real-time systems is still challenging due to manifold reasons. On the one hand, there exist many misconceptions concerning real-time systems, particularly between academia and the industry [22]. On the other hand, ad hoc techniques as well as heuristic approaches are still the favored solution regarding the implementation of real-time systems, which causes several issues such as a large, complex and incomprehensible amount of code, difficult software maintainability and tricky verification of time constraints, notably owing to the fact that the major share of code is realized in low-level languages [7]. Hence, the resulting software may exhibit unpredictable behavior with disastrous consequences in exceptional situations, if the system lacks of specific mechanisms to ensure timing properties.

2.1. Theoretical Fundamentals of Real-Time Systems

Before addressing the topic of mixed-criticality systems, which form a subset of real-time systems, some theoretical prerequisites will be provided to offer the recipient the necessary knowledge for deeper understanding of the problems discussed in this thesis.

Consider a process executed sequentially by a processor which needs to be performed several times. Such sequence of processes is termed *task*, denoted as τ_i, whereas a single instance is denominated *job*. A job of task τ_i is characterized by its *worst-case execution time* C_i, i.e. the amount of time required from the moment in which the job's execution begins, identified as the *start time* s_i, until the moment of its completion, namely the *finishing time* f_i. Notwithstanding, the *response time* R_i arises out of the time span from the instant when τ_i becomes ready for execution, referred to as the *arrival time* or *release time* a_i, until the finishing time. Each job has to be finished before a specified point of time, the *deadline*, which can either refer to absolute time, i.e., the so-called *absolute deadline* d_i, or describe an amount of time relative to the job's arrival time, i.e., the *relative deadline* D_i. These parameters are, by way of illustration, visualized in Figure 2.1. In addition, a job's *lateness* L_i describes the delay between its finishing time and its deadline, i.e., $L_i = f_i - D_i$, while the *tardiness* E_i denotes the time a job remains active after its deadline, i.e., $E_i = \max(0, L_i)$.

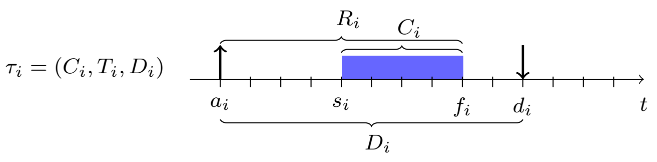

Figure 2.1.: Typical parameters of a real-time task [7].

A task can release its jobs either periodically or aperiodically, whereat in the latter case again two different options are conceivable: The jobs either arrive arbitrarily or sporadically. In all these cases, a task is characterized by the so-called *period* or *interarrival time* T_i, a timing parameter which describes the amount of time before whose expiry no other job of task τ_i is permitted to become ready for execution. If the deadline equals the interarrival time, i.e., $D_i = T_i$, the task is said to have *implicit* deadlines (cf. Figure 2.2), if, however, the deadline is smaller than or equal to the interarrival time, i.e., $D_i \leq T_i$, the task is referred to as *constrained*-deadline task (cf. Figure 2.3). Otherwise, i.e., $D_i > T_i$ for some tasks, the task set is termed *arbitrary*-deadline task set (cf. Figure 2.4). Concerning sporadic tasks, it is furthermore necessary to introduce an additional timing parameter, the moment of the first job's arrival, known as *phase* Φ_i. Moreover, it universally holds that implicit-deadline task sets are a subset of constrained-deadline task sets [19], which, in turn, are a subset of arbitrary-deadline sets.

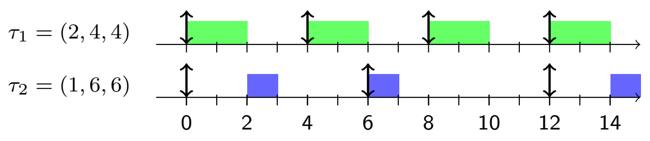

Figure 2.2.: An implicit-deadline task set.

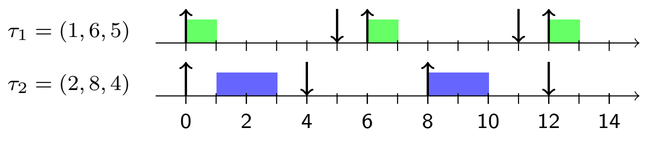

Figure 2.3.: A constrained-deadline task set.

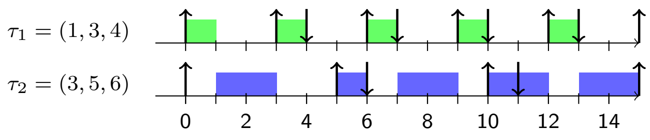

Figure 2.4.: An arbitrary-deadline task set.

Apart from this, each job is endowed with a *priority* by means of which it is assigned processor execution time according to the particular applied scheduling policy. The priority can either be fixed or dynamic and should be chosen with respect to the system's characteristics and purpose.

Regarding a task system $\mathbf{T}$ consisting of concurrent tasks $\tau_1, \ldots, \tau_n$ with $n \in \mathbb{N}$, the order in which the tasks are executed by the processor is established by a *scheduling policy*. Such scheduling policies can display various characteristics concerning their way of operation. Actually, a scheduling algorithm can be either *preemptive*, i.e., the running job may be interrupted at any point in order to grant execution time to another task, or *non-preemptive*, i.e., a job can never be interrupted but is executed until its completion. Furthermore, it can be either *static*, i.e., all scheduling decisions are premised on parameters specified before a task's activation, or *dynamic*, i.e. such parameters may change throughout the system life cycle. Apart from this, a scheduling algorithm can be either *offline*, i.e., the schedule is generated before any task's activation, or *online*, i.e., the schedule is produced and modified while the system is running. Eventually, it can be either *optimal*, i.e., the algorithm minimizes a given cost function or otherwise creates a feasible schedule if existent, or *heuristic*, i.e., the scheduling is performed by dint of a heuristic function which approximates the optimal schedule but does not necessarily achieve it [7].

Moreover, clarifying the aforesaid, a schedule is denoted *feasible* if all tasks of task set $\mathbf{T}$ can be outright executed taking into account their respective timing properties [7]. A set of tasks is termed *schedulable* if there exists at least one algorithm that results in a feasible schedule [7].

3. Mixed-Criticality Systems

During the last ten years the industry's interest in running several independent applications of different so-called criticalities, i.e., of different importance, timing constraints and therefore diversely severe consequences of a deadline miss, in the same integrated platform increased enormously. This evolution is based on the attempt to lower costs of the particular systems while maintaining the prevailing performance [13] as well as on the necessity to comply with other technical or practical requirements regarding e.g. weight, size, power consumption, and heat development [6].

The notion of criticality is most commonly used with respect to functional safety, i.e., the non-existence of devastating consequences with respect to the system's user as well as to the environment [12], although the term has a broader variety of meanings which is discussed by Avizienis et al. [3]. However, the definition of criticality is inconsistent throughout the aggregate of manifold industrial sectors, whose respective technical definitions of the former are manifested in specific safety-related standards, such as the DO-178C for avionic software, the ISO 26262 for the automotive sector, and the generic IEC 61508 which serves as a groundwork for further domain-specific standards [13]. Despite these discrepancies, in almost every standard the assignment of criticality levels to individual system functions is defined as the result of a failure modes and effect criticality analysis (FMECA [16]) routine, after whose completion not only criticality categories but also additional recommendations are given.

Nevertheless, industrial mixed-criticality systems differ from the academic model, among others due to ambiguous termini technici as well as variegated strategies and implementations. For this purpose, the theoretical mixed-criticality model as employed in the common research landscape shall be introduced hereinafter. Thereupon, the issue of fixed-priority scheduling on single-core mixed-criticality systems will be compendiously explored (cf. 3.2).

3.1. The Theoretical Mixed-Criticality Model

In the following, the theoretical mixed-criticality model will be illustrated referring to the review paper on mixed-criticality systems by Burns and Davis [6].

The first publication on mixed-criticality systems, coining this particular term in the specific sense in which it was adopted thenceforward, was issued by Vestal [23] in the year

2007. Since that time, various papers have been published on this topic, not necessarily employing the same system and task model. However, Burns and Davis [6] propose a generally applicable model that will be availed at this point.

A mixed-criticality system consists of a finite set K of components, whereat each single one is characterized by a criticality level L_i and encloses a finite set of sporadic tasks $\mathbf{T}$. A task $\tau_i = (T_i, D_i, C_i, L_i)$ is specified by its minimum interarrival time or period, deadline, worst case execution time, and by the criticality level pertaining to its respective superordinate component. Since tasks descending from disparate components must not affect each other, it is necessary to implement protective mechanisms which impede a job from exceeding its granted execution budget and, besides that, ensure that no job's timing parameters are violated in the job generation process. Otherwise, as a consequence, development expenses would increase tremendously due to the fact that each component were obliged to be realized on the highest criticality level [6].

Unlike conventional real-time systems, in mixed-criticality systems, the computation of a task's worst-case execution time C_i is dependent on its criticality level, more precisely, it becomes larger as a function of a higher criticality level [6].

Furthermore, several models consider a mixed-criticality system to execute in a certain amount of so-called *criticality modes*, whereat the system initially operates in the lowest criticality mode whilst all tasks' timing properties are satisfied. As soon as any task infringes its particular time restrictions, a criticality mode change occurs, in course of which each task's timing properties may be renewed. It is frequently the case that the number of criticality levels is restricted to merely two, a *high- (HI)* and a *low-criticality mode (LO)*, an architecture which is referred to as *dual-criticality system*. Deviantly, in such systems a task is typically characterized as $\tau_i = (T_i, D_i, C_i(HI), C_i(LO), L_i)$, where $C_i(HI)$ denotes the worst-case execution time in high-criticality mode and $C_i(LO)$ specifies the worst-case execution time in low-criticality mode. A task with $L_i = HI$ is denominated as *high-criticality task*, whereas a task with $L_i = LO$ is termed *low-criticality task*. Concerning such systems, a mode change from *LO* to *HI* criticality mode is performed if at least one task exceeds its $C_i(LO)$ budget, whereby it is a common practice to only guarantee the high-criticality tasks' timeliness as soon as the system executes in *HI* criticality mode. Depending on the actual system model, a descending criticality mode change is possible as well, i.e., in terms of dual-criticality systems, from *HI* to *LO* criticality mode, even though the majority of publications refuse this option, so that the system finally remains in the highest criticality mode.

3.2. Fixed-Priority Scheduling on Single Processor Systems

Since Vestal's first publication on mixed-criticality systems [23], a growing interest with respect to this topic emerged in the research community, that resulted in multiple research approaches and results, which will be sketched briefly in what follows, according to the review paper by Burns and Davis [6].

After having demonstrated that neither a rate monotonic (RM) nor a deadline monotonic (DM) scheduling policy is optimal in terms of mixed-criticality systems, Vestal's paper [23] revealed that *Audsley's Optimal Priority Assignment algorithm* [2] (also denoted as *Audsley's approach* or *OPA*) is employable in this scope of application. Nine years later, this strategy was shown to be even optimal for mixed-criticality systems by Dorin et al. [11]. Due to the matter of fact that Audsley's approach is of further interest in the subsequent chapter (cf. 4), it shall be explained concisely at this juncture.

To establish an optimal priority order of a task set $\mathbf{T}$ on a single-core dual-criticality system, an appropriate task has to be identified for each priority level. Assume that a tasks set $\mathbf{T}$ is divided into two disjoint subsets of high- and low-criticality tasks. Commencing with the search for a candidate to take the lowest possible priority, a schedulability analysis is performed with the low-criticality task set's lowest-priority task filling this vacancy. In case of success, the next priority level is contemplated, otherwise, the procedure is repeated considering the high-criticality task set's lowest-priority task. Likewise, the attention is drawn to the next priority-level, if the assessment exhibits a positive outcome. Provided that neither of both tasks can be proven to be schedulable under the examined priority assignment, the task set is indicated as unschedulable [2]. Synoptically, this algorithm is provided in pseudo-code in Algorithm 1.

Algorithm 1 Audsley's Optimal Priority Assignment algorithm for single-core mixed-criticality systems.

```
Input: $\mathbf{T}_{HI} = \tau_1, \dots, \tau_n, \mathbf{T}_{LO} = \tau_1, \dots, \tau_l$ with $\mathbf{T} := \mathbf{T}_{HI} \cup \mathbf{T}_{LO}$

Output: $\mathbf{T}$ with optimal priority assignment or UNSCHEDULABLE

  procedure FIND PRIORITY ASSIGNMENT($\mathbf{T}_{HI} = \tau_1, \dots, \tau_n, \mathbf{T}_{LO} = \tau_1, \dots, \tau_l$ with $\mathbf{T} := \mathbf{T}_{HI} \cup \mathbf{T}_{LO}$)
      for ($i = 1;\ i \leq n;\ i := i + 1$) do
          $\mathbf{T}^* := \emptyset$
          if (ISSCHEDULABLE($\mathbf{T}^* \cup \tau_1, \tau_1 \in \mathbf{T}_{LO}$, prio($\tau_1$) $:= n$) then
              $\mathbf{T}^* := \mathbf{T}^* \cup \tau_1, \mathbf{T}_{LO} := \mathbf{T}_{LO} \setminus \tau_1$
          else if (ISSCHEDULABLE($\mathbf{T}^* \cup \tau_1, \tau_1 \in \mathbf{T}_{HI}$, prio($\tau_1$) $:= n$) then
              $\mathbf{T}^* := \mathbf{T}^* \cup \tau_1, \mathbf{T}_{HI} := \mathbf{T}_{HI} \setminus \tau_1$
          else
              return UNSCHEDULABLE
```

Furthermore, it was shown by Baruah and Vestal [5] in 2008 that fixed-priority scheduling is not dominated by earliest-deadline-first (EDF) scheduling with regard to mixed-criticality systems, but rather ranks behind, since there exist feasible systems which are not schedulable under the EDF policy [6]. On that score and due to the fact that the later introduced Systems with Dynamic Real-Time Guarantees model (cf. 4) by von der Brüggen et al. [24] considers fixed-priority scheduling only, this overview shall be confined to the delineation of fixed-priority scheduling for mixed-criticality systems.

Referring to fixed-priority scheduling for mixed-criticality systems, three different concepts protrude in academic research, namely, strategies based on the usage of response-time analysis, methods employing slack scheduling and those performing period transformations [6], whereat the latter issues shall solely be treated as marginal at this point.

A weak point of Vestal's basic approach consists in the necessity to evaluate high-criticality tasks as well as low-criticality tasks as if they were equally situated on the high-criticality level, which leads to a restriction in terms of resource usage. However, an improvement can be achieved by applying execution time monitoring as well as mechanisms to impede the exceeding of execution time budgets [6] (cf. 1.1). Vestal's approach was further developed in 2011, culminating in a scheduling and dedicated analysis model which outperforms all antecedent attempts relating to fixed-priority scheduling on mixed-criticality systems. Either way, in the course of this just as in the vast majority of subsequent publications, dual-criticality systems are considered, which evince one key characteristic. In fact, all low-criticality tasks are dropped or at least neglected in case any task violates its execution time restriction.

Having illustrated the development in response-time analysis based fixed-priority scheduling, the residual aforementioned approaches will be briefly adumbrated. In point of fact, the so-called slack scheduling provides an alternative solution for scheduling periodic dual-criticality task systems, as low-criticality task instances are executed in the slack induced by those evincing high-criticality properties, but exhibits a weak point when considering sporadic tasks, since it is difficile to determine when the slack of a non-appearing sporadic task can be assigned to a low-criticality task's instance [6]. Another concept attempts to gain advantage of splitting tasks into two or more subtasks, which are individually assigned new modified deadlines. Notwithstanding, due to the excessive overhead engendered by frequent context switches, this approach is not deemed beneficial with respect to single-core scheduling [6], but is revived regarding the research on multicore scheduling (cf. 5) and will be turned into account later on in this thesis (cf. 6.3.1, 6.3.2).

Apart from the outlined concepts, also other strategies are pursued in the academic research landscape, but, however, these are not necessarily conducive to this thesis and will be omitted on this occasion. Nevertheless, a detailed overview can be found in [6].

3.3. Mixed-Criticality Systems in Practice

Having gained an insight into the current state of research on fixed-priority scheduling on single-core mixed-criticality systems, it becomes evident that all above approaches, especially to dual-criticality systems, exhibit the same weaknesses, which are a major point of criticism to many system engineers in the industry [6]. On the one hand, it is a matter of fact that as soon as a system enters the high-criticality mode, low-criticality tasks are discarded, but, however, since these tasks still evince a certain importance, a basic service level should be provided anyway. On the other hand, the possibility should be offered to return from high criticality to low criticality mode at some point of time in which the proper system functioning can be resumed, in particular with respect to systems designed for a long operating lifetime [6].

As a solution to these issues, von der Brüggen et al. [24] provide the *System with Dynamic Real-Time Guarantees* model, which will be introduced in the subsequent chapter.

4. Systems with Dynamic Real-Time Guarantees

As alluded to earlier, the occurrence of faults in mixed-criticality systems may provoke a system mode change from a low-criticality system mode to a higher one. More precisely, as soon as a faulty computation result is identified by a dedicated fault detection mechanism, the criticality mode is increased and a fault recovery routine is employed, namely, the particular task instance is either partially or completely re-executed. The major share of approaches disclosed by the academic research community follows the concept of neglecting low-criticality tasks in such cases in order to guarantee the satisfaction of higher-criticality tasks' timing properties, regardless of the faulty task's increased demand of execution time (cf. 3).

In their paper, von der Brüggen et al. [24] remark that owing to the employment of fault prevention strategies with respect to hardware as well as to software, fault occurrence should not be considered a common case in mixed-criticality systems. On that score, low-criticality tasks should neither be neglected nor aborted under such circumstances, but should rather proceed in execution, provided that all high-criticality tasks meet their respective deadlines. Pursuing this strategy, the obtained results of low-criticality tasks can still prove to be satisfactory in usage, despite their lateness.

With respect to their Systems with Dynamic Real-Time Guarantees model, von der Brüggen et al. [24] do not conventionally consider high- and low-criticality tasks, but instead focus on the particular tasks' timing requirements. Hence, a set of so-called *timing strict tasks* (casually denoted as *hard tasks*) as well as one of *timing tolerable tasks* (casually denominated *soft tasks*) is established. The former are obliged to comply with their respective timing properties in any event and can be regarded as a counterpart to the high-criticality task set in common dual-criticality systems, whereas the latter serve as an equivalent to the regular low-criticality tasks, which may violate some deadlines when executed in the high-criticality mode, but, nevertheless, should evince bounded tardiness.

Furthermore, unlike the majority of mixed-criticality systems proposed by academia (cf. 3), the System with Dynamic Real-Time Guarantees model enables a system to return from high- to low-criticality mode as soon as the normal system behavior is restored and, thus, its proper functioning is endangered no longer.

Due to the presupposition that faults occur quite infrequently, von der Brüggen et al. provide an offline verified fixed-priority preemptive scheduling strategy instead of a cost-intensive online scheduling policy. Merely an online monitoring mechanism is required to observe the system's overall timeliness, subject to the condition that the prerequisites for establishing a System with Dynamic Real-Time Guarantees are fulfilled.

In the following, the Systems with Dynamic Real-Time Guarantees model shall be introduced in detail according to von der Brüggen et al. [24], beginning with the specified task model (cf. 4.1), proceeding with the concrete system model (cf. 4.2) as well as with the applied schedulability test (cf. 4.3) and finally depicting an optimal priority assignment (cf. 4.4).

4.1. Task Model

The task set $\mathbf{T}$ considered in the Systems with Dynamic Real-Time Guarantees model consists of a finite collection $(\tau_1, \ldots, \tau_n)$, with $n \in \mathbb{N}$, of independent mixed-criticality sporadic tasks executed on a single processor.

Each task $\tau_i = (T_i, D_i, C_i^N, C_i^A)$ is characterized by a minimum inter-arrival time T_i, a relative deadline D_i and two distinct worst-case execution times C_i^N and C_i^A. The task sets contemplated are either constrained- or implicit-deadline task sets, i.e., $D_i \leq T_i$ or $D_i = T_i$, respectively.

Since the System with Dynamic Real-Time Guarantees model presumes a dual-criticality system, a task τ_i can be executed in two different *execution modes*, which have an impact on the task's worst-case execution time. Accordingly, the task's plain, flawless execution time (plus the amount of time required for fault detection) is denoted C_i^N or worst-case execution time under *normal execution.* In this case of normal execution, the task is said to be executed in *normal mode.* Otherwise, the task is indicated to be in *abnormal mode*, whereat its worst-case execution time under *abnormal execution*, which comprises the normal worst-case execution time and, additionally, the amount of time necessary for fault recovery, is termed C_i^A. Thus, it always holds that $C_i^A \geq C_i^N$. The exact runtime of the fault recovery routine depends on the particular re-execution mode and is not relevant at this point, but will be explored when broaching the issue of experiments (cf. 7). However, if a certain number of tasks is executed in abnormal mode, the system can be informally declared to be *abnormal* or in *abnormal mode.*

Due to the fact that the System with Dynamic Real-Time Guarantees model assumes at least a few timing strict tasks to be eminently safety-critical, i.e., a deadline miss engenders hazardous consequences, neither modifications in terms of the tasks' release rate nor regarding the deadline are admissible, which results in the assignment of uniform deadlines in normal and abnormal mode.

The utilization of a task τ_i in normal mode, termed *normal utilization*, is denoted by $U_i^N = \frac{C_i^N}{T_i}$, while, analogously, its utilization in abnormal mode, referred to as *abnormal utilization*, is described by $U_i^A = \frac{C_i^A}{T_i}$. Hence, the system utilization is qualified by $U_{sum}^N = \sum_{\{\tau_i \in \mathbf{T}\}} U_i^N$ in normal mode and by $U_{sum}^A = \sum_{\{\tau_i \in \mathbf{T}\}} U_i^A$ in abnormal mode. Moreover, the response time of the j-th instance of task τ_i is referred to as $R_{i,j}$, its worst-case response time as R_i^N and R_i^A in normal as well as in abnormal mode, respectively. A job's tardiness is denoted by $E_i = R_i^A - D_i$.

4.2. System Model

The System with Dynamic Real-Time Guarantees model distinguishes between timing strict tasks, denoted as $\mathbf{T}_{hard}^A$, and timing tolerable tasks, identified as $\mathbf{T}_{soft}^A$, whereby the partition of $\mathbf{T}$ into $\mathbf{T}_{hard}^A$ and $\mathbf{T}_{soft}^A$ is assumed to be given. If for all active tasks a flawless execution can be guaranteed, provided that no further faults occur, the system is denoted as system with *full timing guarantees.* Otherwise, if only the timing strict tasks' timeliness can be assured, it is referred to as a system with *limited timing guarantees.*

Henceforth, the formal specifications of a System with Dynamic Real-Time Guarantees as introduced by von der Brüggen et al. are presented.

Definition 1 (System with Dynamic Real-Time Guarantees [24]).
Consider a set of $\mathbf{T}$ tasks under a fixed-priority scheduling. A job of task $\tau_i \in \mathbf{T}$ cannot start its execution until all the jobs of task τ_i that arrived earlier are completed. The jobs of all tasks always have to be executed and cannot be aborted.
If the system runs with *full timing guarantees*, then the hard real-time guarantees hold for each task:

- $\mathbf{T}$: Each task $\tau_i \in \mathbf{T}$ must meet the hard relative deadline.

If the system runs with *limited timing guarantees*, the service level guarantees are downgraded from hard real time guarantees to bounded tardiness for some of the tasks:

- $\mathbf{T}_{hard}^A \subseteq \mathbf{T}$: Each task $\tau_i \in \mathbf{T}_{hard}^A$ is required to meet the hard relative deadline.
- $\mathbf{T}_{soft}^A \subseteq \mathbf{T}$: Each task $\tau_i \in \mathbf{T}_{soft}^A$ must have bounded tardiness, i.e., $0 \le E_i < \gamma$ for some fixed value γ.

Each task in $\mathbf{T}$ has to be placed either in $\mathbf{T}_{hard}^A$ or in $\mathbf{T}_{soft}^A$, thus $\mathbf{T}_{hard}^A \cap \mathbf{T}_{soft}^A = \emptyset$ and $\mathbf{T}_{hard}^A \cup \mathbf{T}_{soft}^A = \mathbf{T}$.

In conformity with von der Brüggen et al. [24], a task set $\mathbf{T}$ is termed *feasible* or *feasibly schedulable* as a System with Dynamic Real-Time Guarantees, if the conditions providing *full timing guarantees* as well as those ensuring *limited timing guarantees* hold for the given partitioning and priority order. Such fixed priority ordering of a task set is denoted by P, with $P(\tau_i)$ describing the priority of task τ_i, while $P(\tau_i) < P(\tau_j)$ when τ_i has higher priority than τ_j.

According to von der Brüggen et al. [24] shall be defined:

- $hp(\tau_k)$ as the set of tasks having higher priority than τ_k,
- $hep(\tau_k) = hp(\tau_k) \cup \tau_k$,
- $lp(\tau_k)$ as the set of tasks having lower priority than τ_k and
- $\Theta^A_{soft} := \{\tau_i \in \mathbf{T}^A_{soft} \mid \tau_i \in hp(\tau_j), \tau_j \in \mathbf{T}^A_{hard}\}$.

4.3. Schedulability Test

Considering a task set $\mathbf{T}$ with a given partition into $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ and a fixed-priority order P, $\mathbf{T}$ can be identified as a System with Dynamic Real-Time Guarantees, provided that it is scheduled according to P and the following conditions hold [24]:

1. Each task $\tau_i \in \mathbf{T}$ meets its hard deadline if all tasks are executed in the normal mode.
2. Each task $\tau_i \in \mathbf{T}^A_{hard}$ meets its hard deadline if some (or all) tasks are executed in the abnormal mode.
3. Each task $\tau_i \in \mathbf{T}^A_{soft}$ has a bounded tardiness if some (or all) tasks are executed in the abnormal mode.

The schedulability of a task set with constrained deadlines $\mathbf{T}$ scheduled according to a fixed-priority strategy can be examined by applying Time Demand Analysis (TDA) [20], which is an exact, pseudo-polynomial runtime schedulability test with respect to one specific task τ_k, subject to the condition that the priority ordering P is given and $hp(\tau_k)$ has already been proven to be schedulable. If TDA holds not only for one τ_k but for all $\tau_k \in \mathbf{T}$, $\mathbf{T}$ is avouched to be schedulable (by a fixed-priority scheduling policy complying with P).

Definition 2 (Time Demand Analysis [20]).
τ_k is schedulable if the following equation holds:

$$\exists t \text{ with } 0 < t \leq D_k \text{ and } C_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t \tag{4.1}$$

Against this backdrop, three conditions whose fulfillment serves as a prerequisite for a System with Dynamic Real-Time Guarantees can be defined (a proof is provided by von der Brüggen et al. [24]).

Theorem 1 (Exact Schedulability Test for Constrained Deadlines [24]).
For a given fixed priority ordering P, a task set $\boldsymbol{T}$ is a System with Dynamic Real-Time Guarantees as defined in 1, if the following three conditions hold:

1. *Full timing guarantees hold if $\boldsymbol{T}$ can be scheduled according to Time Demand Analysis (TDA) [20] when all tasks are executed in the normal mode, i.e., $C_i = C_i^N \; \forall \; \tau_i$.*

2. *When the system runs with limited timing guarantees all $\tau_i \in \boldsymbol{T}^A_{hard}$ will meet their hard deadlines if they can be proven to be schedulable by TDA [20] when all tasks are executed in the abnormal mode, i.e., $C_i = C_i^A \ \forall\ \tau_i$.*
3. *Each task $\tau_i \in \boldsymbol{T}^A_{soft}$ has bounded tardiness if $U^A_{sum} \leq 1$.*

Since von der Brüggen et al. [24] expect faults to emerge very rarely, a system utilization of $U^A_{sum} > 1$ in case of fault occurrence is considered to be acceptable for short instants, if $U^N_{sum} < 1$ and if faulty intervals of time are substantially shorter than the flawless ones. This follows from the fact that if fault bursts are assumed to only have an impact on a small amount of jobs by contrast with the number of jobs in the interim, the fault-free time is supposed to be sufficient for the system to return to full timing guarantees after each fault burst. Relatedly, if faults are expected to occur with a certain rate, limited timing guarantees are maintained for a longer interval of time only in case the given rate is reasonably high.

4.4. Optimal Priority Assignment

After having eludicated the necessary conditions for establishing a System with Dynamic Real-Time Guarantees under a given priority order, the issue of composing such an optimal priority assignment shall be addressed hereinafter.

In their paper, von der Brüggen et al. [24] prove that neither a deadline-monotonic nor a criticality-monotonic priority assignment is optimal for a System with Dynamic Real-Time Guarantees [24]. However, Audsley's Optimal Priority Assignment algorithm (cf. 3.2) is applicable with regard to finding a feasible priority order in case the employed schedulability test is OPA-compatible, in other words, if the following conditions hold [24] [9]:

1. The schedulability of a task τ_k, according to the applied test, may be dependent on the set of $hp(\tau_k)$, but not on the relative priority ordering of $hp(\tau_k)$.
2. The schedulability of a task τ_k, according to the applied test, may be dependent on the set of $lp(\tau_k)$, but not on the relative priority ordering of $lp(\tau_k)$.
3. When the priorities of any two tasks of adjacent priority levels are swapped, the task being assigned to the higher priority cannot become unschedulable according to the applied test, if it was previously schedulable at the lower priority.

By means of these requirements, von der Brüggen et al. [24] show that the schedulability test employed in the course of the System with Dynamic Real-Time Guarantees model (cf. 4.3) is OPA compatible and therefore can be availed in the context of Audsley's algorithm [24]. The optimal priority assignment algorithm for Systems with Dynamic Real-Time Guarantees proposed by von der Brüggen et al. [24] is portrayed in pseudocode in Algorithm 2 and proceeds similarly to the OPA algorithm for mixed-criticality systems introduced by Vestal [23] (cf. 3.2). In the following, it will be referred to as

procedure Find Optimal Priority Assignment($\mathbf{T}$), where the partition of $\mathbf{T}$ into $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ is considered to be given.

Algorithm 2 Feasible Priority Assignment Algorithm by von der Brüggen et al. [24]

```
Input: $\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$

Output: Feasible Order $P$ of $\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}$ or Not Possible

  Sort $\mathbf{T}^A_{hard}$ by $D_i$ increasingly
  Sort $\mathbf{T}^A_{soft}$ by $D_i$ increasingly
  Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$)

  procedure Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$)
      for ($n = |\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}|$; $n > 0$; $n := n - 1$) do
          $\tau_t$ := last element of $\mathbf{T}^A_{hard}$
          if (Try Priority($\tau_t$, $\left\{\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}\right\} \setminus \{\tau_t\}$,n,hard)) then
              $P(\tau_t) := n$
              $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \setminus \{\tau_t\}$
          else
              $\tau_t$ := last element of $\mathbf{T}^A_{soft}$
              if (Try Priority($\tau_t$, $\left\{\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}\right\} \setminus \{\tau_t\}$,n,soft)) then
                  $P(\tau_t) := n$
                  $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \setminus \{\tau_t\}$
              else
                  return Not Possible
      return List of $\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}$ ordered by $P(\tau_t)$

  procedure Try Priority($\tau_t, hp(\tau_t), priority, task\_type$)
      $P(\tau_t) := n$
      Assign $hp(\tau_t)$ to priorities $1, \dots, n - 1$
      if (task_type==hard) then
          $C_i := C_i^A, \ \forall \tau_i \in hp(\tau_t) \cup \tau_t$
      else
          $C_i := C_i^N, \ \forall \tau_i \in hp(\tau_t) \cup \tau_t$
      if ($\tau_t$ is schedulable according to TDA) then
          return true
      else
          return false
```

Since by means of the System with Dynamic Real-Time Guarantees model of von der Brüggen et al. [24] a possibility has been suggested to circumvent the weaknesses of common approaches to mixed-criticality scheduling, as manifested in 3, the main question of this thesis takes shape at this point, namely, if and how the concept of von der Brüggen et al. [24] can be transferred onto multiprocessor systems. For this purpose, the topic of multicore real-time scheduling will be thoroughly examined in the subsequent chapter (cf. 5), before novel approaches and solutions will be submitted in 6.

5. Multicore Fixed-Priority Scheduling

When turning towards the subject of multiprocessor scheduling, many aspects have to be taken into account. Regarding fixed-priority scheduling policies, not only a priority assignment has to be established, but, besides, the decision has to be made, which task to execute on which processor. Moreover, dependencies between different tasks or subtasks are obliged to be factored in when trying to solve this problem. Not least, the processors' properties and specifications must be included. However, in this thesis only homogeneous multiprocessor systems will be considered, i.e., all processors are identical and therefore exhibit the same execution rate. The contemplated task sets will, in general, not include any task dependencies, unless it is explicitly emphasized (cf. 6.3.1, 6.3.2).

In terms of multicore scheduling for real-time systems, three different approaches are commonly employed in academia as well as in the industry: global scheduling, partitioned scheduling and semi-partitioned scheduling. Applying the partitioned scheduling concept, two steps need to be performed, namely, the establishment of a partition of all given tasks onto a set of processors as well as the formation of a priority order, whereat each subset of tasks pertaining to one specific processor can be interpreted as an individual uniprocessor system. The global scheduling paradigm, in contrast, perceives the system en bloc, so that only one global queue of jobs that are ready for execution needs to be managed, whilst task instances are permitted to be preempted from one processor and to proceed in execution on another processor later on. Nevertheless, this strategy will not be adopted in this thesis, thus, further details will be omitted. Combining aspects of both partitioned and global scheduling, the semi-partitioned scheduling policy is a hybrid approach, which in the first instance allocates tasks to particular processors, but, nevertheless, allows a certain subset of tasks to migrate between processors.

Subsequently, the partitioned as well as the semi-partitioned scheduling approach will be expounded more precisely.

5.1. Partitioned Scheduling

When engaging with the partitioned scheduling approach for multicore systems, two distinct problems have to be solved. In the first place, it is necessary to partition the given task set into disjunct subsets, so that each task $\tau_i \in \mathbf{T}$ is permanently assigned to one specific processor which is in charge of its execution, since no migration of tasks between processors is permitted with respect to this concept. As a second step, a priority level has to be determined for each task in each subset of $\mathbf{T}$, which must not change prospectively due to the fact that a fixed-priority scheduling policy is considered.

The greatest benefit of applying partitioned scheduling is the fact that the multiprocessor scheduling problem is reduced into a set of uniprocessor scheduling problems as soon as the task partitioning stage is completed [10], but, nevertheless, the allocation of tasks to processors is proven to be NP-hard [14], since it is analogous to the bin-packing problem [10].

Based on the problem formulation by Lakshmanan et al. [19], a brief description of the classical bin-packing problem shall be given at this point. A set of n objects O_i with $1 \leq i \leq n, n \in \mathbb{N}$, where the size of each object is located in the range between 0 and 1, is supposed to be packed into a set of m bins B_j with $1 \leq j \leq m, m \in \mathbb{N}$, of capacity 1, subject to:

$$\forall\ j\ (1 \leq j \leq m) \sum_{\forall\ O_i \in B_j} S_i \leq 1$$

For a given bin-packing problem instance, the average size AS is defined as:

$$AS = \frac{1}{m} \sum_{i=1}^{n} S_i$$

Since the worst-case size bound SB is formed by the greatest lower bound on the average size of all unsolvable bin-packing problems, there exists a solution for a bin-packing problem instance with $AS \leq SB$ [19].

As a method of resolution to partitioned scheduling in terms of implicit-deadline task sets, it is a frequent practice in academia to apply approximation algorithms in the shape of bin-packing heuristics such as *First-Fit* (FF), *Arbitrary-Fit* (AF), *Best-Fit* (BF), *Worst-Fit* (WF), or *Decreasing Utilization* (DU) etc. in combination with a rate-monotonic priority order on each processor, as gathered and discussed in the survey by Davis and Burns [10]. To solve the problem of scheduling constrained-deadline sporadic task sets, i.e., $D_i \leq T_i\ \forall\ \tau_i \in \mathbf{T}$, on the other hand, the *deadline-monotonic partitioning* algorithm introduced by Baruah and Fisher [4] is well-known, which is a polynomial-time algorithm derivated from the First-Fit heuristic [17] for bin-packing [4], applicable for fixed-priority scheduling provided that an appropriate schedulability test is used [8].

In the course of deadline-monotonic partitioning, the tasks of task set $\mathbf{T}$ will be preliminarily assumed to be ordered non-decreasingly with regard to their relative deadlines, i.e., $D_i \leq D_{i+1}\ \forall\ i$ with $1 \leq i \leq m$ and $m = |\mathbf{T}|$. Once such a deadline-monotonic order is constructed, each task $\tau_i \in \mathbf{T}$ is assigned to the first processor that satisfies the respective schedulability condition [4]. If the schedulability test does not hold for any processor, the task set is identified to be not schedulable on the particular multicore system. By way of illustration, the algorithm is portrayed in pseudo-code (the notation is aligned with the general notation used in this thesis) in Algorithm 3 according to [4], not putting the schedulability condition into concrete terms, since the issue of a suitable schedulability test will be discussed in 6.2.1.

Algorithm 3 Deadline-Monotonic Partitioning [4]

```
Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$
Output: PARTITIONING SUCCEEDED or PARTITIONING FAILED
  procedure PARTITION($\mathbf{T}, p$) The collection of sporadic tasks $\mathbf{T} = \{\tau_1, \ldots, \tau_n\}$ is to be partitioned on
  $m$ identical, unit-capacity processors (i.e., each processor can be modeled as a bin of capacity 1). (Tasks
  are indexed according to non-decreasing value of relative deadline parameters $D_i \leq D_{i+1}$ for all $i$.) $\mathbf{T}_p$
  denotes the tasks assigned to processor $p$; initially $\mathbf{T}_p := \emptyset$ for all $p$.
      for ($i = 1;\ i \leq n;\ i := i + 1$) do
          $i$ ranges over the tasks, which are indexed by non-decreasing value of the deadline parameter
          for ($p = 1;\ p \leq m;\ p := p + 1$) do
              $p$ ranges over the processors, considered in any order
              if ($\tau_i$ satisfies the schedulability condition) then
                  assign $\tau_i$ to $\mathbf{T}_p$; proceed to next task
                  $\mathbf{T}_p = \mathbf{T}_p \cup \{\tau_i\}$ break
          if ($p > m$) then return PARTITIONING FAILED
      return PARTITIONING SUCCEEDED
```

By virtue of the fact that implicit-deadline task sets form a subset of constrained-deadline task sets, only the latter will be considered concerning the development of a System with Dynamic Real-Time Guarantees model for multicore systems under a partitioned approach (cf. 6.2).

5.2. Semi-Partitioned Scheduling

After having touched upon the partitioned scheduling approach for multiprocessor systems, a hybrid concept, the so-called semi-partitioned scheduling, shall be addressed, by means of which variegated aspects pertaining to global as well as to partitioned scheduling are combined. Analogously to the partitioned paradigm, the semi-partitioned method requires a partitioning of the task set into disjunct subsets, which are allocated to certain processors. In contrast to the fully partitioned scheme, this hybrid approach does not entirely interdict task migration, but permits a specified amount of tasks to migrate between processors.

In terms of task migration, also the notion of *task splitting* is used frequently, since what is commonly understood as a splitting operation is nothing else than suspending the execution of a task on one processor and resuming it on another one. This action can be performed several times, until the task is finalized. A majority of research results proposes strategies which pursue this strategy such as the paper of Kato and Yamasaki [18], who introduce the *Deadline Monotonic with Priority Migration (DM-PM)* algorithm, which shall be briefly sketched.

When applying the Deadline Monotonic with Priority Migration algorithm, the attempt is made to partition the given task set $\mathbf{T}$ by the use of bin-packing heuristics (cf. 5.1). However, if for one task τ_i no suitable processor can be found, the task set is not declared unfeasible, but instead is shared between multiple processors. The splitting points are chosen in such a manner that each processor receiving a subtask of τ_i has no spare capacity, except possibly that one obtaining the last part of the task. Under any circumstances, it is necessary to ensure that no task is executed by more than one processor at the same time. For this reason, Kato and Yamasaki imposed particular conditions: First, a shared task is obliged to be scheduled on the highest priority level on each processor. Second, every instance of a shared task is required to be released on the processor with lowest index and should migrate successively to the next core as soon as the processor's execution capacity is exhausted. All regular tasks, i.e., such tasks that do not migrate between disparate processors but are strictly allocated, are scheduled in compliance with the deadline-monotonic scheduling policy [18].

Apart from that, an enhancement of the DM-PM algorithm was introduced by Lakshmanan et al. [19], namely, the *Partitioned Deadline-Monotonic Scheduling under deadline-monotonic priority assignments, when used with Highest-Priority Task-Splitting*, short *(PDMS_HPTS)*. In contrast to DM-PM, when no appropriate processor can be detected for a task τ_i, not τ_i itself but the considered processor's highest-priority is split, i.e., distributed across several processors, such that τ_i can be allocated to the respective core. Moreover, an already shared task is permitted to be split again if it is executed on the actual processor's highest priority level [19].

The greatest benefit of the illustrated approaches is the increase in acceptance rates of task sets in contrast to fully partitioned strategies due to the matter of fact that by means of task splitting the fragmentation, which might emerge as a consequence of employing bin-packing heuristics, is reduced and the processors' capacity is seized as far as possible. Nevertheless, besides the motivation to share a task which cannot be completely affiliated by any processor, there exists another kind of task migration, as eludicated by Xu and Burns [25], who propose to migrate a certain amount of tasks from one processor to another if a criticality mode change occurs on the contemplated core. More precisely, Xu and Burns consider two different worst-case execution times for each task, $C_i(LO)$ and $C_i(HI)$ (cf. 3.1), whereat low-criticality tasks are never permitted to overshoot their respective $C_i(LO)$. However, as soon as a high-criticality task exceeds its $C_i(LO)$, the criticality mode is increased and all low-criticality migrate to another processor, subject to the permanent condition that all tasks, regardless of their criticality, remain schedulable [25].

Both strategies outlined above will be considered in this thesis, though targeting different objectives. After attempting to establish a multiprocessor version of the System with Dynamic Real-Time Guarantees model employing partitioned scheduling strategies, the aim will be pursued to enlarge the number of accepted task sets by sharing task across processors (cf. 6.3.1, 6.3.2). To follow another intention, the paradigm of Xu and Burns will be adopted thereafter, firstly, to create a failure-proof version of the multicore variant of a System with Dynamic Real-Time Guarantees (cf. 6.3.3), and secondly, to reduce the bounded tardiness of timing tolerable tasks (cf. 4) if for any reason all tasks on one or more processors are executed in abnormal mode (cf. 6.3.4).

6. Multicore Systems with Dynamic Real-Time Guarantees

After having provided the fundamentals of real-time systems in general (cf. 2) as well as of mixed-criticality systems in particular (cf. 3), having depicted the System with Dynamic Real-Time Guarantees model as proposed by von der Brüggen et al. [24] (cf. 4) and having introduced the two main approaches to multiprocessor fixed priority scheduling (cf. 5), an attempt shall be made hereinafter to transfer the System with Dynamic Real-Time Guarantees paradigm onto a multiprocessor environment, taking into consideration the previously discussed strategies. In order to attain this objective, the system model of a so-called *Multicore System with Dynamic Real-Time Guarantees* will be defined subsequently (cf. 6.1) with the intention to develop a guiding principle which to fulfill shall be the aim of ensuing deliberations. Hence, in the first place, the topic of partitioned scheduling will be addressed (cf. 6.2), whereat several bin-packing strategies will be presented by means of which a Multicore System with Dynamic Real-Time Guarantees can be established. Thereafter, to provide the opportunity to feasibly schedule more task sets than by the use of outright partitioned methods, these approaches will be enhanced by enabling task splitting, availing the concept of Kato and Yamasaki [18] (cf. 6.3.1) as well as that of Lakshmanan et al. [19] (cf. 6.3.2). Furthermore, for the purpose of granting failure safety for a respective system, the strategy of Xu and Burns [25] will be pursued (cf. 6.3.3) permanently migrating all tasks from one or more defective cores to the remaining functional processors. Additionally, in order to reduce the tardiness of a certain subset of timing tolerable tasks on a processor exhibiting completely erroneous behavior, e.g. due to electromagnetic interference, a temporary migration strategy is discussed (cf. 6.3.4). Not least, the offered concepts will be assessed and verified in practice by illuminating the results of comprehensive experiments in the next chapter (cf. 7).

6.1. System Model

Henceforth, the Multicore System with Dynamic Real-Time Guarantees model will be introduced, whereat a homogeneous multiprocessor environment will be considered, i.e., all processors pertaining to the particular system are identical and therefore exhibit the same execution rate. The contemplated task set will be assumed to have constrained deadlines, since implicit-deadline task sets form a subset of constrained-deadline task sets, and to evince no dependencies between tasks unless it is explicitly emphasized. Moreover, analogously to the System with Dynamic Real-Time Guarantees approach by von der Brüggen et al. [24], only preemptive fixed-priority scheduling policies will be taken into account.

With respect to the system model, the task partitioning as well as the specific priority order regarding each particular task subset (also termed *subsystem*) $\mathbf{T}_p$ is assumed to be given at this juncture.

Definition 3 (Multicore System with Dynamic Real-Time Guarantees).
Consider a set of tasks $\mathbf{T} = \tau_1, \ldots, \tau_n$ with $n \in \mathbb{N}$, partitioned to a set of m homogeneous processors with $m \in \mathbb{N}$, whereas the set of tasks $\mathbf{T}_p$ with $0 < p \leq m$ pertaining to each processor is sorted according to a fixed-priority order P_p. A job of task $\tau_i \in \mathbf{T}_p$ with $0 < i \leq n$ cannot begin its execution until all jobs of task τ_i that arrived earlier are completed. Furthermore, the jobs of all tasks always have to be finalized and must never be aborted.

Each task set $\mathbf{T}_p$ allocated to a processor with index p forms an individual System with Dynamic Real-Time Guarantees called *Subsystem of a Multicore System with Dynamic Real-Time Guarantees* or short *subsystem.*

A system $\mathbf{S}$ with subsystems $\mathbf{T}_p$ is a *Multicore System with Dynamic Real-Time Guarantees* if and only if each subsystem $\mathbf{T}_p$ satisfies the characteristics of a System with Dynamic Real-Time Guarantees at any point of time.

If a subsystem $\mathbf{T}_p$ runs with *full timing guarantees*, then hard real-time guarantees hold for each task allocated to a processor with index p:

- $\mathbf{T}_p$: Each task $\tau_i \in \mathbf{T}$ assigned to a processor with index p is obliged to meet its hard relative deadline.

If a subsystem $\mathbf{T}_p$ runs with *limited timing guarantees*, the service level guarantees are downgraded from hard real-time guarantees to bounded tardiness for some of the tasks:

- $\mathbf{T}^A_{p,hard} \subseteq \mathbf{T}_p$: Each task $\tau_i \in \mathbf{T}^A_{p,hard}$ assigned to a processor with index p is required to meet its hard relative deadline.
- $\mathbf{T}^A_{p,soft} \subseteq \mathbf{T}$: Each task $\tau_i \in \mathbf{T}^A_{p,soft}$ assigned to a processor with index p must have bounded tardiness, i.e., $0 \leq E_i < \gamma$ for some fixed value γ.

Each task in $\mathbf{T}$ has to be placed either in $\mathbf{T}^A_{hard}$ or in $\mathbf{T}^A_{soft}$, thus $\mathbf{T}^A_{hard} \cap \mathbf{T}^A_{soft} = \emptyset$ and $\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft} = \mathbf{T}$. Furthermore, each task in $\mathbf{T}$ has to be assigned to exactly one processor with index p, hence $\mathbf{T}_1 \cap \mathbf{T}_2 \cap \cdots \cap \mathbf{T}_p = \emptyset$ and $\mathbf{T}_1 \cup \mathbf{T}_2 \cup \cdots \cup \mathbf{T}_p = \mathbf{T}$.

If the system $\mathbf{S}$ runs with *full timing guarantees*, then every subsystem $\mathbf{T}_p$ runs with *full timing guarantees.*

If the system $\mathbf{S}$ runs with *limited timing guarantees*, then at least one subsystem $\mathbf{T}_p$ runs with *limited timing guarantees.*

6.2. Partitioned Approach

Since the definition of a Multicore System with Dynamic Real-Time Guarantees is given above, on this occasion the question shall be answered, how to establish a Multicore System with Dynamic Real-Time Guarantees. On that score, a number of algorithms will be presented hereinafter, partially availing classical bin-packing heuristics and partially modifying suchlike. Before turning to the actual most challenging, NP-hard problem of task partitioning, the issue of finding a suitable schedulability test will be discussed. Thereafter, the subject of constructing a priority order for each particular subsystem will be raised, due to the fact that this course of action will be identical with regard to each particular partitioned scheduling algorithm.

6.2.1. Schedulability Test

Before making the actual task partitioning a subject of discussion, a closer look shall be taken at how to determine if a task set $\mathbf{T}$ is schedulable, owing to the fact that this examination will play a not negligible role when deciding if a specific task τ_i is assigned to a certain processor with index p and thus integrated into a subsystem $\mathbf{T}_p$ or not.

Since a Multicore System with Dynamic Real-Time Guarantees consists of several subsystems satisfying the characteristics of a System with Dynamic Real-Time Guarantees, there is no need to apply a global schedulability test encompassing the union of all processors with their respective task sets $\mathbf{T}_1 \cup \mathbf{T}_2 \cup \cdots \cup \mathbf{T}_p = \mathbf{T}$, but it is rather possible to adopt the schedulability test presented by von der Brüggen et al. [24] (cf. 4.3) in such a manner that it is employed with respect to each processor separately. This property of a Multicore System with Dynamic Real-Time Guarantees permits to establish the following theorem.

Theorem 2 (Exact Schedulability Test for Multicore Systems with Dynamic Real-Time Guarantees with Constrained Deadlines).
For a given partition of a task set $\boldsymbol{T}$ into subsystems $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_m$ of $\boldsymbol{S}$ with $m \in \mathbb{N}$, $\boldsymbol{S}$ is a Multicore System with Dynamic Real-Time Guarantees defined in 3 if the following four conditions hold:

1. *Full timing guarantees hold for a subsystem $\boldsymbol{T}_p$ if it can be scheduled according to Time Demand Analysis (TDA) [20] when all tasks are executed in the normal mode, i.e., $C_i = C_i^N \ \forall \ \tau_i \in \boldsymbol{T}_p$ with $0 < i \leq n, 0 < p \leq m$.*
2. *Full timing guarantees hold for $\boldsymbol{S}$ if full timing guarantees hold for each subsystem $\boldsymbol{T}_p$ with $0 < p \leq m$.*
3. *When a subsystem $\boldsymbol{T}_p$ runs with limited timing guarantees, all $\tau_i \in \boldsymbol{T}_{p,hard}$ will meet their hard deadlines if they can be proven to be schedulable by TDA [20] when all tasks are executed in the abnormal mode, i.e., $C_i = C_i^A \ \forall \ \tau_i \in \boldsymbol{T}_p$.*

4. *When the Multicore System with Dynamic Real-Time Guarantees $\boldsymbol{S}$ runs with limited timing guarantees, at least one subsystem $\boldsymbol{T}_p$ and at most all subsystems $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_m$ run with limited timing guarantees. In this case, all tasks of all subsystems running with full timing guarantees, if existent, meet their deadlines, whereas all tasks $\tau_i \in \boldsymbol{T}_{p,hard}$ of all subsystems $\boldsymbol{T}_p$ running with limited timing guarantees meet their hard deadlines.*
5. *Each task $\tau_i \in \boldsymbol{T}_{p,soft}$ of such subsystems $\boldsymbol{T}_p$ running with limited timing guarantees has bounded tardiness if $U^A_{p,sum} \leq 1$.*

The proof makes use of the proof of 1 provided by von der Brüggen et al. [24] due to the fact that a Multicore System with Dynamic Real-Time Guarantees $\mathbf{S}$ is composed by a set of Systems with Dynamic Real-Time Guarantees $\mathbf{T}_p$ with $0 < p \leq m, m \in \mathbb{N}$.

Proof.

1. Since a subsystem $\mathbf{T}_p$ equals a System with Dynamic Real-Time Guarantees, this follows directly from the proof of 1 by von der Brüggen et al. [24].
2. This follows directly.
3. Since a subsystem $\mathbf{T}_p$ equals a System with Dynamic Real-Time Guarantees, this follows directly from the proof of 1 by von der Brüggen et al. [24].
4. This follows directly.
5. Since a subsystem $\mathbf{T}_p$ equals a System with Dynamic Real-Time Guarantees, this follows directly from the proof of 1 by von der Brüggen et al. [24]. □

Analogously to the approach of von der Brüggen et al. [24] with respect to Systems with Dynamic Real-Time Guarantees, regarding Multicore Systems with Dynamic Real-Time Guarantees, a subsystem utilization of of $U^A_{p,sum} > 1$ in case of fault occurrence is conceded if $U^N_{p,sum} < 1$ and if the duration of erroneous execution intervals is significantly shorter than that of the intervals evincing fault-free execution.

6.2.2. Priority Order

After having introduced the appropriated system model of a Multicore System with Dynamic Real-Time Guarantees, the establishment of a priority order concerning all $\tau_i \in \mathbf{T}$ shall be studied more closely. Due to the fact that the multiprocessor scheduling problem is reduced into a set of uniprocessor scheduling problems once the task allocation is completed (cf. 5.1), an individual priority order P_p can be defined for each subsystem $\mathbf{T}_p$. Since by means of the definition of a Multicore System with Dynamic Real-Time Guarantees (cf. 3) each subsystem $\mathbf{T}_p$ satisfies the characteristics of a System with Dynamic Real-Time Guarantees anytime, the Feasible Priority Assignment Algorithm by von der

Brüggen et al. [24] (cf. Algorithm 2) is applicable to all subsystems successively. Thus, feasible fixed-priority orders $P_1, \ldots, P_m$ can, if existent, be retrieved for $T_1, \ldots, T_m$.

6.2.3. Classical Bin-Packing Heuristics

Being aware of the procedure to create a feasible priority assignment, it is necessary to engender the crucial prerequisites, namely, the partition of task set $\mathbf{T}$ into m subsystems $T_1, \ldots, T_m$. For this purpose, several partitioning strategies will be introduced in the following, beginning with classical bin-packing heuristics. *Classical* in this context is tantamount to simple, since the particular subsystem's task set in this event is contemplated in toto, whereby with respect to each approach only one fitting-strategy is applied. In the course of this, each algorithm is explained in detail and additionally portrayed in pseudo-code. Thereafter, enhanced bin-packing heuristics will be presented (cf. 6.2.4), with regard to which in some instances timing strict and timing tolerable tasks are considered separately and, furthermore, occasionally more than one heuristic is utilized.

Deadline-Monotonic First-Fit Scheduling (DM_FF)

The first and presumably most intuitive bin-packing heuristic to be addressed is the first-fit (FF) strategy, whose fundamental idea is to assign each task to the first suitable processor, i.e., to the first processor concerning which it satisfies the respective schedulability condition. Additionally, before commencing the actual allocation phase, the considered subsystem's tasks are preordered increasingly with respect to their particular deadlines, i.e., $\tau_i < \tau_j$ if $D_i < D_j$, analogously to the proceeding in terms of the deadline-monotonic partitioning algorithm by Baruah and Fisher [4] (cf. Algorithm 3). The applied schedulability condition on this occasion is the schedulability test specified above, which was, concerning uniprocessor systems, originally introduced by von der Brüggen et al. [24] (cf. 4.3).

For clarification, the deadline-monotonic first-fit scheduling algorithm is given as pseudo-code in Algorithm 4. In the procedure Find Assignment($\mathbf{T}$, $\mathbf{S}$) (cf. Algorithm 4, l. 3) an iteration over all $\tau_i \in \mathbf{T}$ is performed, in the course of which the schedulability condition is checked for each subsystem $\mathbf{T}_p$ plus the contemplated task (cf. Algorithm 4, l. 7). If the schedulability test holds, the task is assigned to the respective subsystem (cf. Algorithm 4, l. 8), otherwise the task set is declared as unfeasible (cf. Algorithm 4, l. 12). If all tasks have been successfully allocated to a subsystem $\mathbf{T}_p$, the system $\mathbf{S}$ i.e., the task partitioning as well as the respective priority orders for each subsystem, are returned (cf. Algorithm 4, l. 13).

Algorithm 4 Deadline-Monotonic First-Fit (DM_FF)

```
Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$, or NOT POSSIBLE

 1: Sort $\mathbf{T}$ by $D_i$ increasingly
 2: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

 3: procedure FIND ASSIGNMENT($\mathbf{T}$, $\mathbf{S}$)
 4:     for each $\tau_t \in \mathbf{T}$ do
 5:         $assigned := false$
 6:         for each $\mathbf{T}_p in \mathbf{S}$ do
 7:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
 8:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
 9:                 $\mathbf{T} := \mathbf{T} \setminus \{\tau_t\}$
10:                 $assigned := true$
11:                 break
12:         if ($assigned$ = false) then return NOT POSSIBLE
13:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

Deadline-Monotonic Arbitrary-Fit Scheduling (DM_AF)

The deadline-monotonic arbitrary-fit scheduling policy follows a similar idea as the deadline-monotonic first-fit scheduling policy (cf. 6.2.3). Likewise, the deadline-monotonic partitioning algorithm (cf. Algorithm 3) serves as a groundwork for this method, but accomplishing the modification that a task $\tau_i \in \mathbf{T}$ is not assigned to the first processor for which the schedulability test holds. In point of fact, the schedulability condition for $\tau_i \in \mathbf{T}$ in addition to a subsystem $\mathbf{T}_p$ is checked with respect to each single core, despite the outcome of earlier tests (cf. Algorithm 5, ll. 7-8). Thereupon, the respective processor to which τ_i is allocated is chosen randomly out of the set comprising all suitable processors (cf. Algorithm 5, ll. 10-12). If the schedulability condition did not hold for any subsystem $\mathbf{T}_p$, the task set is identified as unfeasible (cf. Algorithm 5, l. 13).

Algorithm 5 Deadline-Monotonic Arbitrary-Fit (DM_AF)

```
Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$, or NOT POSSIBLE

 1: Sort $\mathbf{T}$ by $D_i$ increasingly
 2: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

 3: procedure FIND ASSIGNMENT($\mathbf{T}$, $\mathbf{S}$)
 4:     for each $\tau_t \in \mathbf{T}$ do
 5:         $\bar{\mathbf{S}} := \emptyset$
 6:         for each $\mathbf{T}_p \in \mathbf{S}$ do
 7:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
 8:                 $\bar{\mathbf{S}} := \bar{\mathbf{S}} \cup \mathbf{T}_p$
 9:         if $\bar{\mathbf{S}} \neq \emptyset$ then
10:             $r :=$ random element out of $[1, |\bar{\mathbf{S}}|]$
11:             $\mathbf{T}_r := \mathbf{T}_r \cup \{\tau_t\}$
12:             $\mathbf{T} := \mathbf{T} \setminus \{\tau_t\}$
13:         else return NOT POSSIBLE
14:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

Deadline-Monotonic Best-Fit Scheduling (DM_BF)

The deadline-monotonic best-fit scheduling policy makes use of the same main principle and schedulability test as the concepts explained before (cf. 6.2.3, 6.2.3). At this point, the choice of the particular processor to which a task $\tau_i \in \mathbf{T}$ is attached in case the schedulability condition holds, is made with respect to a processor's total utilization $U^N_{p,sum} := \sum_{\tau_i \in \mathbf{T}_p} U^N_i$. More precisely, the processor out of all suitable processors according to the schedulability test which exhibits the maximum total utilization is selected to affiliate τ_i. This processor is retrieved by means of the procedure Find Processor With Max. Utilization($\mathbf{S}$, $\mathbf{S}^*$) (cf. Algorithm 6, ll. 18-25), which successively calculates the total utilization of each core available for the current task, i.e., each core which is not contained in $\mathbf{S}^*$, and returns the index of that one exhibiting the maximal load. If the schedulability test holds concerning this specific processor (cf. Algorithm 6, l. 11), it receives the contemplated task (cf. Algorithm 6, l. 11), otherwise the processor is excluded from the list of disposable processors as long as τ_i is not allocated to any subsystem by adding it to the dedicated set $\mathbf{S}^*$ (cf. Algorithm 6, l. 15). In this manner, namely, by steadily choosing the processor with the maximum load, the processors are successively filled to capacity. The exact procedure is depicted as pseudo-code in Algorithm 6.

Algorithm 6 Deadline-Monotonic Best-Fit (DM_BF)

Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or Not Possible

```
 1: Sort $\mathbf{T}$ by $D_i$ increasingly
 2: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

 3: procedure Find Assignment($\mathbf{T}$, $\mathbf{S}$)
 4:     for each $\tau_t \in \mathbf{T}$ do
 5:         $\mathbf{S}^* := \emptyset$
 6:         $assigned := false$
 7:         while ($assigned$ = false) do
 8:             $p :=$ Find Processor with Max. Utilization($\mathbf{S}$, $\mathbf{S}^*$)
 9:             if ( $p = -1$) then return Not Possible
10:             if (Find Optimal Priority Assignment($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ Not Possible) then
11:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
12:                 $\mathbf{T} := \mathbf{T} \setminus \{\tau_t\}$
13:                 $assigned$ := true
14:             else
15:                 $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
16:                 if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return Not Possible
17:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

18: procedure Find Processor with Max. Utilization($\mathbf{S}$, $\mathbf{S}^*$)
19:     $p_{best} := -1$
20:     $U_{sum,best} := 0$
21:     for ($p := 1$; $p \leq |\mathbf{S} \setminus \mathbf{S}^*|$; $p := p + 1$) do
22:         if ($U^N_{sum,p} \geq U_{sum,best}$) then
23:             $p_{best} := p$
24:             $U_{sum,best} := U^N_{sum,p}$
25:     return $p_{best}$
```

Deadline-Monotonic Worst-Fit Scheduling (DM_WF)

In contrast to the aforementioned deadline-monotonic best-fit scheduling policy, the deadline-monotonic worst-fit strategy operates in an antithetical fashion. After preordering the task set **T** increasingly with respect to the particular deadlines, the processor with the minimum total utilization $U^N_{p,sum} := \sum_{\tau_i \in \mathbf{T}_p} U^N_i$ is chosen by the procedure Find Processor With Min. Utilization(**S**, **S***) (cf. Algorithm 7, ll. 18-25) in order to assign a task $\tau_i \in \mathbf{T}$ and to achieve a balanced processor utilization. This method's detailed operating principle is visualized in Algorithm 7.

Algorithm 7 Deadline-Monotonic Worst-Fit (DM_WF)

```
Input: $\mathbf{T} := \tau_1, \dots, \tau_n$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \dots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$, or NOT POSSIBLE

 1: Sort $\mathbf{T}$ by $D_i$ increasingly
 2: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

 3: procedure FIND ASSIGNMENT($\mathbf{T}$, $\mathbf{S}$)
 4:     for each $\tau_t \in \mathbf{T}$ do
 5:         $\mathbf{S}^* := \emptyset$
 6:         $assigned := false$
 7:         while ($assigned$ = false) do
 8:             $p$ := FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
 9:             if ( $p = -1$) then return NOT POSSIBLE
10:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
11:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
12:                 $\mathbf{T} := \mathbf{T} \setminus \{\tau_t\}$
13:                 $assigned$ := true
14:             else
15:                 $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
16:                 if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
17:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

18: procedure FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
19:     $p_{best} := -1$
20:     $U_{sum,best} := 0$
21:     for ($p := 1;\ p \leq |\mathbf{S} \setminus \mathbf{S}^*|;\ p := p + 1$) do
22:         if ($U^N_{sum,p} \leq U_{sum,best}$) then
23:             $p_{best} := p$
24:             $U_{sum,best} := U^N_{sum,p}$
25:     return $p_{best}$
```

Decreasing Utilization First-Fit Scheduling (DU_FF)

Deviating from the basic deadline-monotonic partitioning approach of Baruah and Fisher [4], a utilization based variation modifying some of the preceding algorithms shall be suggested in the following. To begin with, the decreasing utilization first-fit scheduling policy is depicted. As the denotation suggests, this method follows the same principle as the deadline-monotonic first-fit scheduling strategy, despite the fact that the task set **T** is preordered decreasingly with regard to task utilization (cf. Algorithm 8, l. 1). This is justified by the attempt to distribute the workload more evenly onto the processors than it, by intuition, appears to be possible employing a deadline-monotonic task preordering. If this intuitive idea turns out to be correct and if more task sets prove to be schedulable owing to the reduction of fragmentation achieved by this approach, will be examined in 7. The exact algorithm is given in pseudo-code in Algorithm 8.

Algorithm 8 Decreasing Utilization First-Fit (DU_FF)

```
Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$
Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$, or NOT POSSIBLE

 1: Sort $\mathbf{T}$ by $U_i$ decreasingly
 2: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

 3: procedure FIND ASSIGNMENT($\mathbf{T}$, $\mathbf{S}$)
 4:     for each $\tau_t \in \mathbf{T}$ do
 5:         $assigned := false$
 6:         for each $\mathbf{T}_p \in \mathbf{S}$ do
 7:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
 8:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
 9:                 $\mathbf{T} := \mathbf{T} \backslash \{\tau_t\}$
10:                 $assigned := true$
11:                 break
12:         if ($assigned$ = false) then return NOT POSSIBLE
13:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

Decreasing Utilization Arbitrary-Fit Scheduling (DU_AF)

Pursuing the above approach, the decreasing utilization arbitrary-fit scheduling policy exhibits the same behavior as the deadline-monotonic arbitrary-fit strategy (cf. 6.2.3) except the fact that the task set **T** is preordered according to decreasing task utilization. By reason of the strong resemblance to Algorithm 5, the DU_AF algorithm is given in the appendix as Algorithm 14.

Decreasing Utilization Best-Fit Scheduling (DU_BF)

Analogously to the proceeding in 6.2.3 and 6.2.3, the decreasing utilization best-fit scheduling strategy requires to initially preorder the task set **T** in terms of decreasing task utilization and, in a second step, operates according to the best-fit bin-packing heuristic, that is, in the same way as the DM_BF policy, as shown in Algorithm 15.

Decreasing Utilization Worst-Fit Scheduling (DU_WF)

Likewise, the decreasing utilization worst-fit scheduling method proceeds analogically to the deadline-monotonic worst-fit scheduling policy (cf. 6.2.3), but requires, as the above approaches, a task preordering with respect to decreasing task utilization, aiming to fulfill the aforementioned objective. The algorithm is portrayed in pseudo-code in Algorithm 16.

6.2.4. Enhanced Bin-Packing Heuristics

After having touched upon standard bin-packing strategies and having sketched several algorithms combining these with a deadline-monotonic or decreasing utilization based task preorder, an enhancement of those shall be moved into the spotlight. In the following, a number of approaches will be suggested, which do not consider the task set $\mathbf{T}$ as a whole, but instead handle the timing strict and the timing tolerable task subsets successively, beginning with the timing strict tasks (this proceeding is termed *criticality-successive* in what follows). More specifically, both subsets are internally preordered either nondecreasingly with respect to their deadlines or regarding their utilization in a decreasing fashion. In the first instance, the set of timing strict tasks $\mathbf{T}^A_{hard}$ is allocated to the available processors, followed by the set of timing tolerable tasks $\mathbf{T}^A_{soft}$, whereat not necessarily the same fitting-strategy is applied to both subsets.

The underlying idea of this concept is to establish a different task allocation than it can be achieved by standard bin-packing heuristics in order to reduce the processor fragmentation. Since, in contrast to approaches attempting to distribute the task set en bloc, not only one iteration is performed in the course of the task assignment, it should be possible to retrieve a provisional task allocation after the iteration over $\mathbf{T}^A_{hard}$ and to fill in the timing tolerable tasks in a second iteration in such a manner that as little spare capacity as possible is left on each processor.

As a consequence, by intuition, more task sets could be accepted as by means of standard bin-packing strategies, i.e., even task sets evincing a higher system utilization than feasible by the use of the aforementioned policies could, in turn, be scheduled by these enhanced bin-packing heuristics. If this intuitive idea proves to be practicable, will be verified by experiments and discussed in 7 as soon as all paradigms aiming at the establishment of a Multicore System with Dynamic Real-Time Guarantees have been eludicated in theory.

Criticality-Successive Deadline-Monotonic First-Fit Scheduling (CS_DM_FF)

The criticality-successive deadline-monotonic first-fit scheduling strategy evinces a similar functionality as the deadline-monotonic first-fit scheduling policy, initially discussed in 6.2.3, with the sole difference that, in contrast to DM_FF, this approach does not consider the task set $\mathbf{T}$ as one unit but rather processes $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ successively. Initially, the set of timing strict tasks $\mathbf{T}^A_{hard}$ as well as the set of timing tolerable tasks $\mathbf{T}^A_{soft}$ are separately ordered with regard to their respective deadlines (cf. Algorithm 9, ll. 1-2). Thereon, two iterations about the particular task subsets are performed, namely, over $\mathbf{T}^A_{hard}$ (cf. Algorithm 9, ll. 5-13) and over $\mathbf{T}^A_{soft}$ (cf. Algorithm 9, ll. 14-22), whereat in either case nothing else than the actual deadline-monotonic first-fit heuristic, as introduced in 6.2.3, is applied. If both iterations terminate successfully, the task set is determined to be feasible under CS_DM_FF (cf. Algorithm 9, l. 23).

Algorithm 9 Criticality-Successive Deadline-Monotonic First-Fit (CS_DM_FF)

```
Input: $\mathbf{T}^A_{hard} = \tau_1, \dots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \dots, \tau_l$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \dots, \mathbf{T}_m$
Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$, or NOT POSSIBLE

1: Sort $\mathbf{T}^A_{hard}$ by $D_i$ increasingly
2: Sort $\mathbf{T}^A_{soft}$ by $D_i$ increasingly
3: Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

4: procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
5:     for each $\tau_t \in \mathbf{T}^A_{hard}$ do
6:         $assigned := false$
7:         for each $\mathbf{T}_p \in \mathbf{S}$ do
8:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
9:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
10:                $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
11:                $assigned := true$
12:                break
13:        if ($assigned$ = false) then return NOT POSSIBLE
14:    for each $\tau_t \in \mathbf{T}^A_{soft}$ do
15:        $assigned := false$
16:        for each $\mathbf{T}_p \in \mathbf{S}$ do
17:            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
18:                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
19:                $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
20:                $assigned := true$
21:                break
22:        if ($assigned$ = false) then return NOT POSSIBLE
23:    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

Criticality-Successive Deadline-Monotonic Arbitrary-Fit Scheduling (CS_DM_AF)

In analogous fashion to the previous approach, the criticality-successive deadline-monotonic arbitrary-fit scheduling policy does not treat the task set $\mathbf{T}$ in its entirety like the deadline-monotonic arbitrary-fit scheduling method discussed in 6.2.3, but contemplates $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ in a consecutive way, whereat in each case the deadline-monotonic arbitrary-fit policy, as explained in 6.2.3 is applied to the respective task subset. The exact way of operating is illustrated in pseudo-code in Algorithm 17.

Criticality-Successive Deadline-Monotonic Best-Fit Scheduling (CS_DM_BF)

The criticality-successive deadline-monotonic best-fit scheduling policy, analogously to the two strategies presented before (cf. 6.2.4,6.2.4), deals with $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ one by one instead of contemplating $\mathbf{T}$ in total, in the course of which the best-fit bin-packing heuristic (cf. 6.2.3) is applied to each subset successively. The concrete algorithm is given in Algorithm 18.

Criticality-Successive Deadline-Monotonic Worst-Fit Scheduling (CS_DM_WF)

As the strategies outlined above, the criticality-successive deadline-monotonic worst-fit scheduling policy considers timing strict and timing tolerable tasks in succession instead of handling $\mathbf{T}$ at once, whereby in either iteration the worst-fit bin-packing heuristic (cf. 6.2.3) is employed with respect to each particular task subset. The algorithm is sketched in pseudo-code in Algorithm 19.

Criticality-Successive Decreasing Utilization First-Fit Scheduling (CS_DU_FF)

In addition to the earlier introduced algorithms, also the utilization-based variation of the criticality-successive deadline-monotonic first-fit scheduling approach (cf. 6.2.4 is suggested, where $\mathbf{T}^A_{hard}$ as well as $\mathbf{T}^A_{soft}$ are not preordered in terms of their deadlines but with respect to the particular task utilization (cf. Algorithm 10, ll. 1-2). Apart from this only difference, the scheduling paradigm completely equals the criticality-successive deadline-monotonic first-fit scheduling policy delineated above, in the process of which the first-fit bin-packing strategy is applied consecutively to $\mathbf{T}^A_{hard}$ (cf. Algorithm 10, ll. 5-13) as well as to $\mathbf{T}^A_{soft}$ (cf. Algorithm 10, ll. 14-22).

Algorithm 10 Criticality-Successive Decreasing Utilization First-Fit (CS_DU_FF)

Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
1: Sort $\mathbf{T}^A_{hard}$ by $U_i$ decreasingly
2: Sort $\mathbf{T}^A_{soft}$ by $U_i$ decreasingly
3: Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
4: procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
5:     for each $\tau_t \in \mathbf{T}^A_{hard}$ do
6:         $assigned := false$
7:         for each $\mathbf{T}_p \in \mathbf{S}$ do
8:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
9:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
10:                $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
11:                $assigned := true$
12:                break
13:        if ($assigned$ = false) then return NOT POSSIBLE
14:    for each $\tau_t \in \mathbf{T}^A_{soft}$ do
15:        $assigned := false$
16:        for each $\mathbf{T}_p \in \mathbf{S}$ do
17:            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
18:                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
19:                $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
20:                $assigned := true$
21:                break
22:        if ($assigned$ = false) then return NOT POSSIBLE
23:    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

Criticality-Successive Decreasing Utilization Arbitrary-Fit Scheduling (CS_DU_AF)

The criticality-successive decreasing utilization arbitrary-fit scheduling policy operates analogously to its deadline-monotonic pendant (cf. 6.2.4), except the fact that timing strict and timing tolerable tasks are preordered according to decreasing task utilization, as apparent from Algorithm 20.

Criticality-Successive Decreasing Utilization Best-Fit Scheduling (CS_DU_BF)

Likewise, the criticality-successive decreasing utilization best-fit scheduling method exhibits a similar behavior to the deadline-monotonic version, with the sole difference that the preordering of $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ is performed on the basis of decreasing task utilization, as shown in Algorithm 21.

Criticality-Successive Decreasing Utilization Worst-Fit Scheduling (CS_DU_WF)

The alternative version to the criticality-successive deadline-monotonic worst-fit scheduling approach, namely, the variant employing a task preorder with respect to decreasing utilization, is illustrated in Algorithm 22 and, apart from that, operates in the same fashion as Algorithm 19.

Deadline-Monotonic Hard Best-Fit, Soft Worst-Fit Scheduling (DM_HBF_SWF)

Unlike the approaches suggested before, in the subsequent methods disparate fitting-strategies are applied to each subset of $\mathbf{T}$with the intention to utilize the cores evenly and to diminish fragmentation. With respect to the deadline-monotonic hard best-fit, soft worst-fit scheduling policy, the timing-strict tasks are allocated availing the best-fit bin-packing heuristic (cf. Algorithm 11, ll. 5-18) while the timing-tolerable tasks are distributed by means of the worst-fit bin-packing heuristic (cf. Algorithm 11, ll. 19-32), whereat both $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ are internally preordered according to the tasks' respective deadlines.

Deadline-Monotonic Hard Worst-Fit, Soft Best-Fit Scheduling (DM_HWF_SBF)

The deadline-monotonic hard worst-fit, soft best-fit scheduling paradigm exhibits the same underlying principle and the same purpose as the aforementioned strategy, albeit the usurpated bin-packing heuristics are applied in reverse order, as outlined in Algorithm 23.

Decreasing Utilization Hard Best-Fit, Soft Worst-Fit Scheduling (DU_HBF_SWF)

The decreasing utilization hard best-fit, soft worst-fit scheduling policy operates analogously to the deadline-monotonic version (cf. 6.2.4), with the sole difference that the timing-strict as well as the timing-tolerable tasks are internally ordered in terms of decreasing task utilization. The algorithm is portrayed in pseudo-code in Algorithm 24.

Algorithm 11 Deadline-Monotonic Hard Best-Fit, Soft Worst-Fit (DM_HBF_SWF)

Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
1: Sort $\mathbf{T}^A_{hard}$ by $D_i$ increasingly
2: Sort $\mathbf{T}^A_{soft}$ by $D_i$ increasingly
3: Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

4: procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
5:     for each $\tau_t \in \mathbf{T}^A_{hard}$ do
6:         $\mathbf{S}^* := \emptyset$
7:         $assigned := false$
8:         while ($assigned$ = false) do
9:             $p$ := FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
10:            if ( $p = -1$) then return NOT POSSIBLE
11:            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
12:                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
13:                $\mathbf{T} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
14:                $assigned$ := true
15:            else
16:                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
17:                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
18:    for each $\tau_t \in \mathbf{T}^A_{soft}$ do
19:        $\mathbf{S}^* := \emptyset$
20:        $assigned := false$
21:        while ($assigned$ = false) do
22:            $p$ := FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
23:            if ( $p = -1$) then return NOT POSSIBLE
24:            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
25:                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
26:                $\mathbf{T} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
27:                $assigned$ := true
28:            else
29:                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
30:                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
31:    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

32: procedure FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
33:    $p_{best} := -1$
34:    $U_{sum,best} := 0$
35:    for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
36:        if ($U^N_{sum,p} \leq U_{sum,best}$) then
37:            $p_{best} := p$
38:            $U_{sum,best} := U^N_{sum,p}$
39:    return $p_{best}$

40: procedure FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
41:    $p_{best} := -1$
42:    $U_{sum,best} := 0$
43:    for ($p = 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
44:        if ($U^N_{sum,p} \geq U_{sum,best}$) then
45:            $p_best := p$
46:            $U_{sum,best} := U^N_{sum,p}$
47:    return $p_{best}$
```

Decreasing Utilization Hard Worst-Fit, Soft Best-Fit Scheduling (DU_HWF_SBF)

Relatedly, the decreasing utilization hard worst-fit, soft best-fit scheduling policy handling tasks $\tau_i \in \mathbf{T}^A_{hard}$ according to the worst-fit paradigm and $\tau_i \in \mathbf{T}^A_{soft}$ in compliance with the best-best fit method, whereat $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ are internally preordered at the sight of decreasing task utilization. The particular proceeding is clarified in Algorithm 25.

6.2.5. Limits of Partitioned Scheduling

Having introduced manifold algorithms by dint of which the bin-packing problem, namely, the allocation of tasks to processors, can be solved as well as for each bin, respectively each subsystem, a feasible priority assignment can be established and thus a Multicore System with Dynamic Real-Time Guarantees is shaped, the question arises, how effective the aforementioned methods turn out to be.

Considering the underlying bin-packing problem, it is evident that in case one bin cannot affiliate any of the remaining objects, nevertheless the possibility exists that some spare capacity is left. In other words, if no further task can be allocated to a processor, it is anyway possible that some processor capacity is unused. However, in some cases such tasks could be assigned to processors with disposable capacity regardless, if they were not prohibited to be split by specification.

Thus, tasks are conceded to be disassembled into multiple subtasks hereinafter, which turn to account residual processor resources. Accordingly, each core is attempted to be filled to maximum by assigning shares of not yet allocated tasks. In the course of this, it is possible to schedule task sets exhibiting a higher system utilization than schedulable by merely partitioned strategies. This approach constitutes the first half of the subsequent section contemplating the so-called semi-partitioned scheduling (cf. 5.2).

6.3. Semi-Partitioned Approach

As alluded to in the preceding section, it is desirable to permit tasks to be shared between multiple processors in order to be able to accept more task sets than by use of fully partitioned scheduling approaches. This concept shall be followed subsequently, commencing with an adaption of the task splitting paradigm introduced by Kato and Yamasaki [18] (cf. 6.3.1) applicable to Multicore Systems with Dynamic Real-Time Guarantees, before modifying it according to the highest-priority task splitting strategy by Lakshmanan et al. [19]. Thereon, pursuing a different objective, a task migration approach suggested by Xu and Burns [25] (cf. 6.3.4) will be addressed in order to create failure-proof Multicore Systems with Dynamic Real-Time Guarantees on the one hand and to reduce the bounded tardiness with respect to a certain amount of timing tolerable tasks on the other hand, contemplating such cases in which all tasks on one or more processors execute abnormally for some reason.

6.3.1. Task Splitting

Splitting tasks and sharing them across processors provides a broad range of benefits, such as improved acceptance rates for task sets which could not be scheduled properly as a Multicore System with Dynamic Real-Time Guarantees by means of partitioned scheduling strategies. Nevertheless, this idea is not merely advantageous, but also bears many challenges and requires additional considerations. In particular, it must be determined, which tasks to share, how to compute a subtask's worst-case execution time, how to specify its deadline, which priority to assign it and how to ensure the subtasks' correct execution order. These questions will be answered in the following, based on the findings by Kato and Yamasaki [18].

As apparent from the partitioned scheduling approaches suggested above (cf. 5.1), a task set $\mathbf{T}$ is declared to be unschedulable as a Multicore System with Dynamic Real-Time Guarantees $\mathbf{S}$ on a set of m processors, if for any task $\tau_s \in \mathbf{T}$ the schedulability test defined in Theorem 2 does not hold. If this is the case with respect to a certain task τ_s, the attempt is made to share τ_s between at least two processors such that, as noted previously (cf. 5.2), τ_s is not executed by more than one processor at the same time and such that the subtasks $\tau_{s,1}, \ldots, \tau_{s,p}$ with $0 < p \leq m$, of τ_s are completed in proper sequence.

When a task τ_s is split, resulting from this, each processor maintaining one share of τ_s is filled to maximum capacity, except the last one which may evince a certain amount of unused execution capacity. In this regard, it has to be taken into account that after τ_s exhausted the resource provided by a processor with index p, it is successively passed on to the processor with index $p + 1$, in case the schedulability condition holds, and so forth. Hence, each instance of a shared task is assigned the highest priority in the respective processor's task set $\mathbf{T}_p$, whereat, in case a processor already holds a shared task, namely, the last subtask of a task τ_j, the later assigned subtask, i.e. τ_s, receives the highest priority, while the earlier assigned shared task, i.e. τ_j, is relegated to the second place. Hereinafter, a precise definition of a shared task is given.

Definition 4 (Shared Task).
A task $\tau_s \in \mathbf{T}$ is a *shared task* if its execution is not restricted to one processor and if the following conditions hold:

- A shared task is never executed on more than one processor at the same time, but is successively passed on to the next core as soon as its execution budget on the considered processor is exhausted.

- A shared task is always scheduled under the highest priority on each processor, except another shared task has already been assigned to the respective core. In this case, the later allocated task is preferred in terms of priority.
- Concerning all calculations involving the shared task's worst-execution time, C_s^A is considered regardless of the actual execution mode.

Each share of τ_s, denoted as *subtask*, can be treated as an individual task $\tau_{s,p} = (C_{s,p}, T_s, D_s)$.

Since it has been clarified how to distribute a shared task between processors and under which priority to schedule it, henceforth shall be explained how to determine each subtask's $\tau_{s,p}$ worst-case execution time $C_{s,p}^A$. For this purpose, response time analysis is employed, which is explained in the following based on the results by Kato and Yamasaki [18].

Assume that two tasks τ_i and τ_j, with τ_i possessing a lower priority than τ_j, are released at a critical instant t_0, i.e., at the same time, due to the fact that the response times R_i and R_j are maximal in this case, as shown by Liu and Layland [21]. Resulting from this, τ_i can be blocked by τ_j for a certain amount of time, denoted $B_{i,j}(D_i)$, during a contemplated time window of length D_i. Moreover, the maximum number of jobs of τ_j which are entirely executed in this same interval is identified as $F = \left\lfloor \frac{D_i}{T_j} \right\rfloor$. To compute the blocking time $B_{i,j}(D_i)$ which τ_i suffers from τ_j, two different cases have to be taken into account, namely, $D_i \geq F \cdot T_j + C_j$, as illustrated in Figure 6.1, and $D_i \leq F \cdot T_j + C_j$, as outlined in Figure 6.2, whereat $C_j = C_j^A$. In the first case, $B_{i,j}(D_i)$ is given by

$$B_{i,j}(D_i) = F \cdot C_j^A + C_j^A = (F+1) \cdot C_j^A,$$

whereas in the second case by

$$B_{i,j}(D_i) = D_i - F \cdot (T_j - C_j^A).$$

In consequence, the worst-case response time $R_{i,p}$ of each task τ_i allocated to a processor with index p is obtained by:

$$R_{i,p} = \sum_{\tau_j \in \mathbf{T}_p \cap hp(\tau_i)} B_{i,j}(D_i) + C_i^A$$

Henceforth, let $\tau_s \in \mathbf{T}_p$ be a shared task whose time of competition with another task $\tau_i \in \mathbf{T}_p$ during a time interval of length D_i can be retrieved by:

$$W_{s,p}(D_i) = \left\lceil \frac{D_i}{T_s} \right\rceil \cdot C_{s,p}^A$$

The choice of $C_j = C_j^A$ is justified by a simple deliberation. Let τ_s be a shared task. With respect to the value of C_j, a case discrimination could be made: C_j^A could be chosen in the course of the calculation of $C_{s,p}^A$ if $\tau_j \in \mathbf{T}_{hard}^A$, and C_j^N if $\tau_j \in \mathbf{T}_{soft}^A$. By this means, a result for $C^A_{s,p}$ could be retrieved such that hard real-time guarantees hold for $\tau_{s,p}$ in normal as well as in abnormal mode if $\tau_s \in \mathbf{T}^A_{hard}$, and, furthermore, such that τ_s meets its deadline under normal execution and has bounded tardiness otherwise if $\tau_s \in \mathbf{T}^A_{soft}$. However, since $\tau_{s,p}$ is scheduled on the highest priority level by definition (in case more than one shared task has been assigned to respective processor, $\tau_{s,p}$ shall be considered as the highest-priority shared task at this juncture), $\tau_{s,p} \in \mathbf{T}^A_{soft}$ is permitted to miss its deadline when executed abnormally in terms of the definition of a Multicore System with Dynamic Real-Time Guarantees (cf. 3), but, notwithstanding, this tardiness may cause another task's deadline miss (cf. the case depicted in Figure 1.2). To avoid such incidents, $C_j = C^A_j$ is chosen regardless of a shared task's membership in either the timing strict or the timing tolerable subset of $\mathbf{T}$.

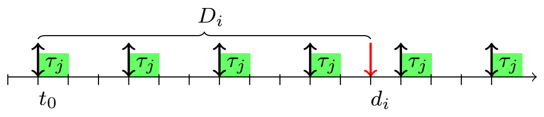

Figure 6.1.: $D_i \geq F \cdot T_j + C_j$ [18]

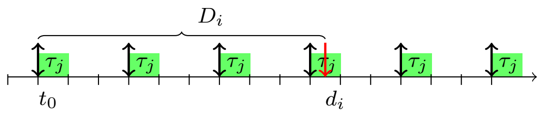

Figure 6.2.: $D_i \leq F \cdot T_j + C_j$ [18]

In order to comply with all tasks' timing requirements, the following condition must hold for each $\tau_i \in \mathbf{T}_p$ if a shared task τ_s is allocated to a processor with index p:

$$R^A_{i,p} + W_{s,p}(D_i) \leq D_i$$

$$\Leftrightarrow C^A_{s,p} \leq \frac{D_i - R^A_{i,p}}{\left\lceil \frac{D_i}{T_s} \right\rceil}$$

Hence, $C^A_{s,p}$ can eventually be derived by:

$$C^A_{s,p} = \min_{\tau_i \in \mathbf{T}_p} \left\{ \frac{D_i - R^A_{i,p}}{\left\lceil \frac{D_i}{T_s} \right\rceil} \right\}$$

Concerning the worst-case execution time of task $\tau_{s,p}$ in normal mode $C^A_{s,p}$, the knowledge about the ratio between $C^N_{s,p}$ and $C^A_{s,p}$ is assumed to be given as a factor ξ with $C^A_{s,p} = \xi \cdot C^N_{s,p}$, so that $C^N_{s,p}$ can be computed by:

$$C^N_{s,p} = \frac{1}{\xi} \cdot C^A_{s,p}$$

Having obtained the understanding how to compute the worst-case execution time of a shared task's subtask $\tau_{s,p}$ as well as of in which manner to select its priority, these techniques can subsequently be combined with the earlier presented partitioned scheduling strategies (cf. 6.2) in the context of the *Semi-Partitioned Scheduling Policy with Task Splitting (TS)* algorithm (cf. Algorithm 12), which works in the following fashion.

After applying a partitioned scheduling strategy of choice until the point of its failure (cf. Algorithm 12, l. 2), the attempt is made to allocate each remaining task $\tau_t \in \mathbf{T}$ availing the concept of task splitting. Iterating over all subsystems $\mathbf{T}_p$, the particular value of $C^A_{s,p}$ is derived according to the previously explained calculations (cf. Algorithm 12, ll. 8-16), whereby it is taken into consideration that an already shared task's response time $R^A_{i,p}$ equals its worst-case execution time $C^A_{i,p}$ (cf. Algorithm 12, ll. 11-12) since it is scheduled under the highest priority on the respective subsystem. Thereon, the schedulability test for a Multicore System with Dynamic Real-Time Guarantees (cf. 6.2.1) is applied to determine if a suchlike can be established with $\tau_{s,p}$ being scheduled under the highest priority in $\mathbf{T}_p$. In case the schedulability condition holds, $C^A_{s,p}$ is computed (cf. Algorithm 12, l. 18) and $\tau_{s,p}$ is assigned to the respective processor. This procedure is repeated until the auxiliary variable C_{req}, which indicates the remaining execution demand of τ_s, reaches zero. As soon as each $\tau_s \in \mathbf{T}$ is distributed, the algorithm terminates successfully. Otherwise, the task set is declared to be unschedulable under this policy.

Algorithm 12 Semi-Partitioned Scheduling Policy with Task Splitting (TS)

Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$, factor ξ

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
 1: Sort T by $D_i$ increasingly
 2: Apply partioned scheduling policy until NOT POSSIBLE is returned
 3: Find Assignment(T, S)
 4: procedure FIND ASSIGNMENT(T, S)
 5:     for each $\tau_s \in \mathbf{T}$ do
 6:         assigned := false
 7:         $C_{req} := C_s^A$
 8:         for each $\mathbf{T}_p \in \mathbf{S}$ do
 9:             $C_{s,p} := 0$
10:             for each $\tau_i \in \mathbf{T}_p$ do
11:                 $x := 0$
12:                 if ($\tau_i$ is shared task) then
13:                     $x := \frac{D_i - C_i^A}{\lceil \frac{D_i}{T_s} \rceil}$
14:                 else
15:                     $x := \frac{D_i - R_{i,p}^A}{\lceil \frac{D_i}{T_s} \rceil}$
16:                 if ($x < C_{s,p}^A$) then
17:                     $C_{s,p}^A := \max(0, x)$
18:             if ($C_{s,p}^A \neq 0$ and FIND OPTIMAL PRIORITY ASSIGNMENT($\{\tau_{s,p}\} \cup \mathbf{T}_p$) $\neq$ NOT POSSIBLE) then
19:                 $C_{s,p}^N := \frac{1}{\xi} \cdot C_{s,p}^A$
20:                 $\mathbf{T}_p := \{\tau_{s,p}\} \cup \mathbf{T}_p$
21:                 $C_{req} := C_{req} - C_{s,p}^A$
22:                 if ($C_{req} = 0$) then
23:                     assigned := true
24:                     $\mathbf{T} := \mathbf{T} \backslash \{\tau_s\}$
25:                     break
26:         if (assigned = false) then return NOT POSSIBLE
27:     return System S with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

By means of the semi-partitioned scheduling policy with task splitting, it is possible to schedule task sets with respect to which strictly partitioned approaches encounter their limits. This phenomenon is investigated more in detail when engaging with experiments in 7. Nevertheless, Lakshmanan et al. [19] strived to attain even greater improvement by modifying the task splitting concept of Kato and Yamasaki [18] as covered subsequently.

6.3.2. Highest-Priority Task Splitting

The highest-priority task splitting approach by Lakshmanan et al. [19] is based on the task splitting paradigm by Kato and Yamasaki [18], but follows a divergent direction in terms of the actual splitting behavior. While in the model of Kato and Yamasaki [18] as well as in the above introduced derivate version applicable for Multicore Systems with Dynamic Real-Time Guarantees (cf. 6.3.1) consistently a task τ_s is chosen as splitting candidate including which the schedulability test does not hold for the respective task set $\mathbf{T}$, the underlying concept of highest-priority task splitting is of different kind. Here, a processor's highest-priority task τ_{hp} is shared across several cores such that the respective subsystem $\mathbf{T}_p$ comprising the contemplated task τ_s is feasible as System with Dynamic Real-Time

Guarantees. Against this backdrop, the *Semi-Partitioned Scheduling Policy with Highest-Priority Task Splitting (HPTS)* has been developed for the application domain of Multicore Systems with Dynamic Real-Time Guarantees and shall hereon be eludicated in detail.

Analogously to the semi-partitioned scheduling policy with task splitting (cf. 6.3.1), the semi-partitioned scheduling policy with highest-priority task splitting is employed at the precise moment when a partitioned scheduling strategy fails. As a consequence, the attempt is made to distribute all remaining, i.e., all unallocated, $\tau_t \in \mathbf{T}$ successively on the available processors, i.e., on those which boast with spare capacity, by sharing a processor's highest-priority task τ_{hp} instead of τ_t. The algorithm, as depicted in Algorithm 13, iterates through all subsystems $\mathbf{T}_p \in \mathbf{S}$ exhibiting unused execution resources and, in the first instance, disjoins τ_{hp} from $\mathbf{T}_p$ (cf. 13, l. 14, l. 20) in order to obtain sufficient processor capacity for $\mathbf{T}_p \cup \{\tau_t\} \backslash \tau_{hp}$ to pass the schedulability test. If removing τ_{hp} once does not lead to the desired result, τ_{hp} is merged with $\mathbf{T}$(cf. 13, l. 15, l. 21), i.e. τ_{hp} needs to be re-allocated to another core, and the procedure is repeated until the schedulability condition is satisfied for $\mathbf{T}_p \cup \{\tau_t\} \backslash \tau_{hp}$ (cf. Algorithm 13, ll. 13-17, 19-22). In this case, τ_t is assigned to $\mathbf{T}_p$ (cf. 13, ll. 17-18, l. 24) and, afterwards, a task splitting routine is invoked (cf. 13, l. 25) for the purpose of dividing the last highest-priority task τ_{hp}, which has been deleted from $\mathbf{T}_p$, into two subtasks $\tau_{hp'}$ and $\tau_{hp''}$. The splitting routine operates equivalently to the actual task splitting and worst-case execution computation process applied in the context of the semi-partitioned scheduling policy with task splitting (cf. Algorithm 12) and divides τ_{hp} in such a way that the allocation of $\tau_{hp'}$ to $\mathbf{T}_p$ under highest priority, if feasible in compliance with the conditions of a System with Dynamic Real-Time Guarantees (cf. Algorithm 13, l. 26), consumes the residual execution capacity, whereas $\tau_{hp''}$ is merged with $\mathbf{T}$(cf. 13, l. 28). In this case, the respective processor is furthermore identified as not available for the particular task (cf. Algorithm 13, l. 29) in order to prevent a deadlock. The set of unallocated tasks $\mathbf{T}$ is then reordered by increasing relative deadline D_i (cf. Algorithm 13, l. 31). If any task cannot be assigned successfully, the task set is declared as unfeasible by this scheduling policy.

Since a shared task's subtask $\tau_{hp''}$ is not directly re-allocated to another processor once τ_{hp} has been split, but rather returned to $\mathbf{T}$, which is why already split tasks can be chosen again for splitting, it is necessary to examine if a contemplated task $\tau_t \in \mathbf{T}$ is a shared task (cf. 13, l. 11) owing to the fact that the definition of a shared task (cf. 4) must be satisfied at any point of time. More specifically, if this is the case, it has to be ensured firstly that not yet any other share of τ_t has been assigned to $\mathbf{T}_p$ (cf. 13, l. 8) and, secondly, that τ_t is assigned under the highest priority on the particular subsystem (cf. 13, l. 17).

To what extent the semi-partitioned scheduling policy with highest-priority task splitting stands out from the previously introduced semi-partitioned scheduling policy with task splitting with respect to the amount of accepted task sets, will be investigated by virtue of experiments and discussed in chapter 7.

Algorithm 13 Semi-Partitioned Scheduling Policy with Highest-Priority Task Splitting (HPTS)

Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
1: Sort $\mathbf{T}$ by $D_i$ increasingly
2: Apply partioned scheduling policy until NOT POSSIBLE is returned
3: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

4: procedure FIND ASSIGNMENT($\mathbf{T}$, $\mathbf{S}$)
5:     for each $\tau_t \in \mathbf{T}$ do
6:         assigned := false
7:         for each $\mathbf{T}_p \in \mathbf{S}$ do
8:             if (available$_p$ and $\tau_t$ is not already shared on $\mathbf{T}_p$) then
9:                 $\mathbf{T}'_p := \emptyset$
10:                $\tau_{hp}$ := highest-priority task in $\mathbf{T}_p$
11:                if ($\tau_t$ is shared task) then
12:                    $\mathbf{T}^*_p := \mathbf{T}_p \cup \{\tau_t\}$ with $\tau_t$ as highest-priority task
13:                    while ((FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}^*_p$) = NOT POSSIBLE)) do
14:                        $\mathbf{T}_p := \mathbf{T}_p \backslash \{\tau_{hp}\}$
15:                        $\mathbf{T} := \mathbf{T} \cup \{\tau_{hp}\}$
16:                        $\tau_{hp}$ := highest-priority task in $\mathbf{T}_p$
17:                        $\mathbf{T}^*_p := \mathbf{T}_p \cup \{\tau_t\}$ with $\tau_t$ as highest-priority task
18:                    $\mathbf{T}_p := \mathbf{T}^*_p$
19:                else
20:                    while ((FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) = NOT POSSIBLE)) do
21:                        $\mathbf{T}_p := \mathbf{T}_p \backslash \{\tau_{hp}\}$
22:                        $\mathbf{T} := \mathbf{T} \cup \{\tau_{hp}\}$
23:                        $\tau_{hp}$ := highest-priority task in $\mathbf{T}_p$
24:                    $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
25:                $(\tau_{hp'}, \tau_{hp''})$ := SPLIT($\tau_{hp}, \mathbf{T}_p$)
26:                if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}'_p \cup \{\tau_{hp'}\}$) $\neq$ NOT POSSIBLE) then
27:                    $\mathbf{T}_p := \mathbf{T}_p \cup \tau_{hp'}$ with $\tau_{hp'}$ as highest-priority task
28:                    $\mathbf{T} := \mathbf{T} \cup \tau_{hp''}$
29:                    available$_p$ := false
30:                    assigned := true
31:                    Sort $\mathbf{T}$ by $D_i$ decreasingly
32:                    break
33:        if (assigned = false) then return NOT POSSIBLE
34:    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

35: procedure SPLIT($\tau_{hp}, \mathbf{T}_p$)
36:    $C_{hp'} := 0$
37:    for each $\tau_i \in \mathbf{T}_p$ do
38:        if ($\tau_i$ is shared task) then
39:            $x := \frac{D_i - C_i^A}{\left\lceil \frac{D_i}{T_{\tau_{hp}}} \right\rceil}$
40:        else
41:            $x := \frac{D_i - R_{i,p}}{\left\lceil \frac{D_i}{T_{\tau_{hp}}} \right\rceil}$
42:        if ($x < C_{t,p}$) then
43:            $C_{hp'} := \max(0, x)$
44:    if (FIND OPTIMAL PRIORITY ASSIGNMENT($\{\tau_{hp'}\} \cup \mathbf{T}_p$) $\neq$ NOT POSSIBLE) then
45:        return $(0, C^A_{\tau_{hp}})$
46:    else
47:        return $(C_{hp'}, C^A_{\tau_{hp}} - C_{hp'})$
```

6.3.3. Complete Task Migration

After having investigated how to successfully schedule more task sets than it is possible by dint of merely partitioned algorithms, namely, with the aid of task splitting, henceforth a diverging issue shall be addressed. Since the knowledge of how to establish a Multicore System with Dynamic Real-Time Guarantees has been provided, hence this model's fundamentals shall be remembered. Due to the matter of fact that a Multicore System with Dynamic Real-Time Guarantees pertains to the category of real-time systems, which inevitably require temporal correctness of all obtained results, and, moreover, that a Multicore System with Dynamic Real-Time Guarantees's timing strict tasks are safety-critical, such systems are verified offline by complying with the respective schedulability conditions during the system establishment process, as presented in previous sections (cf. 6.2, 6.3.1, 6.3.2). By this means, it can be ensured that a Multicore System with Dynamic Real-Time Guarantees fulfills its timing requirements even in case of abnormal execution behavior (cf. 6).

However, it may be the case that, e.g. due to external influences or wear of components, one or more processors of the respective system break down and therefore are not usable any longer. By reason of the fact that such incidents are not taken into account in the Multicore System with Dynamic Real-Time Guarantees model, the system's proper functioning cannot be guaranteed in any respect under these circumstances. Nevertheless, a solution for this problem shall be provided inspired by task migration paradigm suggested in the work of Xu and Burns [25].

In their paper, Xu and Burns [25] contemplate dual-core dual-criticality systems with a task set $\mathbf{T}$ encompassing a subset of high- as well as of low-criticality tasks. Resulting from this, regarding the employed task model, each task $\tau \in \mathbf{T}$ is endowed with two distinct values specifying its worst-case execution time, in fact, $C_i(HI)$ and $C_i(LO)$ (cf. 3). As long as no task surpasses its respective $C_i(LO)$, all deadlines are met and no migration occurs. Notwithstanding low-criticality tasks are not permitted to exceed their low-criticality execution time budget, the possibility exists that a high-criticality task overshoots its respective $C_i(LO)$. In this event, a certain amount of low-criticality tasks migrate from the afflicted core to another processor, whereat all tasks remain schedulable. Provided that high-criticality tasks on more than one processor exceed their low-criticality budget, some low-criticality tasks are abandoned in order to maintain the remaining tasks' schedulability. In the latter case, no migration occurs.

Without going into further detail, since the system model established by Xu and Burns [25] is not compliant with that one required by a Multicore System with Dynamic Real-Time Guarantees, simply the fundamental idea of migrating tasks from one processor to another shall be availed in order to provide the opportunity to verify offline a Multicore System with Dynamic Real-Time Guarantees's behavior in case of one or more processor breakdowns. Against this backdrop, the *l-Failure-Proof Multicore System with Dynamic Real-Time Guarantees* is introduced hereinafter.

Definition 5 (l-Failure-Proof Multicore System with Dynamic Real-Time Guarantees).
Consider a Multicore System with Dynamic Real-Time Guarantees $\mathbf{S}$ with subsystems $\mathbf{T}_p$, whereat a set of tasks $\mathbf{T} = \tau_1, \ldots, \tau_n$, with $n \in \mathbb{N}$, is partitioned to a set of m homogeneous processors. $\mathbf{S}$ is a l-Failure-Proof Multicore System with Dynamic Real-Time Guarantees if the following conditions are satisfied at any point of time:

- No migration occurs if all tasks are executed in normal mode while no processor's functional status is compromised.
- No migration occurs if one or more tasks are executed in abnormal mode while no processor's functional status is compromised.
- If one or more, but at maximum l processors, with $0 < l < m$, break down, the affected subsystems' tasks are permanently migrated onto the remaining set of intact processors or onto the remaining intact processor in such a way that the system $\mathbf{S}$ remains schedulable complying with the characteristics of a Multicore System with Dynamic Real-Time Guarantees.

Following the proceeding of Xu and Burns [25], subsequently shall be clarified how the task migration as described in Definition 5 can be realized.

When a processor breakdown occurs, no information is available about the specific execution state of the affected tasks, i.e., if their execution is completed, partly completed or pending, by reason of which all migrating tasks are obliged to expend their entire execution budget on their new destination core. Owing to the fact that a certain amount of time might have passed in the interval between a task's release and its migration, a new relative deadline has to be derived for each $\tau_i \in \mathbf{T}_b$, where $\mathbf{T}_f$ denotes the failing subsystem. Since the system's feasibility has to be verified offline and the exact value of the time interval elapsed can only be retrieved at runtime, a pessimistic estimation is made, so that the new deadline of task $\tau_i \in \mathbf{T}_f$ can be derived as follows:

$$D'_i = D_i - J_i$$

The so-called *release jitter* for timing strict as well as for timing tolerable tasks is given by:

$$J_i = R_i^N - C_i^N$$

$$J_i = R_i^A - C_i^A$$

The response time R_i^N or R_i^A, respectively, can be calculated according to the proceeding of response time analysis offered in 6.3.1.

Due to the fact that not only the feasiblity of the primal Multicore System with Dynamic Real-Time Guarantees, i.e., in case all processors are available, has to be verified but also the feasiblity after the migration of one or more subsets of $\mathbf{T}$, the schedulability condition as defined in 6.2.1 must be checked multiple times. More precisely, the schedulability test has to be performed $l + 1$ times: Once initially and once again for each collapsing subsystem, whereat it is of no importance if the l processors break down successively or all at once, because according to the logical flow the migration takes places subsystem by subsystem. Concerning these additional schedulability tests, not the original deadlines D_i of $\tau_i \in \mathbf{T}_f$ but the modified ones D'_i have to be taken into consideration. In the course of this, not specific processors are contemplated as failure candidates, but instead each possible subset of l processors in $\mathbf{S}$, to ensure the systems reliability.

6.3.4. Timing Tolerable Task Migration

Unlike the scenario reflected upon before, furthermore the possibility exists that cores of a system not break down but nevertheless are corrupted for a longer interval of time due to, e.g., external influences, which leads to a processor state in which all tasks are executed abnormally. In such cases, the properties of a Multicore System with Dynamic Real-Time Guarantees are maintained anyway, i.e., all timing strict tasks meet their respective deadlines, but the timing tolerable tasks' bounded tardiness increases significantly. Since timing tolerable tasks are known to be not safety-critical, this does not seem to be a major issue, but, however, a certain amount of timing tolerable may be more important than others. For this reason, it would be desirable if those did not deliver their results with utmost tardiness, whereas, regarding other timing tolerable tasks, it is entirely irrelevant if they are delayed even more. On this basis, again availing the task migration paradigm by Xu and Burns [25], the attempt shall be made to reduce a specified group of timing tolerable tasks' tardiness if the processor they are allocated to exhibits outright abnormal execution behavior. Pursuing this objective, the *Multicore System with Dynamic Real-Time Guarantees with Task Migration* is introduced in the following.

Definition 6 (Multicore System with Dynamic Real-Time Guarantees with Task Migration).
Consider a Multicore System with Dynamic Real-Time Guarantees $\mathbf{S}$ with subsystems $\mathbf{T}_p$, whereat a set of tasks $\mathbf{T} = \tau_1, \ldots, \tau_n$, with $n \in \mathbb{N}$, is partitioned to a set of m homogeneous processors. $\mathbf{S}$ is a Multicore System with Dynamic Real-Time Guarantees with Task Migration if the following conditions are satisfied at any point of time:

- No migration occurs if all tasks are executed in normal mode.
- If all tasks are executed in abnormal mode, a certain amount of timing tolerable tasks is migrated from the afflicted subsystem $\mathbf{T}_i$ to another subsystem $\mathbf{T}_j$, so that all tasks on $\mathbf{T}_i$ comply with their respective timing requirements, whereas $\mathbf{T}_j$ still

satisfies the characteristics of a System with Dynamic Real-Time Guarantees, i.e. each timing strict tasks meets its hard deadline while all timing tolerable tasks have bounded tardiness. In this case the not migrating timing tolerable tasks' tardiness is reduced due to lowered processor utilization.

- If tasks on more than one subsystem are executed abnormally, no migration occurs and the timing tolerable tasks on all afflicted subsystems have bounded tardiness.
- If the normal system state of all subsystems is recovered, i.e. all tasks execute normally, each task that has previously been migrated is re-migrated to the processor it has been allocated to initially.

Consider, for convenience only, a dual-core Multicore System with Dynamic Real-Time Guarantees $\mathbf{S} = \mathbf{T}_p \cup \mathbf{T}_q$ with an already established task partitioning of task set $\mathbf{T}$ onto the respective subsystems. Each subsystem $\mathbf{T}_p$, $\mathbf{T}_q$ contains a set of timing strict tasks $\mathbf{T}_{p,hard}$,$\mathbf{T}_{q,hard}$ as well as a set of timing tolerable tasks $\mathbf{T}_{p,soft}$, $\mathbf{T}_{q,soft}$. In the same vein as in the model of Xu and Burns [25], the latter in turn consists of a certain number of tasks authorized to migrate $\mathbf{T}_{p,soft,mig}$, $\mathbf{T}_{q,soft,mig}$ and a set of tasks which are statically allocated to the respective processor $\mathbf{T}_{p,soft,stat}$, $\mathbf{T}_{q,soft,stat}$, so that $\mathbf{T}_{p,soft} = \mathbf{T}_{p,soft,mig} \cup \mathbf{T}_{p,soft,stat}$ where the indication of migrating tasks is assumed to be given. If the subsystem does not execute completely abnormally, it holds that $\mathbf{T}_p = \mathbf{T}_{p,hard} \cup \mathbf{T}_{p,soft,mig} \cup \mathbf{T}_{p,soft,stat}$ and $\mathbf{T}_q = \mathbf{T}_{q,hard} \cup \mathbf{T}_{q,soft,mig} \cup \mathbf{T}_{q,soft,stat}$. As soon as all tasks in $\mathbf{T}_p$ are in abnormal execution mode, it follows that $\mathbf{T}_p = \mathbf{T}_{p,hard} \cup \mathbf{T}_{p,soft,stat}$ and $\mathbf{T}_q = \mathbf{T}_{q,hard} \cup \mathbf{T}_{q,soft,mig} \cup \mathbf{T}_{q,soft,stat} \cup \mathbf{T}_{p,soft,mig}$, while, if the same situation occurs on $\mathbf{T}_q$, this leads to $\mathbf{T}_p = \mathbf{T}_{p,hard} \cup \mathbf{T}_{p,soft,mig} \cup \mathbf{T}_{p,soft,stat} \cup \mathbf{T}_{q,soft,mig}$ and $\mathbf{T}_q = \mathbf{T}_{q,hard} \cup \mathbf{T}_{q,soft,stat}$. In general, it is possible that either one core individually enters a completely abnormal execution state or both simultaneously. The second case can be refined, because, concerning this matter, two options must be distinguished: Either fully abnormal task execution commences on both processors one by one or in exactly the same point of time, which, as a consequence, leads to increased bounded tardiness with respect to the set of timing tolerable tasks.

As already discussed in 6.3.3, it is necessary to modify a task's deadline when migrating it to another core. This operation is performed in identical fashion as with respect to the l-Failure-Proof Multicore System with Dynamic Real-Time Guarantees so that the modified deadline of a task $\tau_i \in \mathbf{T}_{p,soft}$ can be retrieved as $D_i' = D_i - J_i$ with the release jitter given as $J_i = R_i^N - C_i^N$. Here, the schedulability condition defined in 6.2.1 must be checked five times: firstly, in the context of the partitioned scheduling stage, secondly, to verify that $\mathbf{T}_p = \mathbf{T}_{p,hard} \cup \mathbf{T}_{p,soft,mig} \cup \mathbf{T}_{p,soft,stat} \cup \mathbf{T}_{q,soft,mig}$ is feasible in terms of the requirements of a System with Dynamic Real-Time Guarantees in case all tasks on subsystem $\mathbf{T}_q$ exhibit abnormal execution behavior, and thirdly, to prove that $\mathbf{T}_q = \mathbf{T}_{q,hard} \cup \mathbf{T}_{q,soft,mig} \cup \mathbf{T}_{q,soft,stat}$ is still feasible as a System with Dynamic Real-Time Guarantees under the given circumstances. Furthermore, the last two steps must be repeated in analogous fashion, taking into consideration that, in contrast, all tasks on $\mathbf{T}_p$ may be executed abnormally. These additional tests are justified on account of the modified relative deadlines of migrating tasks, so that the schedulability condition in the first migration event must be assessed applying modified deadlines D'_i, as explained above, to all tasks $\tau_i \in \mathbf{T}_{p,soft,mig}$ or $\tau_j \in \mathbf{T}_{q,soft,mig}$, respectively, while concerning the re-migration the migration tasks' deadlines have to be adjusted once again. At this juncture, again the time elapsed between a task's release and its migration is considered. Although the second job of a migrated task is likely to execute under its original timing parameters, since it is released on its destination core $\mathbf{T}_q$, it is nevertheless imaginable that a task instance executing under a modified deadline is re-migrated to its source processor $\mathbf{T}_p$ before its completion. On that score, again, a pessimistic rescaling of the task deadline is conducted, whereat the release jitter on its shelter subsystem $\mathbf{T}_q$ is contemplated, in fact, $D''_i = D'_i - J_{i,q} = D_i - J_i - J_{i,q}$. These parameters can be calculated in the same manner for the event of all tasks on $\mathbf{T}_q$ being in abnormal execution state. However, as soon as a migration task returns to its original core, its next released job is executed based on its primal specifications.

The Multicore System with Dynamic Real-Time Guarantees with Task Migration paradigm is the last approach thematized in this thesis and thus concludes a miscellaneous series of methods providing the opportunity to transfer the System with Dynamic Real-Time Guarantees model by von der Brüggen et al. [24] onto a homogeneous multicore environment. While the thoroughly partitioned scheduling policies (cf. 6.2) aim at the fundamental establishment of a Multicore System with Dynamic Real-Time Guarantees and the task splitting strategies (cf. 6.3.1, 6.3.2) endeavor to increase the number of accepted task sets under the Multicore System with Dynamic Real-Time Guarantees model, the task migration concept attempts to increase the system's reliability on the one hand (cf. 6.3.3) and to meliorate the objective of tardiness on the other hand. More precisely, timing tolerable task migration paves the way to reduce tardiness concerning a specified set of timing tolerable tasks in the context of outright abnormal execution behavior on a particular processor. By this means, the possibility is offered to treat a target group of tasks extraordinarily, which do not evince the safety characteristics of timing strict tasks, but stand out of the multitude of timing tolerable tasks.

Conclusively, in the next chapter all approaches elaborated in this thesis will be analyzed, compared and assessed being premised on the results of implemented experiments.

7. Evaluation

Having learned about various possibilities to transfer the Systems with Dynamic Real-Time Guarantees model of von der Brüggen et al. [24] onto a multiprocessor environment and, building on this, to improve several objectives such as the acceptance rate of task sets as a function of the system utilization, the system reliability as well a certain group of tasks' bounded tardiness, henceforward shall be investigated if these methods not only seem reasonable in theory but also turn out to be beneficial in practice. For this purpose, the experimental setup shall be revealed initially (cf. 7.1), before proceeding with the contemplation of the experiments' findings, dealing with each objective successively. Thus, after comparing the partitioned scheduling strategies (cf. 7.2), it shall be examined which improvement can be achieved by the use task splitting as well as of highest-priority task-splitting (cf. 7.3). Moreover, a closer look will be taken at the trade-off between the number of accepted task sets and the gained system reliability (cf. 7.4) or reduced tardiness (cf. 7.5), respectively, resulting from the employment of the task migration paradigm. Finally, the obtained outcomes will be summarized and evaluated, before concluding this thesis with a short outlook (cf. 7.6).

7.1. Experiment Setup

In the following, the developed scheduling policies shall be tested by means of randomly generated implicit-deadline task-sets according to the approach of von der Brüggen et al. [24]. In the course of this, the ratio between the worst-case execution time in normal mode C_i^N and in abnormal mode C_i^A is specified by a factor ξ with $C_i^A = \xi \cdot C_i^N$. Since at this point the single re-execution of a task instance shall be selected as the fault-recovery routine of choice, this factor is given as $\xi = 1.83$. This is justified by the adherence to the experimental specifications defined by von der Brüggen et al. [24] in the context of evaluating their Systems with Dynamic Real-Time Guarantees model, whereby the execution of a fault-detection routine is assumed to consume 20% of a job's plain worst-case execution time, so that, if only one fault-detection is performed at the end, this results in a ratio $\xi = \frac{C_i^A}{C_i^N} = \frac{2.2}{1.2} \approx 1.83$ [24]. Von der Brüggen et al. [24] also consider two further fault-recovery strategies, namely, twofold re-execution as well as checkpointing, but, however, these will not be taken into consideration at this juncture.

While the merely partitioned as well as the task splitting scheduling strategies are compared on the basis of one specific setting, i.e., a timing strict task percentage of 50%, $\xi = 1.83$, and a task set size of 10 tasks per run among 1000 runs in total, which engenders a sound conclusion, the migration approaches are assessed under varying parameters to make an accurate statement concerning their cost-benefit factors.

7.2. Partitioned Scheduling Strategies

Hence, the partitioned scheduling-policies introduced earlier (cf. 6.2) will be evaluated fitting-strategy by fitting-strategy, i.e., all approaches being premised on the same underlying bin-packing heuristic will be contemplated collectively, beginning with the first-fit paradigm.

In Figure 7.1, the percentage of accepted task sets as a function of the system utilization is depicted for DM_FF, CS_DM_FF, DU_FF and CS_DU_FF likewise. Against all expectations, the criticality-successive methods, i.e., those not contemplating $\mathbf{T}$ at large but $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ consecutively instead, do not outperform DM_FF and DU_FF but rather rank behind significantly. Moreover, the idea of a deadline-monotonic task preordering comes to fruition, since the respective scheduling policies yield substantially better results than those employing a preordering according to decreasing task utilization.

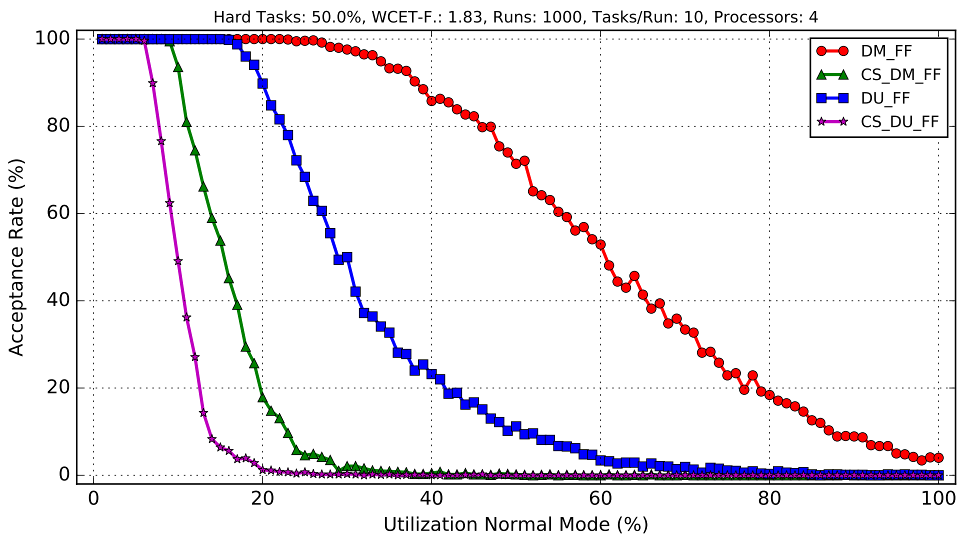

Figure 7.1.: Comparison of all scheduling policies based on the first-fit bin-packing heuristic developed in this thesis.

This trend is also evident in Figure 7.2, where again the criticality successive concepts turn out to be inferior than those considering the union of timing strict and timing tolerable tasks and, besides, the strategies applying a deadline-monotonic task preorder dominate over those preordered according to decreasing utilization. Furthermore, comparing Figure 7.1 and Figure 7.2, it becomes apparent that DM_AF clearly surpasses DM_FF in terms of effectiveness, since a decrease in the percentage of accepted task sets begins noticeably later when making use of DM_AF, namely, at a system utilization of about 40%, than of DM_FF, where the descent commences at a system utilization of about 30%. Apart from that, the acceptance rate of DM_AF does not drop under a value of 40%, whereas at full system utilization DM_FF does not achieve an acceptance rate higher than 5%.

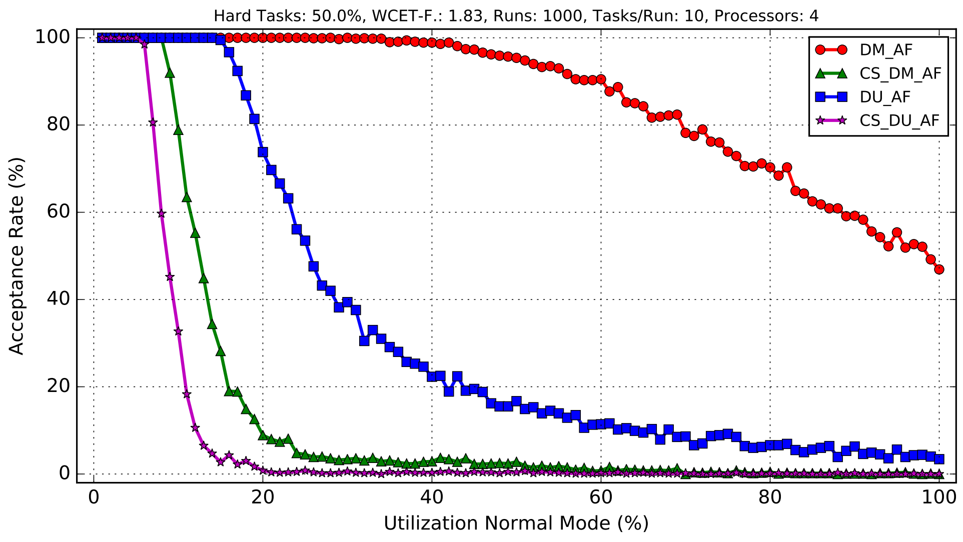

Figure 7.2.: Comparison of all scheduling policies based on the arbitrary-fit bin-packing heuristic developed in this thesis.

The scheduling policies being derived from the best-fit bin-packing heuristic emerge as the overall worst approaches in this series of experiments, as discernible from Figure 7.3. Due to the fact that even the best performing algorithm, i.e. DM_BF, already begins to decrease dramatically in terms of the acceptance rate at system utilization of approximately 15% and reaches 0% at a utilization of less than 40%, all developed best-fit approaches can be regarded as completely ineffective.

Resulting from Figure 7.4, the scheduling policies based on the worst-fit heuristic prove to perform better than those applying best-fit, but nevertheless rank behind the earlier considered first-fit and arbitrary-fit derivates. Again, it is obvious that the criticality-successive algorithms deliver significantly worse results than those contemplating timing strict and timing tolerable tasks in toto and, furthermore, that once again a task preordering turns out to be more practicable when established according to the deadline-monotonic paradigm than when arranged with respect to decreasing utilization. However, also the worst-fit approaches cannot be recommended in terms of Multicore Systems with Dynamic Real-Time Guarantees, owing to the fact that even DM_BF exhibits a rapid acceptance rate decrease at a system utilization of about 20%, reaches 10% at a utilization close to 40%, and finally 0% at a utilization of approximately 70%.

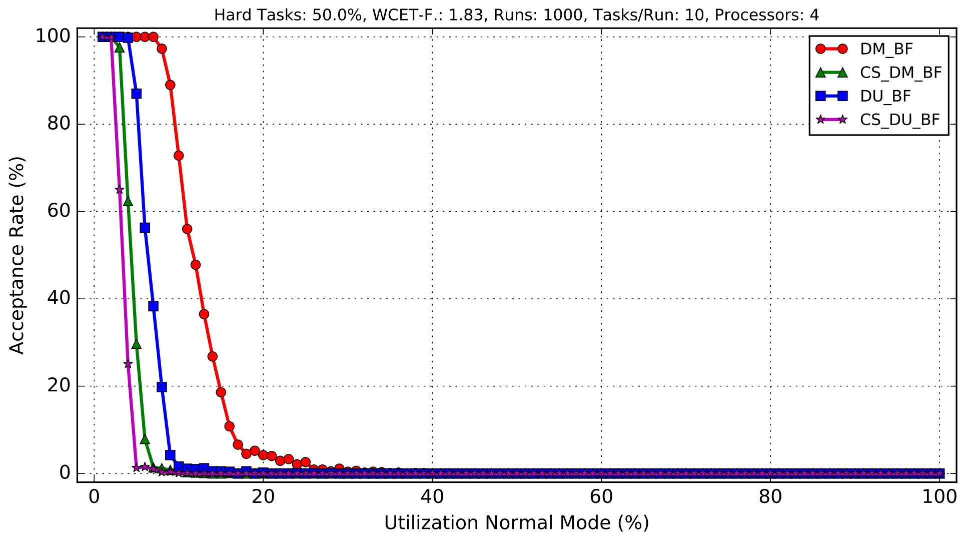

Figure 7.3.: Comparison of all scheduling policies based on the best-fit bin-packing heuristic developed in this thesis.

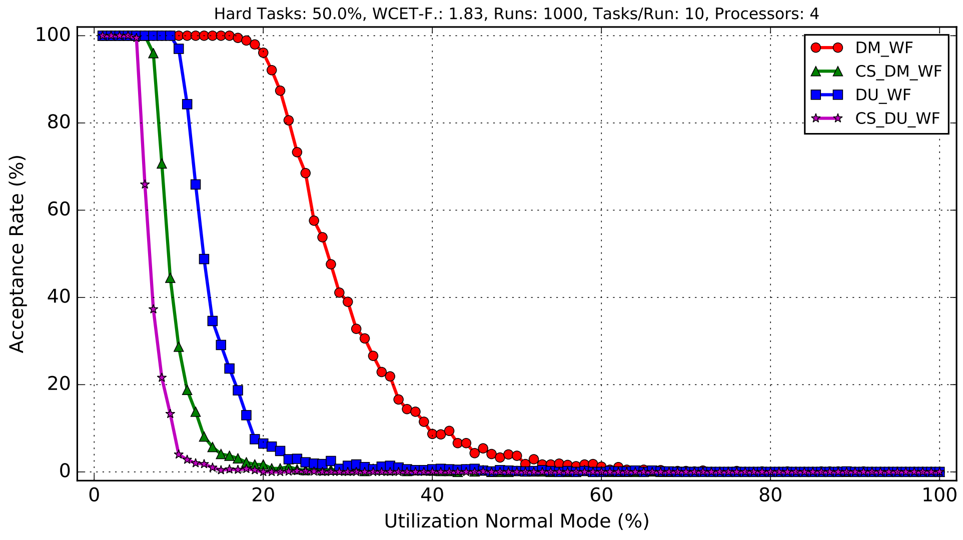

Figure 7.4.: Comparison of all scheduling policies based on the worst-fit bin-packing heuristic developed in this thesis.

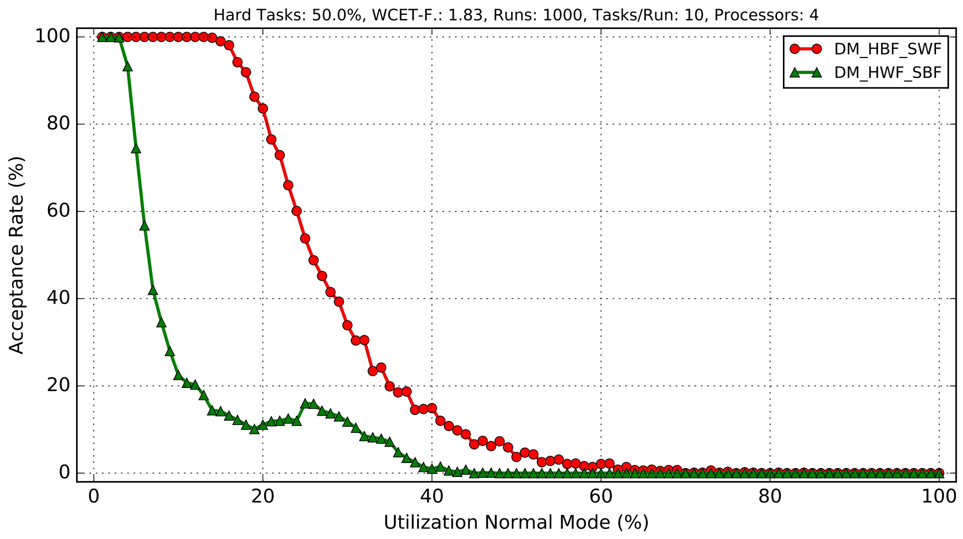

Figure 7.5.: Comparison of all deadline-monotonic criticality-successive scheduling policies developed in this thesis.

Figure 7.5 depicts the experimental results of the deadline-monotonic criticality-successive strategies developed in this thesis which apply disparate bin-packing heuristics to each subset of **T**. Unlike the criticality-successive approaches contemplated before, DM_HBF_SWF does not exhibit an equally steep descent with respect to the task acceptance rate but, nevertheless, does not outperform DM_WF. DM_HWF_SBF stays behind DM_HBF_SWF and diverges sparsely from the aforementioned methods.

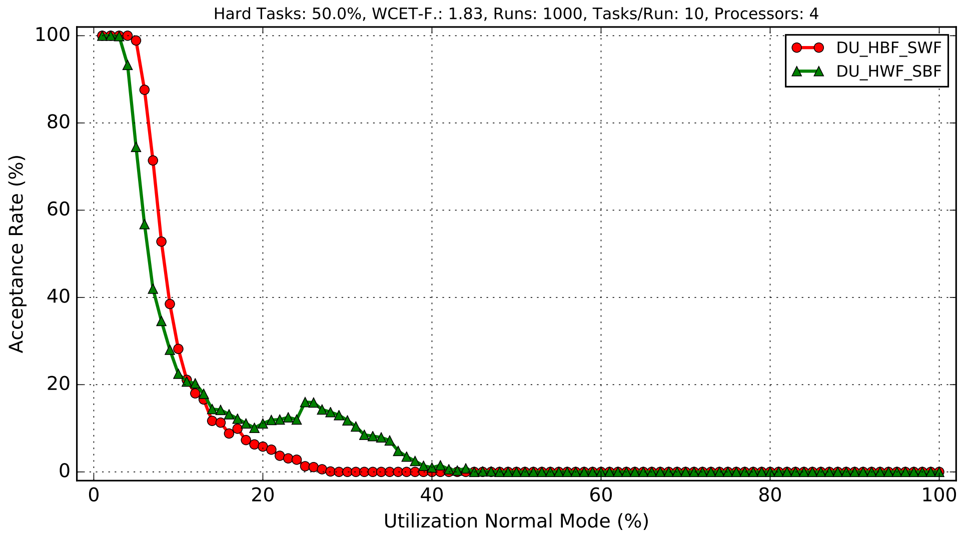

Figure 7.6.: Comparison of all criticality-successive scheduling policies developed in this thesis preordered according to decreasing task utilization.

In Figure 7.6, the findings resulting from inhomogeneous criticality-successive scheduling policies with task preordering according to decreasing task utilization are depicted. While DU_HWF_SBF barely evinces a different behavior than known from DM_HWF_SBF, DU_HBF_SWF bears a certain resemblance to DU_WF (cf. 7.4). Either way, neither of these combined criticality-successive strategies proves satisfactory with respect to the objective of establishing a Multicore System with Dynamic Real-Time Guarantees, may they boast a deadline-monotonic task preorder or a decreasing utilization based.

In general, it turns out to be much more effective to consider timing strict and timing tolerable tasks as a union than contemplating $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ separately, so that the employment of criticality-successive scheduling policies is not advisable in this context. Moreover, in any event the task set T should be preordered with respect to decreasing task utilization in order to attain more satisfactory outcomes. However, DM_AF protrudes from the multitude of partitioned approaches and therefore can be identified as the most suitable scheduling policy for the establishment of a Multicore System with Dynamic Real-Time Guarantees under the given setting.

7.3. Task Splitting and Highest-Priority Task Splitting

Since DM_AF has been determined to be the favored solution to the problem of creating a Multicore System with Dynamic Real-Time Guarantees, subsequently a closer look shall be taken at the findings descending from the employment of task splitting as well as of highest-priority task splitting. Under the same setup as before the particular splitting routines are applied as soon as the conventional DM_AF policy declares a task set as unfeasible in order to improve the acceptance rate as a function of the system utilization.

As evident from Figure 7.7, the usage of task splitting in point of fact leads to an improvement of the regular DM_AF result but, however, to a very marginal. Concerning DM_AF_TS, the decrease in terms of the task set acceptance rate takes place insignificantly slower and reaches a negligibly higher value at a system utilization of 100% than DM_AF. On that score, it remains questionable whether the paltry gain can justify the computational expense caused by the task splitting routine. However, it certainly depends on the respective case of application if it is reasonable to apply this method.

In contrast to the above case, the employment of highest-priority task-splitting does not entail any improvement compared to the regular DM_AF policy, as visualized in Figure 7.8. Due to the fact that the underlying highest-priority task splitting paradigm by Lakshmanan et al. [19] was designed to go beyond the level of task splitting regarding the amount of successfully schedulable task sets, this result is quite surprising and hence indicates that the developed concept is not applicable in terms of Multicore Systems with Dynamic Real-Time Guarantees, at least not in the current shape. Notwithstandig, the attempt could be made to modify the provided algorithm as a future perspective.

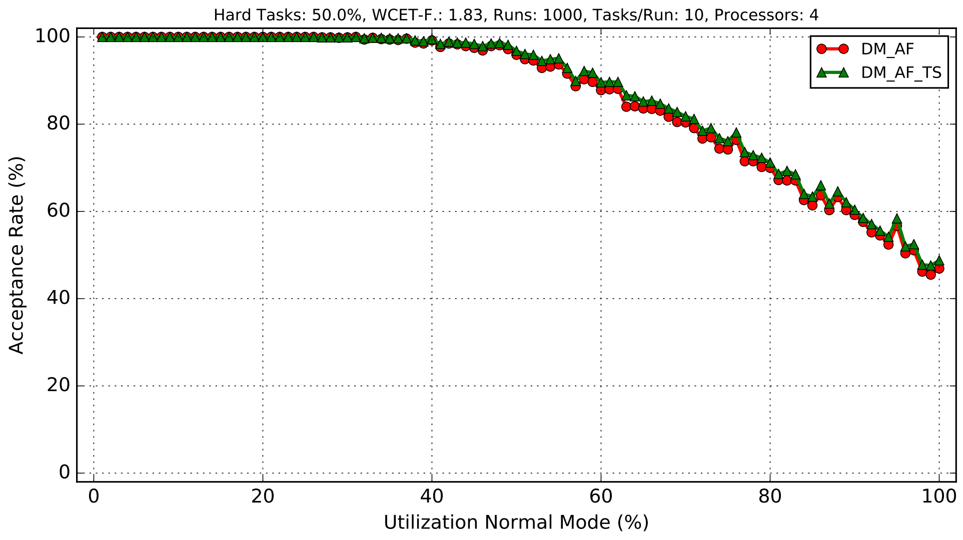

Figure 7.7.: DM_AF with task splitting routine compared to DM_AF.

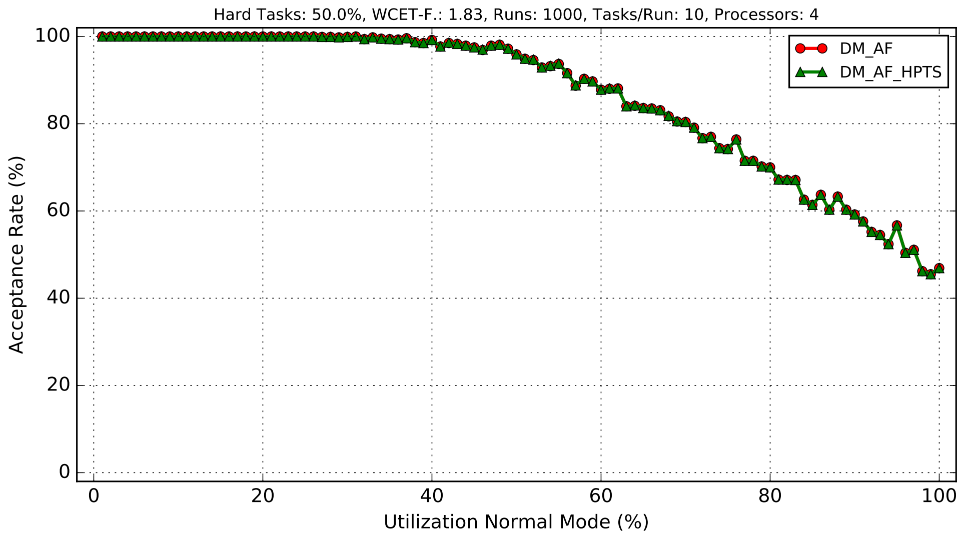

Figure 7.8.: DM_AF with highest-priority task splitting routine compared to DM_AF.

7.4. l-Failure-Proof Multicore Systems with Dynamic Real-Time Guarantees

Having learned how to establish an l-Failure-Proof Multicore Systems with Dynamic Real-Time Guarantees (cf. 6.3.3), as of now shall be examined which is the price for the increased system reliability. More precisely, the task set acceptance rate as a function of the system utilization shall be contemplated based on experiments employing DM_AF and an additional migration routine which is capable of verifying the conditions of a 1-Failure-

Proof Multicore System with Dynamic Real-Time Guarantees, i.e., the case that one core breaks down.

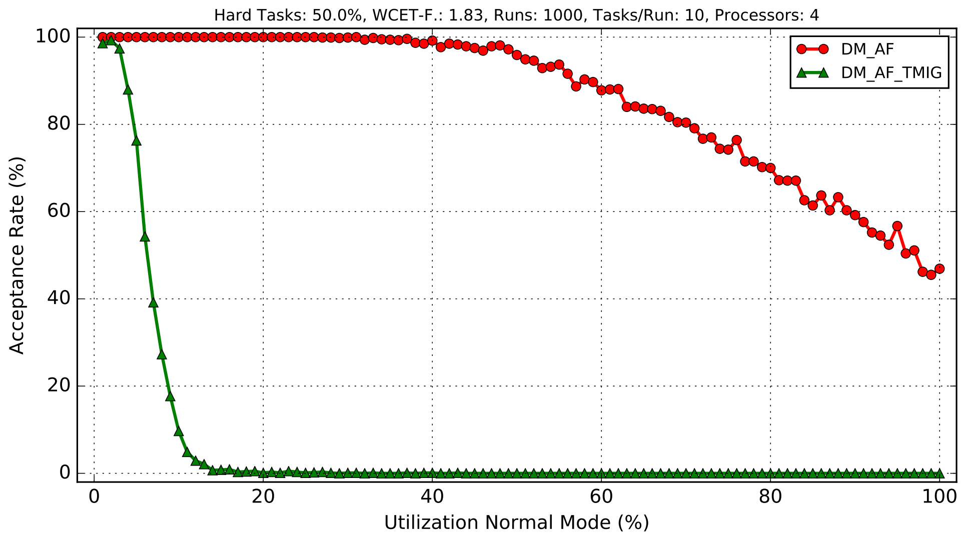

Figure 7.9.: DM_AF with complete task migration routine for the establishment of a 1-Failure-Proof Multicore Systems with Dynamic Real-Time Guarantees compared to DM_AF.

In the first instance, the same experiment setup is used as described in the beginning. In contrast to regular DM_AF scheduling, the acceptance rate of the algorithm establishing a 1-Failure-Proof Multicore System with Dynamic Real-Time Guarantees, DM_AF_TMIG, decreases nearly immediately, in fact, at a system utilization of less than 5%, and reaches 0 at a utilization of approximately 15%. Hence, a a 1-Failure-Proof Multicore System with Dynamic Real-Time Guarantees can only be established considering a total system utilization of at maximum 15% under the given setting.

In order to find a possibly more suitable experiment setup, two parameters have been diversified one by one. In Figure 7.10, the percentage of timing strict tasks has been varied between the amounts of 30%, 40%, 50%, 60%, and 70%, while the residual factors have been maintained. As consequence, it can be noticed that the percentage of timing strict tasks scarcely influences the algorithms result. Interestingly enough, the acceptance rate decreases slightly slower at a timing strict task percentage of 70%, but, however, this could be provoked by random effects.

Furthermore, Figure 7.11 comprises the results of experiments after an adjustment of the task set size, so that the algorithm is executed on a set of 5, 10, 20, and 50 tasks. Resulting from this, it can be observed that the optimal number of tasks with respect to this setting can be identified as 10, since for a smaller task set an acceptance rate of 100% can be obtained at no point of time and, moreover, for larger task sets, the task acceptance rate dependent on the system utilization declines more rapidly.

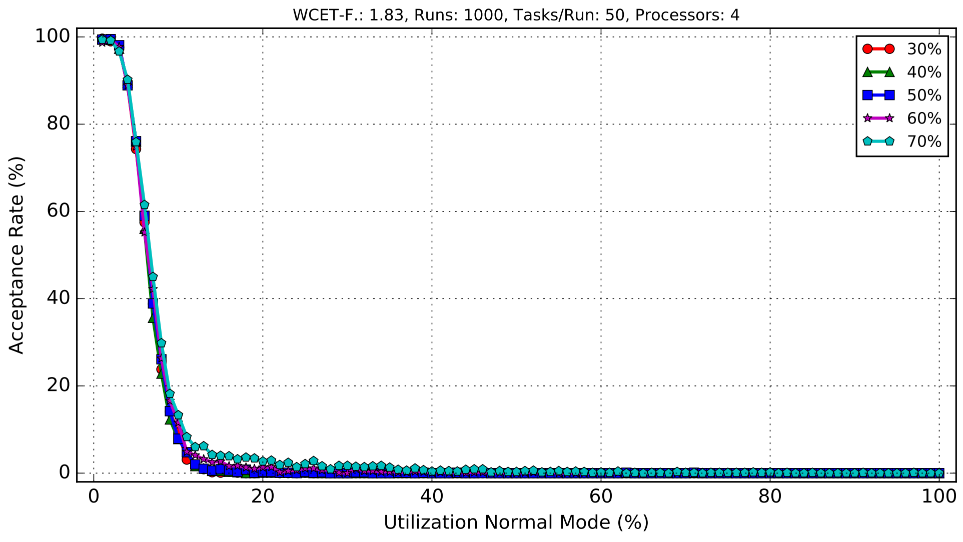

Figure 7.10.: DM_AF with complete task migration routine under varying percentage of timing strict tasks.

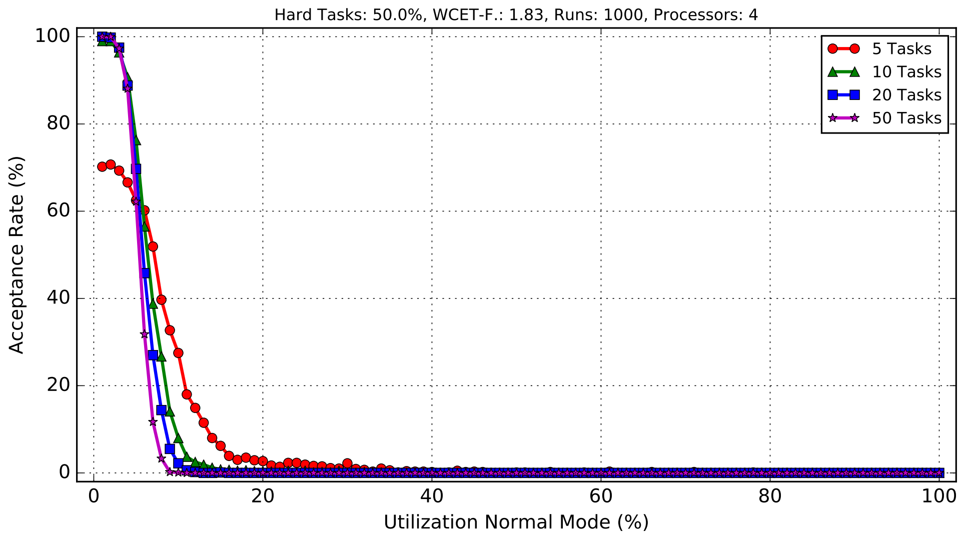

Figure 7.11.: DM_AF with complete task migration routine under varying task set size.

7.5. Timing Tolerable Task Migration

After having contemplated the experimental results of an algorithm performing complete task migration, the findings with respect to a routine performing a migration of a subset of timing tolerable tasks for a certain interval of time (cf. 6.3.4) shall be touched upon. Since the amount of migrating tasks is assumed to be given, one setting for the number of such tasks is chosen at the beginning and employed throughout all experiments, namely, a percentage of 50%. Moreover, solely one core is chosen as the migrating tasks' origin.

From Figure 7.12 it become obvious that also in the course of timing tolerable task migration losses are suffered with respect to the task set acceptance rate. The decent of success cases begins at a system utilization of about 20% and proceeds moderately until a utilization of 100% where still not an acceptance rate of 0 is reached. Nevertheless, the algorithm is surely less efficient than DM_AF but in consideration of its objective, the performance can be identified as satisfactory.

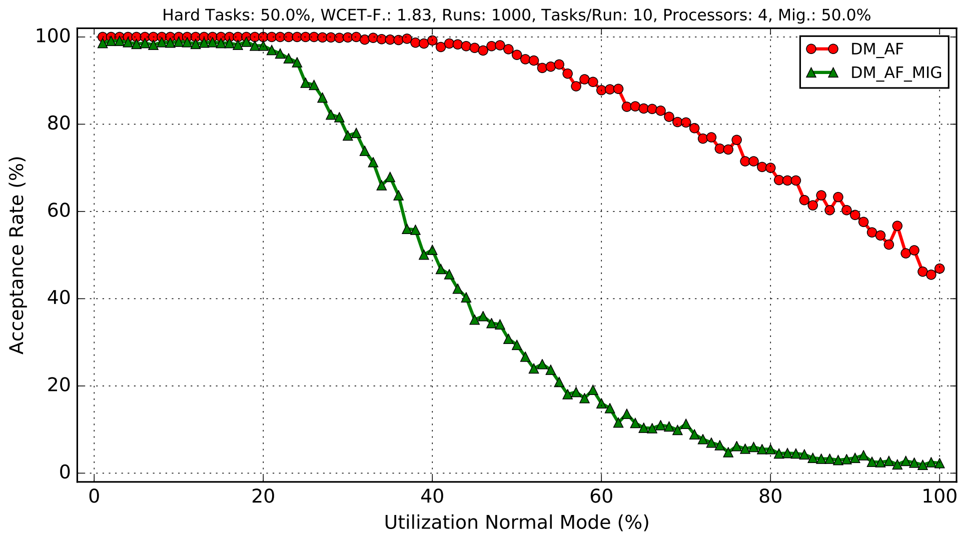

Figure 7.12.: DM_AF with timing tolerable task migration routine to reduce tardiness of a specified amount of timing tolerable tasks.

Analogously to the examination of complete task migration, subsequently the percentage of timing strict tasks and, consecutively, the size of the task set shall be modified. Resulting from this, from Figure 7.13 can be discovered that the algorithm performs better if the percentage of timing strict tasks increases. This can be explained by the fact that only timing tolerable tasks are migrated. If the amount of timing strict tasks grows and thus the number of timing tolerable tasks declines, it is obvious that the number of migrating tasks becomes smaller and, as a consequence, the probability of migrating these successfully rises.

Apart from this, Figure 7.14 portrays the outcome of experiments employing a set of 5, 10, 20, and 50 tasks. In analogous fashion to the complete task migration case, a set size of 10 tasks turns out to be optimal concerning this particular setting due to the fact that a smaller set size does not provoke an acceptance rate of 100% at a system utilization of 1% and, moreover, larger sets lead to a quicker decline of the acceptance rate contingent on the total system utilization.

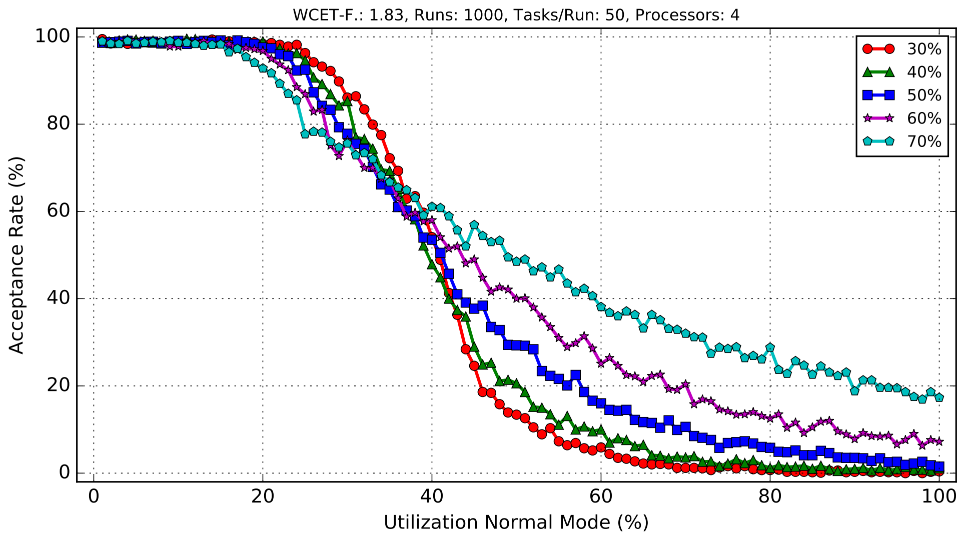

Figure 7.13.: DM_AF with timing tolerable task migration routine under varying percentage of timing strict tasks.

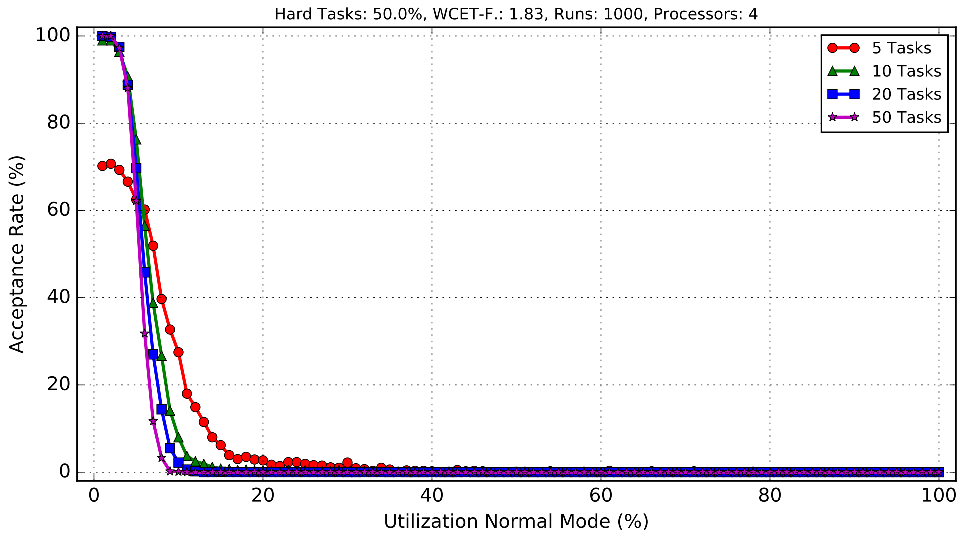

Figure 7.14.: DM_AF with timing tolerable task migration routine under varying task set size.

7.6. Conclusion

Within the scope of this master thesis, manifold ideas have been proposed how to transfer the Systems with Dynamic Real-Time Guarantees model by von der Brüggen et al. [24] onto a multiprocessor environment. Unfortunately, by means of experimental outcomes, the majority of these approaches turned out to be ineffective. Nevertheless, the deadline-monotonic arbitrary-fit scheduling policy proved to dominate all investigated strategies in terms of its task set acceptance rate as a function of the system utilization and can be even further improved by applying a task-splitting routine. While the task-splitting approach is capable of enhancing a scheduling policy's success, the highest-priority task splitting paradigm could be identified as entirely useless with respect to this particular field of application.

Moreover, the Multicore Systems with Dynamic Real-Time Guarantees model has been extended with respect to system reliability by creating a failure-proof version of the Multicore System with Dynamic Real-Time Guarantees, the l-Failure-Proof Multicore Systems with Dynamic Real-Time Guarantees, and providing an algorithm to obtain such systems. Notwithstanding, as the cost for this robustness, the number of possibly accepted task sets is restricted significantly.

Not least, a possibility has been suggested to reduce a specified amount of timing tolerable tasks' bounded tardiness by migrating them to another core in case on their respective processor all tasks are executed abnormally for a longer interval of time. Here again, the task set acceptance rate dependent on the system utilization decreases for the benefit of the desired objective.

In summary, the given problem statement has been solved successfully by providing miscellaneous opportunities to construct a Multicore System with Dynamic Real-Time Guarantees. What remains for future work, is the issue of modifying the highest-priority task splitting paradigm in such a way that it can be employed in addition to regular scheduling strategies in order to attain a significant improvement. Finally, it could be an interesting idea to combine both task splitting and task migration for the purpose of successfully scheduling task sets with higher system utilization under an l-Failure-Proof Multicore Systems with Dynamic Real-Time Guarantees.

A. Appendix

Algorithm 14 Deceasing Utilization Arbitrary-Fit (DU_AF)

Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or Not Possible

```
 1: Sort $\mathbf{T}$ by $U_i$ decreasingly
 2: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

 3: procedure Find Assignment($\mathbf{T}$, $\mathbf{S}$)
 4:     for each $\tau_t \in \mathbf{T}$ do
 5:         $\bar{\mathbf{S}} := \emptyset$
 6:         for each $\mathbf{T}_p \in \mathbf{S}$ do
 7:             if (Find Optimal Priority Assignment($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ Not Possible) then
 8:                 $\bar{\mathbf{S}} := \bar{\mathbf{S}} \cup \mathbf{T}_p$
 9:         if $\bar{\mathbf{S}} \neq \emptyset$ then
10:             $r$ := random element out of $[1, |\bar{\mathbf{S}}|]$
11:             $\mathbf{T}_r := \mathbf{T}_r \cup \{\tau_t\}$
12:             $\mathbf{T} := \mathbf{T} \setminus \{\tau_t\}$
13:         else return NOT POSSIBLE
14:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

Algorithm 15 Decreasing Utilization Best-Fit (DU_BF)

Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
 1: Sort $\mathbf{T}$ by $U_i$ decreasingly
 2: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

 3: procedure FIND ASSIGNMENT($\mathbf{T}$, $\mathbf{S}$)
 4:     for each $\tau_t \in \mathbf{T}$ do
 5:         $\mathbf{S}^* := \emptyset$
 6:         $assigned := false$
 7:         while ($assigned$ = false) do
 8:             $p :=$ FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
 9:             if ( $p = -1$) then return NOT POSSIBLE
10:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
11:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
12:                 $\mathbf{T} := \mathbf{T} \backslash \{\tau_t\}$
13:                 $assigned$ := true
14:             else
15:                 $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
16:                 if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
17:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

18: procedure FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
19:     $p_{best} := -1$
20:     $U_{sum,best} := 0$
21:     for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
22:         if ($U^N_{sum,p} \geq U_{sum,best}$) then
23:             $p_{best} := p$
24:             $U_{sum,best} := U^N_{sum,p}$
25:     return $p_{best}$
```

Algorithm 16 Decreasing Utilization Worst-Fit (DU_WF)

Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or Not Possible

```
 1: Sort $\mathbf{T}$ by $U_i$ decreasingly
 2: Find Assignment($\mathbf{T}$, $\mathbf{S}$)

 3: procedure Find Assignment($\mathbf{T}$, $\mathbf{S}$)
 4:     for each $\tau_t \in \mathbf{T}$ do
 5:         $\mathbf{S}^* := \emptyset$
 6:         $assigned := false$
 7:         while ($assigned$ = false) do
 8:             $p :=$ Find Processor with Min. Utilization($\mathbf{S}$, $\mathbf{S}^*$)
 9:             if ( $p = -1$) then return Not Possible
10:             if (Find Optimal Priority Assignment($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ Not Possible) then
11:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
12:                 $\mathbf{T} := \mathbf{T} \setminus \{\tau_t\}$
13:                 $assigned :=$ true
14:             else
15:                 $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
16:                 if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return Not Possible
17:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

18: procedure Find Processor with Min. Utilization($\mathbf{S}$, $\mathbf{S}^*$)
19:     $p_{best} := -1$
20:     $U_{sum,best} := 0$
21:     for ($p := 1$; $p \leq |\mathbf{S} \setminus \mathbf{S}^*|$; $p := p + 1$) do
22:         if ($U^N_{sum,p} \leq U_{sum,best}$) then
23:             $p_{best} := p$
24:             $U_{sum,best} := U^N_{sum,p}$
25:     return $p_{best}$
```

Algorithm 17 Criticality-Successive Deadline-Monotonic Arbitrary-Fit (CS_DM_AF)

Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
 1: Sort $\mathbf{T}^A_{hard}$ by $D_i$ increasingly
 2: Sort $\mathbf{T}^A_{soft}$ by $D_i$ increasingly
 3: Find Assignment($\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}$, $\mathbf{S}$)
 4: procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}$, $\mathbf{S}$)
 5:     for each $\tau_t \in \mathbf{T}^A_{hard}$ do
 6:         $\bar{\mathbf{S}} := \emptyset$
 7:         for each $\mathbf{T}_p \in \mathbf{S}$ do
 8:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
 9:                 $\bar{\mathbf{S}} := \bar{\mathbf{S}} \cup \mathbf{T}_p$
10:         if $\bar{\mathbf{S}} \neq \emptyset$ then
11:             $r$ := random element out of $[1, |\bar{\mathbf{S}}|]$
12:             $\mathbf{T}_r := \mathbf{T}_r \cup \{\tau_t\}$
13:             $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
14:         else return NOT POSSIBLE
15:     for each $\tau_t \in \mathbf{T}^A_{soft}$ do
16:         $\bar{\mathbf{S}} := \emptyset$
17:         for each $\mathbf{T}_p \in \mathbf{S}$ do
18:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
19:                 $\bar{\mathbf{S}} := \bar{\mathbf{S}} \cup \mathbf{T}_p$
20:         if $\bar{\mathbf{S}} \neq \emptyset$ then
21:             $r$ := random element out of $[1, |\bar{\mathbf{S}}|]$
22:             $\mathbf{T}_r := \mathbf{T}_r \cup \{\tau_t\}$
23:             $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
24:         else return NOT POSSIBLE
25:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

Algorithm 18 Criticality-Successive Deadline-Monotonic Best-Fit (CS_DM_BF)

Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
1: Sort $\mathbf{T}^A_{hard}$ by $D_i$ increasingly
2: Sort $\mathbf{T}^A_{soft}$ by $D_i$ increasingly
3: Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

4: procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
5:     for each $\tau_t \in \mathbf{T}^A_{hard}$ do
6:         $\mathbf{S}^* := \emptyset$
7:         $assigned := false$
8:         while ($assigned$ = false) do
9:             $p$ := FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
10:            if ( $p = -1$) then return NOT POSSIBLE
11:            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
12:                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
13:                $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
14:                $assigned$ := true
15:            else
16:                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
17:                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
18:     for each $\tau_t \in \mathbf{T}^A_{soft}$ do
19:         $\mathbf{S}^* := \emptyset$
20:         $assigned := false$
21:         while ($assigned$ = false) do
22:             $p$ := FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
23:             if ( $p = -1$) then return NOT POSSIBLE
24:             if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
25:                 $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
26:                 $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
27:                 $assigned$ := true
28:             else
29:                 $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
30:                 if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
31:     return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

32: procedure FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
33:     $p_{best} := -1$
34:     $U_{sum,best} := 0$
35:     for ($p := 1$; $p \leq |\mathbf{S} \backslash \mathbf{S}^*|$; $p := p + 1$) do
36:         if ($U^N_{sum,p} \geq U_{sum,best}$) then
37:             $p_{best} := p$
38:             $U_{sum,best} := U^N_{sum,p}$
39:     return $p_{best}$
```

Algorithm 19 Criticality-Successive Deadline-Monotonic Worst-Fit (CS_DM_WF)

```
Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$, or NOT POSSIBLE

1: Sort $\mathbf{T}^A_{hard}$ by $D_i$ increasingly
2: Sort $\mathbf{T}^A_{soft}$ by $D_i$ increasingly
3: Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

4: procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
5:     for each $\tau_t \in \mathbf{T}^A_{hard}$ do
6:         $\mathbf{S}^* := \emptyset$
7:         $assigned := false$
8:         while ($assigned$ = false) do
9:             $p$ := FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
10:            if ( $p = -1$) then return NOT POSSIBLE
11:            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
12:                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
13:                $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
14:                $assigned$ := true
15:            else
16:                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
17:                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
18:    for each $\tau_t \in \mathbf{T}^A_{soft}$ do
19:        $\tau_1$ := first element of $\mathbf{T}^A_{soft}$
20:        $\mathbf{S}^* := \emptyset$
21:        $assigned := false$
22:        while ($assigned$ = false) do
23:            $p$ := FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
24:            if ( $p = -1$) then return NOT POSSIBLE
25:            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
26:                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
27:                $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
28:                $assigned$ := true
29:            else
30:                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
31:                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
32:    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

33: procedure FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
34:    $p_{best} := -1$
35:    $U_{sum,best} := 0$
36:    for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
37:        if ($U^N_{sum,p} \leq U_{sum,best}$) then
38:            $p_{best} := p$
39:            $U_{sum,best} := U^N_{sum,p}$
40:    return $p_{best}$
```

Algorithm 20 Criticality-Successive Decreasing Utilization Arbitrary-Fit (CS_DU_AF)

Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or Not Possible

```
Sort $\mathbf{T}^A_{hard}$ by $U_i$ decreasingly
Sort $\mathbf{T}^A_{soft}$ by $U_i$ decreasingly
Find Assignment($\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}$, $\mathbf{S}$)

procedure Find Assignment($\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft}$, $\mathbf{S}$)
    for each $\tau_t \in \mathbf{T}^A_{hard}$ do
        $\bar{\mathbf{S}} := \emptyset$
        for each $\mathbf{T}_p \in \mathbf{S}$ do
            if (Find Optimal Priority Assignment($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ Not Possible) then
                $\bar{\mathbf{S}} := \bar{\mathbf{S}} \cup \mathbf{T}_p$
        if $\bar{\mathbf{S}} \neq \emptyset$ then
            $r :=$ random element out of $[1, |\bar{\mathbf{S}}|]$
            $\mathbf{T}_r := \mathbf{T}_r \cup \{\tau_t\}$
            $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
        else return Not Possible
    for each $\tau_t \in \mathbf{T}^A_{soft}$ do
        $\bar{\mathbf{S}} := \emptyset$
        for each $\mathbf{T}_p \in \mathbf{S}$ do
            if (Find Optimal Priority Assignment($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ Not Possible) then
                $\bar{\mathbf{S}} := \bar{\mathbf{S}} \cup \mathbf{T}_p$
        if $\bar{\mathbf{S}} \neq \emptyset$ then
            $r :=$ random element out of $[1, |\bar{\mathbf{S}}|]$
            $\mathbf{T}_r := \mathbf{T}_r \cup \{\tau_t\}$
            $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
        else return Not Possible
    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$
```

Algorithm 21 Criticality-Successive Decreasing Utilization Best-Fit (CS_DU_BF)

```
Input: $\mathbf{T}^A_{hard} = \tau_1, \dots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \dots, \tau_l$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \dots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$, or NOT POSSIBLE

  Sort $\mathbf{T}^A_{hard}$ by $U_i$ decreasingly
  Sort $\mathbf{T}^A_{soft}$ by $U_i$ decreasingly
  Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

  procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
      for each $\tau_t \in \mathbf{T}^A_{hard}$ do
          $\mathbf{S}^* := \emptyset$
          $assigned := false$
          while ($assigned$ = false) do
              $p :=$ FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
              if ( $p = -1$) then return NOT POSSIBLE
              if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
                  $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                  $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
                  $assigned$ := true
              else
                  $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                  if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
      for each $\tau_t \in \mathbf{T}^A_{soft}$ do
          $\mathbf{S}^* := \emptyset$
          $assigned := false$
          while ($assigned$ = false) do
              $p :=$ FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
              if ( $p = -1$) then return NOT POSSIBLE
              if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
                  $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                  $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
                  $assigned$ := true
              else
                  $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                  if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
      return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

  procedure FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
      $p_{best} := -1$
      $U_{sum,best} := 0$
      for ($p := 1$; $p \leq |\mathbf{S} \backslash \mathbf{S}^*|$; $p := p + 1$) do
          if ($U^N_{sum,p} \geq U_{sum,best}$) then
              $p_{best} := p$
              $U_{sum,best} := U^N_{sum,p}$
      return $p_{best}$
```

Algorithm 22 Criticality-Successive Decreasing Utilization Worst-Fit (CS_DU_WF)

Input: $\mathbf{T} := \tau_1, \ldots, \tau_n$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
Sort $\mathbf{T}^A_{hard}$ by $U_i$ decreasingly
Sort $\mathbf{T}^A_{soft}$ by $U_i$ decreasingly
Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
    for each $\tau_t \in \mathbf{T}^A_{hard}$ do
        $\mathbf{S}^* := \emptyset$
        $assigned := false$
        while ($assigned$ = false) do
            $p$ := FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
            if ( $p = -1$) then return NOT POSSIBLE
            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
                $assigned$ := true
            else
                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
    for each $\tau_t \in \mathbf{T}^A_{soft}$ do
        $\mathbf{S}^* := \emptyset$
        $assigned := false$
        while ($assigned$ = false) do
            $p$ := FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
            if ( $p = -1$) then return NOT POSSIBLE
            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
                $assigned$ := true
            else
                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

procedure FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
    $p_{best} := -1$
    $U_{sum,best} := 0$
    for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
        if ($U^N_{sum,p} \leq U_{sum,best}$) then
            $p_{best} := p$
            $U_{sum,best} := U^N_{sum,p}$
    return $p_{best}$
```

Algorithm 23 Deadline-Monotonic Hard Worst-Fit, Soft Best-Fit (DM_HWF_SBF)

Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or NOT POSSIBLE

```
Sort $\mathbf{T}^A_{hard}$ by $D_i$ increasingly
Sort $\mathbf{T}^A_{soft}$ by $D_i$ increasingly
Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
    for each $\tau_t \in \mathbf{T}^A_{hard}$ do
        $\mathbf{S}^* := \emptyset$
        $assigned := false$
        while ($assigned$ = false) do
            $p :=$ FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
            if ( $p = -1$) then return NOT POSSIBLE
            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
                $assigned$ := true
            else
                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
    for each $\tau_t \in \mathbf{T}^A_{soft}$ do
        $\mathbf{S}^* := \emptyset$
        $assigned := false$
        while ($assigned$ = false) do
            $p :=$ FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
            if ( $p = -1$) then return NOT POSSIBLE
            if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
                $assigned$ := true
            else
                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

procedure FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
    $p_{best} := -1$
    $U_{sum,best} := 0$
    for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
        if ($U^N_{sum,p} \leq U_{sum,best}$) then
            $p_{best} := p$
            $U_{sum,best} := U^N_{sum,p}$
    return $p_{best}$

procedure FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
    $p_{best} := -1$
    $U_{sum,best} := 0$
    for ($p = 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
        if ($U^N_{sum,p} \geq U_{sum,best}$) then
            $p_best := p$
            $U_{sum,best} := U^N_{sum,p}$
    return $p_{best}$
```

Algorithm 24 Decreasing Utilization Hard Best-Fit, Soft Worst-Fit (DU_HBF_SWF)

Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with m identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$

Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by P_p, or Not Possible

```
Sort $\mathbf{T}^A_{hard}$ by $U_i$ decreasingly
Sort $\mathbf{T}^A_{soft}$ by $U_i$ decreasingly
Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

procedure Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
    for each $\tau_t \in \mathbf{T}^A_{hard}$ do
        $\mathbf{S}^* := \emptyset$
        $assigned := false$
        while ($assigned$ = false) do
            $p :=$ Find Processor with Max. Utilization($\mathbf{S}$, $\mathbf{S}^*$)
            if ( $p = -1$) then return Not Possible
            if (Find Optimal Priority Assignment($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ Not Possible) then
                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
                $assigned$ := true
            else
                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return Not Possible
    for each $\tau_t \in \mathbf{T}^A_{soft}$ do
        $\mathbf{S}^* := \emptyset$
        $assigned := false$
        while ($assigned$ = false) do
            $p :=$ Find Processor with Min. Utilization($\mathbf{S}$, $\mathbf{S}^*$)
            if ( $p = -1$) then return Not Possible
            if (Find Optimal Priority Assignment($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ Not Possible) then
                $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
                $assigned$ := true
            else
                $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return Not Possible
    return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

procedure Find Processor with Min. Utilization($\mathbf{S}$, $\mathbf{S}^*$)
    $p_{best} := -1$
    $U_{sum,best} := 0$
    for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
        if ($U^N_{sum,p} \leq U_{sum,best}$) then
            $p_{best} := p$
            $U_{sum,best} := U^N_{sum,p}$
    return $p_{best}$

procedure Find Processor with Max. Utilization($\mathbf{S}$, $\mathbf{S}^*$)
    $p_{best} := -1$
    $U_{sum,best} := 0$
    for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
        if ($U^N_{sum,p} \geq U_{sum,best}$) then
            $p_best := p$
            $U_{sum,best} := U^N_{sum,p}$
    return $p_{best}$
```

Algorithm 25 Decreasing Utilization Hard Worst-Fit, Soft Best-Fit (DU_HWF_SBF)

```
Input: $\mathbf{T}^A_{hard} = \tau_1, \ldots, \tau_n, \mathbf{T}^A_{soft} = \tau_1, \ldots, \tau_l$, System with $m$ identical processors $\mathbf{S} := \mathbf{T}_1, \ldots, \mathbf{T}_m$
Output: System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$, or NOT POSSIBLE

  Sort $\mathbf{T}^A_{hard}$ by $U_i$ decreasingly
  Sort $\mathbf{T}^A_{soft}$ by $U_i$ decreasingly
  Find Assignment($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)

  procedure FIND ASSIGNMENT($\mathbf{T}^A_{hard}$, $\mathbf{T}^A_{soft}$, $\mathbf{S}$)
      for each $\tau_t \in \mathbf{T}^A_{hard}$ do
          $\mathbf{S}^* := \emptyset$
          $assigned := false$
          while ($assigned$ = false) do
              $p$ := FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
              if ( $p = -1$) then return NOT POSSIBLE
              if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
                  $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                  $\mathbf{T}^A_{hard} := \mathbf{T}^A_{hard} \backslash \{\tau_t\}$
                  $assigned$ := true
              else
                  $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                  if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
      for each $\tau_t \in \mathbf{T}^A_{soft}$ do
          $\mathbf{S}^* := \emptyset$
          $assigned := false$
          while ($assigned$ = false) do
              $p$ := FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
              if ( $p = -1$) then return NOT POSSIBLE
              if (FIND OPTIMAL PRIORITY ASSIGNMENT($\mathbf{T}_p \cup \{\tau_t\}$) $\neq$ NOT POSSIBLE) then
                  $\mathbf{T}_p := \mathbf{T}_p \cup \{\tau_t\}$
                  $\mathbf{T}^A_{soft} := \mathbf{T}^A_{soft} \backslash \{\tau_t\}$
                  $assigned$ := true
              else
                  $\mathbf{S}^* := \mathbf{S}^* \cup \mathbf{T}_p$
                  if ($|\mathbf{S}^*| = |\mathbf{S}|$) then return NOT POSSIBLE
      return System $\mathbf{S}$ with each subsystem $\mathbf{T}_p$, $0 < p \leq m$, ordered by $P_p$

  procedure FIND PROCESSOR WITH MIN. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
      $p_{best} := -1$
      $U_{sum,best} := 0$
      for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
          if ($U^N_{sum,p} \leq U_{sum,best}$) then
              $p_{best} := p$
              $U_{sum,best} := U^N_{sum,p}$
      return $p_{best}$

  procedure FIND PROCESSOR WITH MAX. UTILIZATION($\mathbf{S}$, $\mathbf{S}^*$)
      $p_{best} := -1$
      $U_{sum,best} := 0$
      for ($p := 1;\ p \leq |\mathbf{S} \backslash \mathbf{S}^*|;\ p := p + 1$) do
          if ($U^N_{sum,p} \geq U_{sum,best}$) then
              $p_best := p$
              $U_{sum,best} := U^N_{sum,p}$
      return $p_{best}$
```

List of Figures

1.1. A feasible schedule according to the rate-monotonic policy without error occurrence. 1

1.2. Task τ_3 misses its deadline due to multiple re-executions of τ_1 by reason of fault recovery (light red). 2

2.1. Typical parameters of a real-time task [7]. 6

2.2. An implicit-deadline task set. 6

2.3. A constrained-deadline task set. 7

2.4. An arbitrary-deadline task set. 7

6.1. $D_i \geq F \cdot T_j + C_j$ [18] 44

6.2. $D_i \leq F \cdot T_j + C_j$ [18] 44

7.1. Comparison of all scheduling policies based on the first-fit bin-packing heuristic developed in this thesis. 56

7.2. Comparison of all scheduling policies based on the arbitrary-fit bin-packing heuristic developed in this thesis. 57

7.3. Comparison of all scheduling policies based on the best-fit bin-packing heuristic developed in this thesis. 58

7.4. Comparison of all scheduling policies based on the worst-fit bin-packing heuristic developed in this thesis. 58

7.5. Comparison of all deadline-monotonic criticality-successive scheduling policies developed in this thesis. 59

7.6. Comparison of all criticality-successive scheduling policies developed in this thesis preordered according to decreasing task utilization. 59

7.7. DM_AF with task splitting routine compared to DM_AF. 61

7.8. DM_AF with highest-priority task splitting routine compared to DM_AF. . 61

7.9. DM_AF with complete task migration routine for the establishment of a 1-Failure-Proof Multicore Systems with Dynamic Real-Time Guarantees compared to DM_AF. 62

7.10. DM_AF with complete task migration routine under varying percentage of timing strict tasks. 63

7.11. DM_AF with complete task migration routine under varying task set size. . 63

7.12. DM_AF with timing tolerable task migration routine to reduce tardiness of a specified amount of timing tolerable tasks. 64
7.13. DM_AF with timing tolerable task migration routine under varying percentage of timing strict tasks. 65
7.14. DM_AF with timing tolerable task migration routine under varying task set size. 65

List of Algorithms

1. Audsley's Optimal Priority Assignment algorithm for single-core mixed-criticality systems. 11

2. Feasible Priority Assignment Algorithm by von der Brüggen et al. [24] . . . 20

3. Deadline-Monotonic Partitioning [4] 23

4. Deadline-Monotonic First-Fit (DM_FF) 32
5. Deadline-Monotonic Arbitrary-Fit (DM_AF) 32
6. Deadline-Monotonic Best-Fit (DM_BF) 33
7. Deadline-Monotonic Worst-Fit (DM_WF) 34
8. Decreasing Utilization First-Fit (DU_FF) 35
9. Criticality-Successive Deadline-Monotonic First-Fit (CS_DM_FF) 37
10. Criticality-Successive Decreasing Utilization First-Fit (CS_DU_FF) 38
11. Deadline-Monotonic Hard Best-Fit, Soft Worst-Fit (DM_HBF_SWF) 40
12. Semi-Partitioned Scheduling Policy with Task Splitting (TS) 46
13. Semi-Partitioned Scheduling Policy with Highest-Priority Task Splitting (HPTS) 48

14. Deceasing Utilization Arbitrary-Fit (DU_AF) 67
15. Decreasing Utilization Best-Fit (DU_BF) 68
16. Decreasing Utilization Worst-Fit (DU_WF) 69
17. Criticality-Successive Deadline-Monotonic Arbitrary-Fit (CS_DM_AF) . . . 70
18. Criticality-Successive Deadline-Monotonic Best-Fit (CS_DM_BF) 71
19. Criticality-Successive Deadline-Monotonic Worst-Fit (CS_DM_WF) 72
20. Criticality-Successive Decreasing Utilization Arbitrary-Fit (CS_DU_AF) . . 73
21. Criticality-Successive Decreasing Utilization Best-Fit (CS_DU_BF) 74
22. Criticality-Successive Decreasing Utilization Worst-Fit (CS_DU_WF) 75
23. Deadline-Monotonic Hard Worst-Fit, Soft Best-Fit (DM_HWF_SBF) 76
24. Decreasing Utilization Hard Best-Fit, Soft Worst-Fit (DU_HBF_SWF) . . . 77
25. Decreasing Utilization Hard Worst-Fit, Soft Best-Fit (DU_HWF_SBF) . . . 78

Bibliography

[1] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 33–40, July 2003.

[2] N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Inf. Process. Lett.*, 79(1):39–44, May 2001.

[3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[4] Sanjoy Baruah and Nathan Fisher. The Partitioned Multiprocessor Scheduling of Sporadic Task Systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, pages 321–329, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Sanjoy Baruah and Steve Vestal. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 147–155, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Alan Burns and Robert I. Davis. Mixed Criticality Systems – a Review. York, UK, July 2016. Department of Computer Science, University of York. Eighth edition.

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

[8] J. J. Chen. Partitioned Multiprocessor Fixed-Priority Scheduling of Sporadic Real-Time Tasks. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251–261, July 2016.

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

[10] Robert I. Davis and Alan Burns. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[11] François Dorin, Pascal Richard, Michaël Richard, and Joël Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46(3):305–331, 2010.

[12] Rolf Ernst and Marco Di Natale. Mixed Criticality Systems — A History of Misconceptions? *IEEE Design & Test*, 33(5):65–74, October 2016.

[13] Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, and Tovar Eduardo. How realistic is the mixed-criticality real-time system model? In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 139–148, New York, NY, USA, 2015. ACM.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[15] M. A. Haque, H. Aydin, and D. Zhu. Real-time scheduling under fault bursts with multiple recovery strategy. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 63–74, April 2014.

[16] International Electrotechnical Commission (IEC). Analysis Techniques for System Reliability — Procedure for Failure Mode and Effects Analysis (FMEA). International Standard IEC 60812:2006, 2006.

[17] David S. Johnson. Fast Algorithms for Bin Packing. *J. Comput. Syst. Sci.*, 8(3):272–314, June 1974.

[18] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32, April 2009.

[19] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 239–248, July 2009.

[20] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 166–171, Dec 1989.

[21] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[22] John A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *Computer*, 21(10):10–19, October 1988.

[23] Steve Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 239–243, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Georg von der Brüggen, Kuan-Hsun Chen, Wen-Hung Huang, and Jian-Jia Chen. Systems with Dynamic Real-Time Guarantees in Uncertain and Faulty Execution Environments. In *Real-Time Systems Symposium (RTSS)*, Porto, Portugal, Nov. 29 - Dec. 2 2016.

[25] Hao Xu and Alan Burns. Semi-partitioned Model for Dual-core Mixed Criticality System. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 257–266, New York, NY, USA, 2015. ACM.

Eidesstattliche Versicherung

______________________________ ____________________

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

____

____

____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

__________________________ _______________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____________________________ _________________________

Ort, Datum Unterschrift