Systems with Dynamic Real-Time Guarantees in Uncertain and Faulty Execution Environments

Georg von der Brüggen, Kuan-Hsun Chen, Wen-Hung Huang, and Jian-Jia Chen
Department of Informatics
TU Dortmund University, Germany

***Abstract*—In many practical real-time systems, the physical environment and the system platform can impose uncertain execution behaviour to the system. For example, if transient faults are detected, the execution time of a task instance can be increased due to recovery operations. Such fault recovery routines make the system very vulnerable with respect to meeting hard real-time deadlines. In theory and in practical systems, this problem is often handled by aborting *not so important* tasks to guarantee the response time of the *more important* tasks. However, for most systems such faults occur rarely and the results of *not so important* tasks might still be useful, even if they are a bit late. This implicates to not abort these *not so important* tasks but keep them running even if faults occur, provided that the *more important* tasks still meet their hard real time properties. In this paper, we present *Systems with Dynamic Real-Time Guarantees* to model this behaviour and determine if the system can provide *full timing guarantees* or *limited timing guarantees* without any online adaptation after a fault occurred. We present a schedulability test, provide an algorithm for optimal priority assignment, determine the maximum interval length until the system will again provide *full timing guarantees* and explain how we can monitor the system state online. The approaches presented in this paper can also be applied to mixed criticality systems with dual criticality levels.**

1 Introduction

Continuous technology scaling has introduced multiple threats that reduce the reliability of computing hardware, not only in the memory hierarchy but also in the logic components [4], [25], [29]. Such reliability threats include soft errors, aging, and process variations. Specifically, the soft errors known as transient faults of the computing hardware or the memory subsystem are typically due to high-energy particle strikes. Without fault-tolerance design in mind, a transient fault can corrupt the correct application execution state, and lead to wrong execution results or even system failure. There are several well-known techniques to handle such faulty execution behaviour, both from hardware and software perspectives.

- Retry/Re-execution [24]: to eliminate the effects of a fault, re-execution can be adopted. However, the number of re-executions per job must be limited if hard real-time guarantees must be provided. One way is to set the maximal trials based on the acceptable quality of service.
- Checkpoint-based recovery [13]: The idea of checkpointing is to save the state of task instances at multiple points within the execution. If faults are detected at a checkpoint, a rollback to the task's execution state at the previous checkpoint is performed. A sanity or acceptance test must be enforced to check the values of data variables and contents of registers at every checkpoint. This ensures that only a smaller part of the task has to be re-executed but leads to a higher overhead due to more acceptance tests, checkpoint creation, and possible rollbacks.
- Hardened components: In most hardware-based techniques, the regular component can be hardened to prevent faults to affect the execution. This requires additional gates or die area. Such techniques include, for example, voting mechanisms [17], instruction set customizations [8], simultaneous and redundant threads [28].
- Vulnerability-aware adaptation of Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR): DMR and TMR are well-known redundancy approaches in hardware and software perspectives. With different amounts of replicas, the protected task can be enhanced with different levels of dependability.

When the fault rate is usually expected to be low, the software techniques presented above can be applied; otherwise, the hardware should have been hardened to be more robust against the potential faults. Therefore, since faults occur rarely, we should not consider the additional workload due to the fault recovery mechanisms as normal execution behaviour.

Consider the following simple execution model. A control task is protected by a re-execution mechanism. After finishing the computation, a fault detection mechanism verifies whether the results are correct. If no fault is detected, the output of the first run of the task is applied; otherwise, the task is re-executed and the new output is applied as the result of the computation without an additional fault detection. Suppose that the worst-case execution time (WCET) of the first run excluding the time for fault detection is X time units and that the fault detection takes additional 20% of WCET. This means that in the normal situation when there is no fault, the WCET of this task is $1.2X$ time units, but the WCET of this task becomes $2.2X$ time units when a fault is detected and the task has to be re-executed.

In all fault tolerance and recovery mechanisms mentioned above, the prolongment of the WCET is generally unavoidable. Depending on the applied fault detection, fault recovery, and checkpointing mechanisms, the ratio of the WCET with recovery to the WCET without recovery can differ largely. On the one hand, re-executing the whole task once (or twice) with a fault detection at the end of the task's execution will lead to nearly double (or triple) the WCET if recovery has to be done, as the whole task has to be re-executed while the fault detection mechanism will usually have a much smaller runtime than the task itself[1]. On the other hand, checkpointing will usually only lead to a smaller increase of the WCET if error recovery has to be done, as only a part of the task has to be re-executed. However, checkpointing may introduce a larger overhead for the fault detection than in the case where the whole task is re-executed, as a number of checkpoints have to be created during the execution and fault detection has to be done for every checkpoint.

[1] If not, one could use sequential DMR directly.

Since faults are supposed to happen rarely, it would be more meaningful **not** to consider the WCET with recovery all the time to avoid over-dimensioning the system resources. However, if so, the problem is to handle the *occasional/rare fault events properly*. Specifically, we would like to ensure that such fault events do not *completely* destroy the timing guarantees that can be provided for real-time applications. Such fault events can be modelled as overshoots. Kumar and Thiele [21] proposed a Rare-Event with Settling-Time (REST) model, and explored how to quantify the response time bound by considering overshoots and the settling-time of overshoots, based on real-time calculus. Quinton et al. [26], [27], Hammadeh et al. [19] and Wu et al. [32] developed methods to compute the worst-case response time and the number of potential deadline misses in a given sequence of events with potential rare events that make the system overloaded. All papers mentioned above assumed that deadline misses are tolerable and tried to quantify the deadline misses.

In this paper, we consider the alternative to only allow some predefined tasks to miss their deadline occasionally, if some tasks overshoot in uncertain or faulty execution environments. This means, we would like to maintain the deadline satisfactions for *all* tasks when the system runs *normally*. If faults occur, we allow that some *timing tolerable* tasks miss their deadlines while still giving timing guarantees for the *timing strict* tasks. The *timing tolerable* tasks should still have bounded tardiness even if faults occur. For example, in an unmanned aerial vehicle (UAV) or a service robot, the tasks in the flight control system and in the body balancing system, respectively, are more important than the tasks in the surveillance system. If the system runs normally, all tasks are important and should meet their deadlines; however, the tasks in the surveillance system can tolerate some deadline misses, if faults take place. As soon as the impact of the faults has been resolved, the system should return to run all tasks normally and all deadlines should be satisfied again.

We propose a general model and use error recovery as an important special case to motivate this paper. However, the applicability of the model is quite general to consider rare overshoots and overloaded situations for real-time sporadic task systems. With different execution behaviour, the system state may change between normal execution and abnormal execution. This leads to a natural connection to mixed-criticality systems [31]. Identifying whether the system runs normally or abnormally is needed if the scheduling policy has to be adaptive. However, such identifications and adaptations can be costly. A simple solution is to use fixed-priority preemptive scheduling regardless of the faults, since it is assumed that faults only occur rarely. If the system behaves correctly, we only need an online monitor mechanism to safely identify the execution correctness with respect to timing.

Connections to Mixed-Criticality Systems: Our studied problem is highly linked to the topic of mixed-criticality systems that has been widely studied since the seminal research by Vestal in [31] in 2007. Specifically, in a dual-criticality system, the system can be in the high-criticality or in the low-criticality mode. If the system enters the high-criticality mode, only timing guarantees for tasks that are marked as high-criticality have to be provided. If the system is in the low-criticality mode, all tasks have to be guaranteed to meet their deadlines. Therefore, we can imagine that the system is in the low-criticality mode if *full timing guarantees* are needed and in the high-criticality mode if only *limited timing guarantees* are provided. Please refer to the survey by Burns and Davis [9] for a detailed overview and review of mixed-criticality systems.

However, in most research results regarding mixed-criticality systems, the identification of the criticality mode of the system is not considered, i.e., such mode changes are assumed to be provided. The system only switches from the low-criticality to the high-criticality mode once, without ever returning to the low-criticality mode. Moreover, in most studies, the low-criticality tasks are considered to be second-class tasks that are either ignored, skipped, or run with best efforts as background tasks. The model has received criticism as system engineers claim that it does not match their expectations. Specifically,

- the low-criticality tasks should not be abandoned, as the low-criticality tasks are still critical, and
- it should be possible for the system to return from the high-criticality mode to the low-criticality mode after a sufficient amount of time.

For such criticism, please refer to Esper et al. [16], Ernst and Di Natale [15], and Burns and Davis [9, Section 6].

The proposed *System with Dynamic Real-Time Guarantees* does not have these drawbacks, as low-criticality tasks are never abandoned but run with guaranteed bounded tardiness and the system can dynamically change from the low criticality mode to the high criticality mode and vise versa as often as necessary. Furthermore, as neither online adaptation nor dynamic scheduling decisions or priority changes are part of the proposed system model, a *System with Dynamic Real-Time Guarantees* can be scheduled by using fixed priority scheduling implementations provided in current Real-Time Operating Systems.

Our Contributions: This paper focuses on providing dynamic real-time guarantees in uncertain and faulty execution environments without any online adaptation.

- We provide a general model called *Systems with Dynamic Real-Time Guarantees* to be adopted when uncertain execution behaviour of tasks is imposed by the environment and the system platform. We define the system and the related terms *full timing guarantees* and *limited timing guarantees* in Section 2 and provide an exact schedulability test in Section 3. In Section 4, we prove several important properties of an optimal priority order, present an algorithm to determine such a priority order, and prove its optimality.
- We analyse how to calculate the maximum interval length until we can return to *full timing guarantees* again when the system is only providing *limited timing guarantees* due to some abnormal execution behaviour in the past in Section 5. We provide a way to monitor the system state and approximate the amount of time needed to go back to *full timing guarantees* in Section 6.
- In the evaluations in Section 7, we explore how our optimal fixed-priority assignment performs compared to other scheduling policies, i.e., Rate Monotonic, Criticality Monotonic, and EDF-VD by Baruah et. al. [3], a state-of-the-art approach with online adaption and dynamic-priority scheduling for Mixed-Criticality Systems. The evaluations show that using our optimal priority assignment for *Systems with Dynamic Real-Time Guarantees* does not sacrifice much with regards to schedulability

compared to EDF-VD in most settings. For some settings, especially when the difference between the WCETs in the high criticality and the low criticality modes is larger, our approach can schedule more task sets than EDF-VD for higher utilization values. This suggests that fixed-priority scheduling without any online adaptation is a sensible way to schedule Mixed-Criticality Systems as well, if the priority assignment is done carefully. This suggestion is supported by the fact that fixed-priority scheduling has less runtime overhead than dynamic priority scheduling. Furthermore, our strategy does not drop any low criticality tasks when only *limited timing guarantees* are provided, and still guarantees bounded tardiness. Most Mixed-Criticality approaches, including EDF-VD, allow to discard low-criticality tasks in the high-criticality mode.

2 System Model

2.1 Task Model and Execution Uncertainty

This paper considers n independent sporadic real-time tasks $\mathbf{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ in a uniprocessor system. Each task τ_i can release an infinite number of jobs (also called task instances) under the given minimum inter-arrival time (temporal) constraint T_i. This means if at time θ_a a job of task τ_i arrives, the next instance of the task must arrive no earlier than $\theta_a + T_i$. The relative deadline of task τ_i is D_i, i.e., a task instance released at θ_a must be finished before $\theta_a + D_i$. If $D_i = T_i \ \forall \tau_i$, a task set is called an *implicit-deadline* task set, while a task set is a *constrained-deadline* task set if $D_i \leq T_i \ \forall \tau_i$. If a task set has neither implicit nor constrained deadlines, it is called an *arbitrary-deadline* task set. We will only consider constrained- and implicit-deadline task sets in this paper.

We consider a platform with uncertain execution behaviour, imposed by the physical environment and the execution platform, where tasks have two execution modes. The *Worst Case Execution Time* (WCET) differs depending on these modes. We assume tasks have a smaller WCET most of the time and a larger WCET for rare special cases. We refer to these two different execution behaviours of a task as *normal* executions and *abnormal* executions. In an abstract view, each task has two different versions of WCET, C_i^N and C_i^A with $C_i^A \geq C_i^N$, e.g., for tasks with potential error recovery, C_i^N is the WCET of τ_i when no fault occurred during τ_is execution. The time that is needed for fault detection is part of C_i^N, as in our model the fault detection has to be done every time a job finishes its calculation or at the predefined checkpoints.

C_i^A is the worst-case execution time of task τ_i when the task runs abnormally, i.e., requires fault recovery. These values may differ largely, depending on the used fault detection and recovery model. For example, if we assume the fault detection to cost 20% of the task execution time, one re-execution would prolong C_i^A by $\frac{2.2}{1.2} \approx 1.83$ compared to C_i^N, while for two re-executions we get $C_i^A = \frac{3.4}{1.2} \cdot C_i^N \approx 2.83 \cdot C_i^N$. If we adopt checkpointing, only a smaller part of the task has to be re-executed, e.g., 20%, but the checkpointing has a higher effect on the WCET, e.g., 40% of the WCET in normal mode. In this case $C_i^A = \frac{1.6}{1.4} \cdot C_i^N \approx 1.14 \cdot C_i^N$. These are the 3 settings for the relation between C_i^A and C_i^N we use in the simulations.

We assume that we do not know in what manner a job will be executed at the moment the job arrives or starts its execution, but at some time point during or at the end of the execution process, i.e., we do not know if a fault will be detected and a fault recovery has to be done when the job starts. We assume that at least for some *timing strict* tasks missing a deadline will have catastrophic consequences. This means, releasing these tasks in a lower rate, allowing some of the jobs to be abandoned or miss the deadline, and enlarging the tasks deadline are not valid reactions to abnormal execution. Thus, for each task, D_i and T_i are identical in the normal and the abnormal mode.

Throughout this paper, we refer to the *normal utilization* of task τ_i as $U_i^N = C_i^N / T_i$ and to the *abnormal utilization* of task τ_i as $U_i^A = C_i^A / T_i$. The total or system utilization in the normal mode will be denoted $U_{sum}^N = \sum_{\{\tau_i \in \mathbf{T}\}} U_i^N$ and the total utilization in the abnormal mode is $U_{sum}^A = \sum_{\{\tau_i \in \mathbf{T}\}} U_i^A$.

We assume a fixed-priority scheduling policy where the tasks' priorities are the same regardless of the mode the tasks are executed in. One reason is that many real-time operations systems only support one fixed priority assigned to a task that cannot be changed during runtime. Another reason is that it is not clear if a job will be executed in the normal or the abnormal mode when it starts executing. This is only clear at the moment a fault is detected. If different priorities for normal and abnormal executions are assumed, this means the priority would be changed during the task execution.

For a job of τ_i we denote the response time of the j-th job with $R_{i,j}$. The Worst Case Response Time (WCRT) of τ_i will be denoted as R_i^N if all tasks are executed in the normal mode and as R_i^A if all tasks are executed in the abnormal mode. The tardiness of τ_i is defined as $E_i = R_i^A - D_i$, i.e., the maximum amount of time a job of τ_i stays active after its deadline.

2.2 Dynamic Real-Time Guarantees

When some tasks are executed abnormally in a certain time interval, informally, we can also say that the system is abnormal in that interval. The issues of abnormal executions are possible deadline misses due to the longer task execution times in the abnormal mode even if the schedulability in the normal mode is ensured. In theory and in practical systems this problem is often handled by aborting *timing tolerable* tasks to guarantee the response time of the *timing strict* tasks. We call the *timing strict* tasks $\mathbf{T}_{\text{hard}}^A$ and the *timing tolerable* tasks $\mathbf{T}_{\text{soft}}^A$ and assume the partition of $\mathbf{T}$ in $\mathbf{T}_{\text{hard}}^A$ and $\mathbf{T}_{\text{soft}}^A$ to be given.

Generally, aborting tasks is not preferable behaviour for multiple reasons. The main reason is that the results of the aborted tasks may still be useful, even if they are a bit late, as long as the timing behaviour of the *timing strict* tasks is not jeopardized. Therefore, in this case, we want to reduce the guarantees for the *timing tolerable* tasks from hard real-time guarantees to bounded tardiness. However, the user should be informed that the results of *timing tolerable* tasks may be late. Another reason is that executing one or more tasks in the abnormal mode will not necessarily lead to missed deadlines. Figure 6 in the Appendix gives a concrete example. Lastly, even if the system is proven safe in the normal mode, the trivial approach of abandoning the tasks in $\mathbf{T}_{\text{soft}}^A$ at the moment a fault occurs that would jeopardize the deadline of a task in $\mathbf{T}_{\text{hard}}^A$ only works, if all tasks $\tau_i \in \mathbf{T}_{\text{soft}}^A$ have lower priority than all tasks $\tau_i \in \mathbf{T}_{\text{hard}}^A$ as shown in Example 1.

Example 1. *Assume $\varepsilon > 0$ but very small and two tasks, $\tau_1 \in \mathbf{T}_{soft}^A$ with $C_1^N = 6$, $C_1^A = 6 + \varepsilon$, $T_1 = D_1 = 16$ and $\tau_2 \in \mathbf{T}_{hard}^A$ with $C_2^N = 11$, $C_2^A = 12 + \varepsilon$, $T_1 = D_1 = 24$ that are scheduled according to Rate Monotonic (RM), i.e., τ_1 has the higher priority. This task set is schedulable in the*

normal mode but if we assume the critical instant of τ_2 and a fault happening at $t \in [22, 23]$, then τ_2 will miss its deadline as the first two jobs of τ_1 are already completed.

Based on our argumentation, we will call it a system with *full timing guarantees*, if we can guarantee that all tasks that are currently ready to be executed will always meet their deadline if no further faults occur. We will call it a system with *limited timing guarantees*, if we can only guarantee the timing behaviour for the *timing strict* tasks, while for the *timing tolerable* tasks we will guarantee bounded tardiness. This is formalized in the following definition.

Definition 1. *[System with Dynamic Real-Time Guarantees]*

Consider a set $\mathbf{T}$ of 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 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:*

- *$\mathbf{T}^A_{hard} \subseteq \mathbf{T}$: Each task $\tau_i \in \mathbf{T}^A_{hard}$ is required to meet the hard relative deadline.*
- *$\mathbf{T}^A_{soft} \subseteq \mathbf{T}$: Each task $\tau_i \in \mathbf{T}^A_{soft}$ must have bounded tardiness, i.e., $0 \leq E_i < \gamma$ for a 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 $\mathbf{T}^A_{hard} \cup \mathbf{T}^A_{soft} = \mathbf{T}$.

We assume the partition into $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ to be given, as a computer cannot decide whether deadline misses will lead to potentially catastrophic consequences. We will call a task set *feasible* or *feasibly schedulable* as a *System with Dynamic Real-Time Guarantees*, if for the given partition and the given priority ordering, the conditions for *full timing guarantees* and *limited timing guarantees* both hold. While this definition can be used for both preemptive and non-preemptive scheduling, we only consider preemptive fixed priority scheduling here.

We already know that executing tasks in the abnormal mode will not necessarily lead to deadline misses. On the other hand, after faults occurred, some task instances may still miss their deadlines even if no abnormal job is still in the system and all tasks are executed normally, as some remaining workload, which was postponed due the abnormal execution of higher priority tasks or an earlier job of the tasks itself, can push back the time intervals where the job executes as shown in Figure 1. Faults are marked by $\lightning$ and $C^A_i = 2 \cdot C^N_i$. The first jobs of τ_2 and τ_3 miss their deadlines directly due to the additional workload created by the faults in the jobs of τ_1 and τ_2. The workload that is executed after the deadline is labeled with a black cross. After the first job of τ_2 finishes, all remaining tasks in the system execute normally. However, the second job of τ_3 (purple) misses its deadline due to the workload added by the late execution of the first job of τ_2 and τ_3. Note that the amount of workload created by higher priority tasks and the second job of τ_3 in $[12, 24]$ would not be large enough to let the second job of τ_3 miss its deadline. This only happens due to the remaining workload from the first job of τ_3 in $[12, 24]$ and thus is called a self-pushing phenomenon.

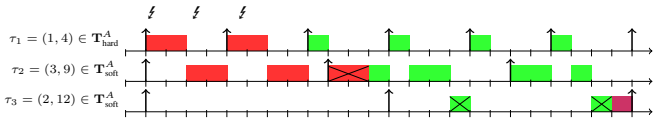

Fig. 1: Tasks can miss a deadline due to self-pushing.
The first jobs of τ_2 and τ_3 miss their deadlines due to this additional workload. A black cross labels the late executions. The purple job of τ_3 misses its deadline due to the self-pushing by its earlier instance.

We assume a fixed priority ordering P of the tasks and let $P(\tau_i)$ denote the priority of task τ_i. $P(\tau_i) < P(\tau_j)$ means τ_i has a higher priority than τ_j. For brevity we define:

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

The question remains what information should be displayed to the system user at which point in time. In general, the user will not be interested in the information that faults happened and fault recovery is performed as long as the timing behaviour is not affected and all results can be trusted. In this paper, we will focus on the timing behaviour and assume that the results of tasks can always be trusted, i.e., that in the abnormal mode the error recovery mechanism will always lead to a correct result or to a partially incorrect but still acceptable result. The information that the currently displayed result may have some delay should be displayed, as well as the actual delay and the time interval until the moment the system will return to provide *full timing guarantees* if no further faults occur.

2.3 Fault-Tolerance and Mixed-Criticality

As explained in Section 1, our studied problem has a clear link to mixed-criticality systems. However, the existing results in the literature of mixed-criticality systems typically abandoned the low-criticality tasks and assumed that the criticality mode of the system is given. These features are not suitable for providing a *System with Dynamic Real-Time Guarantees*.

As reported by Burns and Davis in their survey paper [9, Section 6], instead of ignoring the low criticality tasks during a criticality mode change, there have been several research efforts that focused on changing properties of the low criticality tasks, e.g., reduce their priorities, increase their periods/deadlines, impose only weakly hard real-time constraints on them, decrease their computation times, etc. Specifically, the concept to impose only weakly hard real-time constraints studied in [18] is related to the problem studied here, as the performance is downgraded for the low criticality task by using the (m, k)-firm real-time property; i.e., $(k - m)$ out of k jobs have to meet their deadlines. However, the approaches in [18] and others reported in [9, Section 6] require online *adaptive scheduling* to handle the scheduling properly.

"*Although there is this clear link between fault-tolerant systems (FTS) and mixed-criticality systems (MCS) there has not yet been much work published that directly addresses fault-tolerant mixed criticality systems,*" is reported in the survey by Burns and Davis [9, Section 5.2]. Related researches also considered mixed-criticality systems combined with fault tolerance. For example, Thekkilakattil et al. [30] used EDF scheduling to ensure that 1) the high criticality tasks are feasible under the presence of an error burst and 2) the low criticality jobs/tasks are feasible if they are not hit by the error burst. Huang et al. [20] modeled the impact of the hardware

transient faults as mixed-criticality behaviour and presented analysis techniques to bound the effects of task killing and service degradation on the system with online adaptation.

To the best of our knowledge, there was no previous research discussing whether such adaptive scheduling policies are always needed. The important property of the problem studied in this paper is **not** to provide any run-time (online) adaptation. This leads to the nice consequence that the scheduling algorithm can be robust regardless of mode changes, if the schedule is verified offline to be always feasible to provide dynamic timing guarantees. The impact on the system behaviour, i.e., the impact due to the tardiness of the tasks in $\mathbf{T}^A_{\text{soft}}$, can be analysed in advance without disturbing the run-time system. The system arbitrarily changes between normal and abnormal execution due to the impact of the physical environment, e.g., transient faults are encountered or not. If such an impact is acceptable in the system behaviour, e.g., guaranteeing bounded tardiness instead of hard deadlines in $\mathbf{T}^A_{\text{soft}}$ is sufficient as long as this affects only 1% of the system runtime, then there is no need for any run-time adaptation.

3 Schedulability Test

Assume to be given: a task set $\mathbf{T}$, a partition of $\mathbf{T}$ into two subsets $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$, and a fixed priority order P. A sufficient schedulability test to determine if $\mathbf{T}$ is a *System with Dynamic Real-Time Guarantees* (Definition 1), if scheduled according to P, must test the following three conditions:

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

To test the schedulability of a preemptive task set with constrained deadlines under a fixed priority assignment, we can apply Time Demand Analysis (TDA) [22] as an exact test with pseudo-polynomial runtime. It determines the schedulability of a task τ_k assuming the priority order of the task set is given and the schedulability of all tasks in $hp(\tau_k)$ is ensured already:

Definition 2 (Time Demand Analysis [22])**.** *τ_k is schedulable if the following equation holds:*

$$\exists t \text{ with } 0 < t \leq D_k \text{ and } C_k + \sum_{\tau_i \in hp(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t \quad (1)$$

If this holds true for all $\tau_k \in \mathbf{T}$, the task set is schedulable under the preemptive fixed-priority scheduling. We can use the following schedulability tests to determine the schedulability as a System with Real-Time Service Level Guarantees.

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

1) Full timing guarantees hold, if $\mathbf{T}$ can be scheduled according to Time Demand Analysis (TDA) [22] 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 $\tau_i \in \mathbf{T}^A_{hard}$ will meet their hard deadlines, if they can be proven to be schedulable by TDA [22] when all tasks are executed in the abnormal mode, i.e., $C_i = C_i^A \; \forall \tau_i$.
3) Each task $\tau_i \in \mathbf{T}^A_{soft}$ has bounded tardiness if $U^A_{sum} \leq 1$.

Proof: We only sketch the proof here and give some details in the Appendix.

1) Follows directly, as making sure that all tasks meet their deadline when no faults occur is identical to tasks having only one WCET where TDA is an exact test.
2) When the system runs with *limited timing guarantees* only $\tau_i \in \mathbf{T}^A_{\text{hard}}$ have to be tested. The workload created by tasks in $hep(\tau_i)$ is maximized if all jobs of all tasks are affected by a fault as we assume $C_i^A \geq C_i^N \forall \tau_i \in hep(\tau_i)$.
3) If $U^A_{sum} \leq 1$ the maximum total workload created in one hyperperiod[2] is smaller or equal to the period length. Thus the latest possible time for a job to finish is one hyperperiod after its release if $U^A_{sum} \leq 1$.

These are the 3 conditions we have to match for a *System with Dynamic Real-Time Guarantees* (Definition 1). ■

However, the given definition of bounded tardiness seems to be far too restrictive for some practical cases. When many executions are affected by faults, hardened hardware should be used instead of recovery mechanisms. To successfully use recovery mechanisms, a low fault rate is usually expected. Without restricting ourselves to a specific fault model, we consider two general possibilities:

1) Faults happen as a burst in an interval with a small length and with a very low probability.
2) Faults happen with a given, low probability at each point in time.

In both cases $U^A_{sum} > 1$ is tolerable for short intervals if $U^N_{sum} < 1$ and if the intervals where no faults occur are significantly longer than the intervals where faults occur. Thus, in both scenarios, the setting itself will lead to a bounded tardiness for most practical cases. If we assume that a burst of faults only affects a small number of jobs compared to the number of jobs between two bursts of faults, the system should have a sufficient amount of time to return to *full timing guarantees* after each burst of faults. The length of the interval with *limited timing guarantees* for a given task set can be upper bounded if we suppose the maximum length Δ of a burst interval to be known, as shown in Section 5. If we assume the faults to happen with a given rate, this rate needs to be high to affect a sufficient number of tasks to lead to *limited timing guarantees* over a longer time interval.

Due to this consideration, we drop the condition that $U^A_{sum} \leq 1$ in the experiments regarding the acceptance rate in Section 7. We analyse the amount of time in which we can only give *limited timing guarantees* for task sets with a high utilization in the experiments in Section 7.2 to validate if our decision to drop the condition will have a high impact on the stability of the system for reasonable fault rates.

4 Properties of Priority Assignments

As we know how to test if a given priority ordering for a given task set will result in a *System with Dynamic Real-Time Guarantees*, we now explain how to construct such a priority ordering for a given task set, if one exists. We start by showing that existing or trivial priority orderings, namely Deadline Monotonic order and Criticality Monotonic order, are not optimal for *System with Dynamic Real-Time Guarantees*.

[2]The hyperperiod of a task set is the least common multiple of the periods of all tasks in the set.

Lemma 1 (Deadline Monotonic ordering is not optimal)**.** *For a System with Dynamic Real-Time Guarantees (Definition 1) Deadline Monotonic priority order for constrained-deadline task sets is not an optimal scheduling algorithm.*

Proof: This is proven by providing an example task set with two tasks that is schedulable as a *System with Dynamic Real-Time Guarantees* if the tasks are not in DM order but not schedulable if the tasks are in DM order. Such a task set can be found in the Appendix. ■

As mentioned in Example 1, in general, aborting the execution of $\tau_i \in \mathbf{T}^A_{\text{soft}}$ is only able to keep up hard real time guarantees for $\mathbf{T}^A_{\text{hard}}$, if all the tasks in $\mathbf{T}^A_{\text{soft}}$ have lower priorities than all the tasks in $\mathbf{T}^A_{\text{hard}}$.

Definition 3 (Criticality Monotonic)**.** *We say a task set* $\mathbf{T}$ *with two subsets* $\mathbf{T}^A_{hard}$ *and* $\mathbf{T}^A_{soft}$ *has a Criticality Monotonic ordering, when all tasks* $\tau_{\in}\mathbf{T}^A_{hard}$ *have higher priority than all tasks in* $\mathbf{T}^A_{soft}$*, i.e.,* $\Theta^A_{soft} = \emptyset$*.*

For the following lemma the internal order of $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ is not important. However, in general, we assume that $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ are internally ordered according to DM.

Lemma 2 (Criticality Monotonic Ordering is Not Optimal)**.** *For a System with Dynamic Real-Time Guarantees (Definition 1), Criticality Monotonic order is not an optimal scheduling algorithm.*

Proof: It is sufficient to provide a task set with two tasks that is schedulable as a *System with Dynamic Real-Time Guarantees* if the tasks are not in Criticality Monotonic order but not schedulable if the tasks are in Criticality Monotonic order. Such a task set can be found in the Appendix. ■

As we now know that neither DM nor Criticality Monotonic scheduling is optimal for *System with Dynamic Real-Time Guarantees*, we have to look at a more general approach. Audsley's Algorithm [2], also called optimal priority assignment (OPA), can be applied to find a feasible fixed priority assignment if the used schedulability test S is OPA compatible [12], i.e., the following three conditions all hold [12]:

1) The schedulability of a task τ_k, according to test S, 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 test S, 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 the higher priority cannot become unschedulable according to test S, if it was previously schedulable at the lower priority.

Lemma 3. *The Schedulability Test in Theorem 1 is OPA compatible.*

Proof: We will only sketch the proof here.

1) As TDA sums up the workload of all jobs from tasks in $hp(\tau_k)$, the order of those tasks has no impact on the result, as this only changes the order in which the workload is summed up. This holds true for both the normal and the abnormal case in Theorem 1. The task order has no impact on the condition $U^A_{sum} \leq 1$.
2) The workload of tasks in $lp(\tau_k)$ is not considered in the TDA at all, thus the order has no impact. The order of the tasks has no impact on the condition $U^A_{sum} \leq 1$.
3) If τ_k is assigned a higher priority due to a swap with τ_{k-1}, the workload of the higher priority tasks in the TDA will only be reduced and thus τ_k remains schedulable. The switch has no impact on the condition $U^A_{sum} \leq 1$ either.

Thus, we reach the conclusion. ■

In the next step, we will show that if a feasible priority assignment exists, the tasks in $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ can both be ordered according to DM.

Lemma 4 ($\mathbf{T}^A_{\text{hard}}$ in Deadline Monotonic Order)**.** *If for a given task set* $\mathbf{T}$ *a feasible priority assignment* P *for a System with Dynamic Real-Time Guarantees (Definition 1) exists, the tasks in* $\mathbf{T}^A_{hard}$ *can be reordered according to Deadline Monotonic order and Dynamic Real-Time Guarantees are still provided.*

Proof: We will only sketch the proof here as it is very similar to the prove for the optimality of DM [23]. We use the interchanging argument and look at the first two consecutive tasks τ_j and τ_k in the internal priority order of $\mathbf{T}^A_{\text{hard}}$ that are not in DM order, i.e., $D_j > D_k$ and $P(\tau_j) < P(\tau_k)$. If τ_j and τ_k are direct successors in P, we can directly swap them due to the optimality of DM for constrained-deadline task sets.

The case that τ_j and τ_k are not direct successors in P remains, i.e., $S := \left\{\tau_i \in \mathbf{T}^A_{\text{soft}} \mid P(\tau_j) < P(\tau_i) < P(\tau_k)\right\} \neq \emptyset$. We increase the priority of each $\tau_i \in S$ by 1 and set the priority of τ_j to $P(\tau_{k-1})$, thus τ_j and τ_k are direct successors in P again. All tasks in S remain schedulable as their priorities are increased, while τ_j remains schedulable because $D_j > D_k$ due to the precondition that τ_j and τ_k are not in DM order and the workload created by all tasks in $hp(\tau_j) \cup \tau_j \cup \tau_k$ up to D_k is smaller than D_k, as τ_k is schedulable. This means, we can swap τ_j and τ_k due to the optimality of DM and continue until all tasks in $\mathbf{T}^A_{\text{hard}}$ are in DM order. ■

Lemma 5 ($\mathbf{T}^A_{\text{soft}}$ in Deadline Monotonic Order)**.** *If a feasible priority assignment for a System with Dynamic Real-Time Guarantees (Definition 1) exists, the tasks in* $\mathbf{T}^A_{soft}$ *can be reordered according to Deadline Monotonic ordering while all Dynamic Real-Time Guarantees still hold.*

Proof: The proof is similar to the proof of Lemma 4. ■

We now know that we can reorder both subsets $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ to be in Deadline Monotonic order.

Theorem 2 ($\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ in DM order)**.** *If a feasible priority assignment for System with Dynamic Real-Time Guarantees (Definition 1) exists, a feasible priority assignment where the tasks in* $\mathbf{T}^A_{hard}$ *and* $\mathbf{T}^A_{soft}$ *are internally ordered according to the Deadline Monotonic order also exists.*

Proof: We know that $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ can be reordered according to DM individually while still obtaining feasibility. As reordering $\mathbf{T}^A_{\text{hard}}$ does not change the internal order of $\mathbf{T}^A_{\text{soft}}$ and vice versa, we can first reorder the tasks in $\mathbf{T}^A_{\text{hard}}$ to be in DM order and then reorder the tasks in $\mathbf{T}^A_{\text{soft}}$ to be in DM order, without changing the internal priority order of $\mathbf{T}^A_{\text{hard}}$. After reordering the two sets, we still obtain a feasible priority ordering for a *System with Dynamic Real-Time Guarantees*. ■

We can use Theorem 2 to find a feasible priority assignment for a *System with Dynamic Real-Time Guarantees*, if one

Algorithm 1 Feasible Priority Assignment

Input: $\mathbf{T}^A_{\text{hard}}$, $\mathbf{T}^A_{\text{soft}}$
Output: Feasible Order P of $\mathbf{T}^A_{\text{hard}} \cup \mathbf{T}^A_{\text{soft}}$ or **NOT POSSIBLE**

```
Sort T^A_hard by D_i increasingly
Sort T^A_soft by D_i increasingly
Find Assignment(T^A_hard, T^A_soft)

procedure FIND ASSIGNMENT(T^A_hard, T^A_soft)
    for (n = |T^A_hard ∪ T^A_soft|; n > 0; n := n − 1) do
        τ_t := last element of T^A_hard
        if (Try Priority(τ_t, {T^A_hard ∪ T^A_soft} \ {τ_t},n,hard)) then
            P(τ_t) := n
            T^A_hard := T^A_hard \ {τ_t}
        else
            τ_t := last element of T^A_soft
            if (Try Priority(τ_t, {T^A_hard ∪ T^A_soft} \ {τ_t},n,soft)) then
                P(τ_t) := n
                T^A_soft := T^A_soft \ {τ_t}
            else
                return NOT POSSIBLE
    return List of T^A_hard ∪ T^A_soft ordered by P(τ_t)

procedure TRY PRIORITY(τ_t, hp(τ_t), priority, task_type)
    P(τ_t) := n
    Assign hp(τ_t) to priorities 1, ..., n − 1
    if (task_type==hard) then
        C_i := C^A_i, ∀τ_i ∈ hp(τ_t) ∪ τ_t
    else
        C_i := C^N_i, ∀τ_i ∈ hp(τ_t) ∪ τ_t
    if (τ_t is schedulable according to TDA) then
        return true
    else
        return false
```

exists by using the priority assignment algorithm presented in pseudo-code in Algorithm 1. The idea is similar to OPA [2]: Find a task that can take the lowest priority under the assumption that all other tasks have higher priority. If such a task is found, assign it to the lowest priority (among the tasks), remove it from the task set, and redo the process with the remaining tasks. If for some priority no suitable task is found, we return *NOT POSSIBLE*, otherwise we return a feasible priority assignment. The main difference to OPA is that Algorithm 1 has to test at most two candidates for each priority: the remaining tasks in $\mathbf{T}^A_{\text{hard}}$ and in $\mathbf{T}^A_{\text{soft}}$ with the longest relative deadline, respectively. Hence, the tasks in $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ are ordered according to DM order in a preprocessing step. Note that regarding *Dynamic Real-Time Guarantees* it does not matter if the task with the longest deadline in $\mathbf{T}^A_{\text{hard}}$ or $\mathbf{T}^A_{\text{soft}}$ is assigned to the priority if TDA returns schedulable for both tasks.

Theorem 3 (Feasible Priority Assignment)**.** *If a feasible priority assignment for a given System with Dynamic Real-Time Guarantees exists, Algorithm 1 will find a feasible assignment.*

Proof: Theorem 2 proves that, if a feasible priority assignment exists, there also is an assignment where $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ are internally in Deadline Monotonic order. Assume such a priority assignment S to be given, e.g., an OPA where the tasks in $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ are reordered to be in DM ordering in the way presented in Lemma 4 and 5. We will reorder S until it has the same order as provided by Algorithm 1 to conclude the proof. An important observation is that the tasks in $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ will always have the same internal order in S and in the priority assignment provided by Algorithm 1, as S was reordered to have DM order in both subsets, and Algorithm 1 only tries to assign the task $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ with the longest relative deadline.

Let $\tau_j \in \mathbf{T}^A_{\text{hard}}$ be the task in $\mathbf{T}^A_{\text{hard}}$ with the lowest priority in S, i.e., the task in $\mathbf{T}^A_{\text{hard}}$ with the longest period. We use the interchanging argument again and try to exchange τ_j with tasks that have lower priority in S until τ_j would not be schedulable at its new priority. This means that S remains schedulable, as for all other tasks the priority is only increased. Let T^j_{low} denote all tasks that have lower priority than τ_j after the priority of τ_j was decreased. We have two cases:

1) $T^j_{low} = \emptyset$: the new position of τ_j is the lowest.
2) There are $T^j_{low} \subseteq \mathbf{T}^A_{\text{soft}}$ that have lower priority than τ_j. All $\tau_i \in T^j_{low}$ are schedulable, as P was schedulable and the priority of those tasks was not changed.

Now τ_j and $\tau_i \in T^j_{low}$ are in the same order as Algorithm 1 provides, as Algorithm 1 only schedules a tasks in $\mathbf{T}^A_{\text{soft}}$ to a priority if the task in $\mathbf{T}^A_{\text{hard}}$ cannot be assigned. In the next step we take the task $\tau_k \in \mathbf{T}^A_{\text{hard}}$ that has the lowest priority in $\mathbf{T}^A_{\text{hard}} \setminus \{\tau_j\}$ and decrease its priority until it would not be schedulable anymore or we would exchange it with a tasks in $\mathbf{T}^A_{\text{hard}}$. Now τ_k and all tasks with a priority lower then τ_k are in the same order as Algorithm 1 provides with the same argument. We repeat this until all tasks in $\mathbf{T}^A_{\text{hard}}$ are in the same order as provided by Algorithm 1. ■

One could apply Audsley's Algorithm [2] (OPA) directly to find a feasible priority assignment. However, in general, Algorithm 1 has a much better runtime than OPA. Assume the task set contains n tasks. In the worst case TDA has to test a pseudo-polynomial number of time points to determine if a task is schedulable on a given level. Each of these tests has a runtime of $O(n)$ to sum up the workload of the higher priority tasks. Due to this, we will denote the time needed to test the schedulability of a task at a priority level with $O(n \cdot TDA(n))$ where $TDA(n)$ is the number of tests for that level.

In the Worst Case, we have to test $O(n)$ tasks with TDA on each priority level if we use OPA. The time complexity of OPA is $O(n^2 \cdot n \cdot TDA(n)) = O(n^3 \cdot TDA(n))$, as there are n priority levels. When Algorithm 1 is used, for each priority level only at most two tasks have to be considered. This leads to a time complexity of $O(2n \cdot n \cdot TDA(n)) = O(n^2 \cdot TDA(n))$ to find the assignment. Ordering $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ according to the relative deadlines can be done in $O(n \log n)$ which is dominated by $O(n^2 \cdot TDA(n))$. Thus, the time complexity of Algorithm 1 is a power less than the one of OPA when assigning priorities to give *Real-Time Service Level Guarantees*.

5 System Mode Analysis

In this section, we analyse the system mode, assuming that faults occur as a burst during an interval and that this interval length Δ is known. We will calculate the maximum time the system will provide only *limited timing guarantees* under the assumption that no more faults will occur. Let θ_b be the latest time instant at which a fault is detected. Our objective is to find the time when the system will give *full timing guarantees* again, i.e., all jobs of all tasks will meet their deadlines.

Let θ_0 denote the latest time-instant $\leq \theta_b$ at which the processor is idle, and θ_a be the time instant at which a fault is first detected over $[\theta_0, \theta_b]$. Let $\theta_b - \theta_a = \Delta$, as illustrated in Figure 2. To provide *full timing guarantees* again after a fault interval, it is sufficient that the system is idle at a point $\theta_f \geq \theta_b$ as no fault that happened before θ_f can affect any

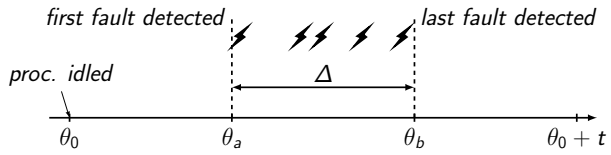

Fig. 2: The illustration of an busy interval of length t undergoing an interval of burst fault $[\theta_a, \theta_b]$ equal to length Δ.

task that is realised at a time $t \geq \theta_f$. If we can show that the interval length during which a system is busily executing for a given Δ is no more than some value, we know that after this interval the system will provide *full timing guarantees* again.

We denote $\Omega(t)$ as the maximum total amount of execution time from the tasks in $\mathbf{T}$ in $[\theta_0, \theta_0 + t)$. Clearly, the interval length during which a system is busily executing is no more than the smallest value of t satisfying $\Omega(t) \leq t$.

Theorem 4 (Computing $\Omega(t)$)**.** *Let Γ donate the interval over $[\theta_0, \theta_0 + t)$ except $[\theta_a, \theta_b]$. An upper bound on $\Omega(t)$ can be calculated as:*

$$\Omega(t) = \Delta + F + I(t) \tag{2}$$

where the terms are as described below.

- *$I(t)$: the work from all tasks executed* normally *over the interval $[\theta_0, \theta_0 + t)$.*
- *F: the work of all tasks performing recovery actions over the interval Γ.*
- *Δ: the interval length during which faults are detected.*

To prove Theorem 4, we first derive equations for each of the aforementioned terms and show that we account for the maximum possible amount of workload.

Lemma 6. *In the interval $[\theta_0, \theta_0 + t)$, there are at most $\left\lceil \frac{t}{T_i} \right\rceil$ jobs of task τ_i released over $[\theta_0, \theta_0 + t)$.*

Proof: As the processor is idle at θ_0 by definition, it is clear that there are at most $\left\lceil \frac{t}{T_i} \right\rceil$ jobs of task τ_i, released over $[\theta_0, \theta_0 + t)$, which concludes this lemma. ∎

By Lemma 6, it follows that $I(t) = \sum_{\tau_i \in \tau} W_i(t)$ where $W_i(t) = \left\lceil \frac{t}{T_i} \right\rceil C_i^N$. We know that the amount of recovery time of a task instance is at most $C_i^A - C_i^N$.

Lemma 7. *Outside the interval Δ, there is at most one job of each task performing recovery actions where the workload for this recovery actions is upper bounded by $C_i^A - C_i^N$. Therefore,*

$$F = \sum_{\tau_i \in \tau} \left(C_i^A - C_i^N \right) \tag{3}$$

Proof: We prove this lemma by considering two cases:

- Recovery in $[\theta_0, \theta_a]$. By definition of θ_a as the first time a fault is detected after the last idle time, no recovery action is performed over $[\theta_0, \theta_a]$.
- Recovery after θ_b. As each task may have at most one job executed at any time instant, we know that each task has at most one job that has been partially executed and is not yet completed. As, by the definition of θ_b, no further faults occur after θ_b, there is at most one recovery part possibly carried out after θ_b, upper bounded by $C_i^A - C_i^N \ \forall \tau_i$.

Hence, we can conclude this lemma. ∎

Lastly we need a simple observation about Δ.

Observation 1. *The abnormal workload executed in the time interval $[\theta_a, \theta_b]$ is no more than Δ, regardless of whichever job is executing.*

Now we can prove Theorem 4.

Proof (Theorem 4): We have to show that we accounted for all possible workload in the time interval $[\theta_0, \theta_0 + t)$. We account for all normal executions in $I(t)$. This includes normal executions that are done in Δ. For abnormal executions, we also account for the part up to C_i^N in $I(t)$, regardless if they are executed in Δ or outside as shown in Lemma 6. The recovery part can only be executed once outside Δ as shown in Lemma 7. Thus, we get the most amount of additional workload in Δ if only recovery operations are executed in Δ. This amount of work is bounded by Δ due to Observation 1. ∎

Please note that it is not possible that only recovery operations will be performed in Δ, as the faults happen in this interval as well, and in our model faults will only prolong the execution time if they happen before C_i^N. However, tasks may only be executed for a very small part of C_i^N during Δ and thus we use Δ as an upper bound.

6 System Monitor Design

Up to this point, all essential analysis in terms of system scheduling is provided. However, as faulty-aware system design is desirable in the industrial practice, having an online monitor to reflect the system status is also important. The monitor should trigger warnings if the system can only give *limited timing guarantees* for an individual task or the whole system, and display the next time the task/system will return to *full timing guarantees*, i.e., the monitored task or all tasks in $\mathbf{T}^A_{\text{soft}}$ will meet their hard deadline. We propose to use approximation to detect the change from *full timing guarantees* to *limited timing guarantees*, and for the calculation of an upper bound of the next time instance the system will return to *full timing guarantees*.

As we guarantee the timing behaviour of the *timing strict* tasks offline, we only have to monitor *timing tolerable* tasks. Assume we want to monitor $\tau_k \in \mathbf{T}^A_{\text{soft}}$. For notational brevity, we define $hp(\tau_k)^H := hp(\tau_k) \cap \mathbf{T}^A_{\text{hard}}$ and $hp(\tau_k)^S := hp(\tau_k) \cap \mathbf{T}^A_{\text{soft}}$, i.e., the subset of the tasks with higher priority than τ_k in $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$, respectively. Assume for each task is known if a job of this task is ready to be executed at the time we analyse, and that the current execution mode of the task is known as well.

We know that the interference from tasks in $hp(\tau_k)$ and/or self-pushing (e.g., Figure 1) can prolong the execution of a job of τ_k. Due to this interference, multiple deadline misses may happen once the system switched to *limited timing guarantees*. The next time we can guarantee that τ_k meets its deadline, if no further faults occur, is the moment a lower priority task is executed, i.e., the end of the level k busy period, denoted with $busy_k$. Let $|busy_k|$ be the length of $busy_k$. If $|busy_k|$ is smaller than the time the current job of τ_k has left to finish its execution, we can give *full timing guarantees* for τ_k; otherwise τ_k may miss its deadline, we can only give *limited timing guarantees*, and *full timing guarantees* for τ_k can be provided again at the end of $busy_k$.

Similar to Section 5, we will denote the maximum total amount of execution time from $hep(\tau_k)$ in an interval from $[\theta_0, \theta_0 + t)$ as $\Omega_k(t)$ and look for the smallest value of t where $\Omega_k(t) \leq t$. To apply the the formula from Section 5 directly, we would have to keep track of the last time the processor executed a tasks in $lp(\tau_k)$ for each τ_k, and the amount of additional interference due to higher priority tasks during this interval. We use an alternative approach here by setting θ_0 to the current time, then calculate the carry in and the future workload due to tasks in $hep(\tau_k)$. The reason is not to have additional, potentially high, overhead for keeping track of the interference of tasks in $hep(\tau_k)$ in $busy_k$ for each $\tau_k \in \mathbf{T}^A_{\text{soft}}$.

Let $I_k(t)$ denote the maximum workload due to jobs of tasks in $hep(\tau_k)$ with normal executions in $[\theta_0, \theta_0 + t]$, i.e.,

$$I_k(t) = \sum_{\tau_i \in hep(\tau_k)} W_i(t) = \sum_{\tau_i \in hep(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i^N \quad (4)$$

The carry in workload from task in $hep(\tau_k)$ must be taken into account as well. We denote the carry in workload of τ_i by $G(\tau_i)$. To determine $G(\tau_i)$, we need to know how much workload remains for a job of τ_k that is currently executable. This can be estimated by keeping track of the time the job already ran. Depending on the mode of the job we subtract that value from C_i^N or C_i^A to get $G(\tau_i)$; thus we also need to know if a job is currently in normal or abnormal mode. If keeping track of the time a task has been executed would be too much overhead, depending on τ_is mode C_i^N or C_i^A can directly be used as an upper bound on $G(\tau_i)$. At most one job of each task in $hp(\tau_k)^H$ can be in the system, as $hp(\tau_k)^H$ will always meet their deadline by system design. For tasks in $hp(\tau_k)^S \cup \tau_k$ there can be carry in from more than one postponed execution, thus we may have to sum this up with the remaining workload of a currently active job to get $G(\tau_i)$ for $\tau_i \in \mathbf{T}^A_{\text{soft}}$. The total carry in can be calculated as $G_k = \sum_{\tau_i \in hep(\tau_k)} G(\tau_i)$.

We have to look for the smallest t with:

$$\Omega_k(t) = \sum_{hep(\tau_k)} \left\lceil \frac{t}{T_i} \right\rceil C_i^N + \sum_{\tau_i \in hep(\tau_k)} G(\tau_i) \leq t \quad (5)$$

As we do not know how many jobs of τ_k will be executed before $busy_k$ ends, we create a virtual task $\tau_{k'}$ with $C_{k'} = G_k$ and with the virtual priority $k + 1$. We now calculate an upper bound on the WCRT of $\tau_{k'}$ using Theorem 1 in [6] by Bini et al. This states that for each sporadic task τ_i in a fixed priority system the WCRT R_i will be upper bounded by $R_i \leq \frac{C_i + \sum_{j=1}^{i-1} C_j(1-U_j)}{1 - \sum_{j=1}^{i-1} U_j}$. In our case, we have to take the future jobs of all tasks in $hep(\tau_k)$ into account and have the carry in G_k as the execution time of our virtual task. This means we can calculate an upper bound for the length of the level k busy period $busy_k$ as $|busy_k|^*$ and get:

$$|busy_k| \leq \frac{G_k + \sum_{j \in hep(\tau_k)} C_j^N (1 - U_j^N)}{1 - \sum_{j \in hep(\tau_k)} U_j^N} = |busy_k|^* \quad (6)$$

Only the tasks that already started may have abnormal execution behaviour, thus we assume normal execution for all future tasks and use C_j^N and U_j^N in the formula. The related workload is explicitly summed up in G_k. We can provide *full timing guarantees* for τ_k at θ_0 if $|busy_k|^* \leq D_k$. If $|busy_k|^* > D_k$ we only provide *limited timing guarantees*, and $|busy_k|^*$ is an upper bound on the time the systems needs to go back to to *full timing guarantees* for τ_k, assuming no further faults occur.

Note that we could also tighten the analysis, e.g., using the tighter bounds provided in [7], [10] that require to sort the tasks in $hep(\tau_k)$ by their periods.

The remaining question is when to check if *timing guarantees* have changed. There are two general approaches. One is to check periodically, where the period of this check can be determined depending on the needed granularity during system design. The other is to check in an event-driven manner. The moment a fault is detected is a natural choice for an event, as this is the only point in time a change from *full timing guarantees* to *limited timing guarantees* may happen for the affected tasks. This is an important observation, as no checks are needed while all tasks are with *full timing guarantees* if no faults occur. A change from *limited timing guarantees* back to *full timing guarantees* for a task τ_k may happen when a task with higher priority than τ_k or an instance of τ_k itself finishes.

7 Evaluations

We focused on two questions. First we determined the possible acceptance rates for *Systems with Dynamic Real-Time Guarantees* under different scenarios in a schedulability analysis in Section 7.1. In addition, we provided a system state analysis where we explored the behaviour of tasks sets with high utilization under different fault rates in Section 7.2, i.e., we analysed the percentage of time where *full timing guarantees* are provided for these task sets under different fault rates.

7.1 Schedulability Analysis

We generated random implicit-deadline task sets with a given U^N_{sum} according to the UUniFast method [5], using the approach suggested by Emberson et al. [14] to generate the task periods according to a log-uniform distribution with two orders of magnitude, i.e., $[1ms - 100ms]$. Specifically, $\log_{10} T_i$ is a uniform distribution in the defined range. The WCET in the normal mode was set according to the utilization, i.e., $C_i^N = U_i \cdot T_i$. We analysed task sets of 5 different cardinalities, i.e., 5, 10, 20, 50 and 100 tasks, where we randomly picked 30%, 40%, 50%, 60% and 70% of these tasks to be in $\mathbf{T}^A_{\text{hard}}$. We calculated C_i^A for $\tau_i \in \mathbf{T}^A_{\text{hard}}$ according to 3 different ratios, called WCET-Factors, to simulate the 3 scenarios presented in Section 2.1, where we assumed that for re-execution the fault detection takes 20% of the WCET without fault detection (only one detection at the end) and for the checkpointing case we assumed 40% overhead:

- Re-Execution: $C_i^A \approx 1.83 \cdot C_i^N$ as $\frac{2.2}{1.2} \approx 1.83$
- Two Re-Executions: $C_i^A \approx 2.83 \cdot C_i^N$ as $\frac{3.4}{1.2} \approx 2.83$
- Checkpointing: $C_i^A \approx 1.14 \cdot C_i^N$ as $\frac{1.6}{1.4} \approx 1.14$

We used two different values for the relation of C_i^A to C_i^N for $\tau_i \in \mathbf{T}^A_{\text{soft}}$: The same value as used for $\mathbf{T}^A_{\text{hard}}$ or 1.0, i.e., just fault detection without any kind of fault recovery.

For each of these in total $5 \cdot 5 \cdot 3 \cdot 2 = 150$ settings, we analysed 1000 randomly generated task sets for each utilization value $U^N_{sum} \in [1\%, 100\%]$ (step size 1%). We tested the schedulability of the task sets as a *System with Dynamic Real-Time Guarantees* using 4 fixed priority orderings: Rate Monotonic (RM), Criticality Monotonic (CM), Optimal Priority Assignment (OPA) [2], and the ordering provided by Algorithm 1, all tested by the schedulability test in Theorem 1. In these tests, we dropped the condition $U^A_{sum} \leq 1$ according

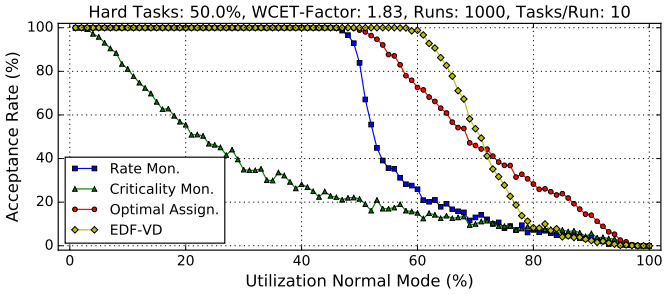

Fig. 3: Acceptance rate for 10 tasks/run.

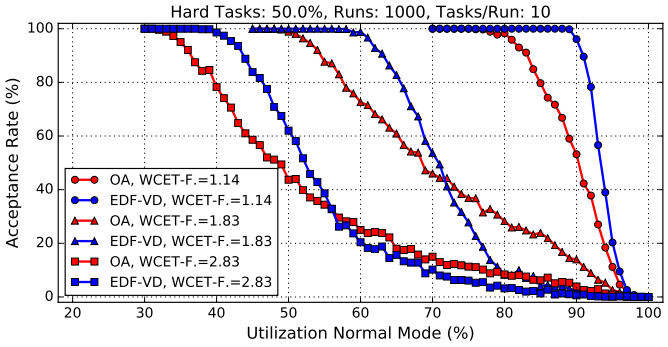

Fig. 4: Comparison of Optimal Assignment and EDF-VD. For a WCET-Factor of 1.14 EDF-VD is always superior to Optimal Assignment while for WCET-Factors of 1.83 and 2.83 Optimal Assignment is better than EDF-VD for Utilizations higher then 71% and 56% respectively.

to the arguments at the end of Section 3. We analysed the behaviour of some task sets with high utilization under different fault rates in the Section 7.2 to support this argument. In addition to the acceptance test, we monitored if RM, CM or OPA were able to schedule a task set that was not schedulable by Algorithm 1. That case never occurred and Algorithm 1 and OPA always provided an identical acceptance rate which strongly supports our claim that Algorithm 1 provides an optimal assignment (Theorem 3). Due to this, only one curve, labeled *Optimal Assignment* (OA), is used to represent OPA and Algorithm 1 in Figure 3, showing the results for sets with 10 tasks, 5 of them in $\mathbf{T}^A_{\text{hard}}$, and a WCET-Factor of ≈ 1.83 for both $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$. We also tested if the task sets are schedulable with EDF-VD [3] according to the schedulability test provided by Baruah et. al in [3, Section 3]. EDF-VD is the dynamic state-of-the-art scheduling algorithm for Mixed-Criticality Systems. Contrary to a *System with Dynamic Real-Time Guarantees*, EDF-VD does not provide any guarantees for low-criticality tasks in high-criticality mode.

The most interesting part of Figure 3 is the comparison of EDF-VD [3] and our optimal assignment (OA). The curve for OA starts dropping earlier than that for EDF-VD (Utilization 52% and 61% respectively) but EDF-VD drops faster. From 72% onwards, OA can schedule more task sets than EDF-VD. In this area $\sum_{\tau_i \in \mathbf{T}^A_{\text{hard}}} U^A_i$ can be too large to find values for the virtual deadlines in EDF-VD as those virtual deadlines are generated from the original deadlines by multiplying with a factor ≤ 1. As OA performs an exact test with the actual deadlines it is still able to find a feasible schedule. Similar behaviour could be observed in most of the settings.

In Figure 4 we compared the optimal assignment (OA) and EDF-VD for the three different WCET-Factors. For a WCET-Factor of 1.14 EDF-VD always outperforms OA. It can be seen that EDF-VD only performs better than OA up to a

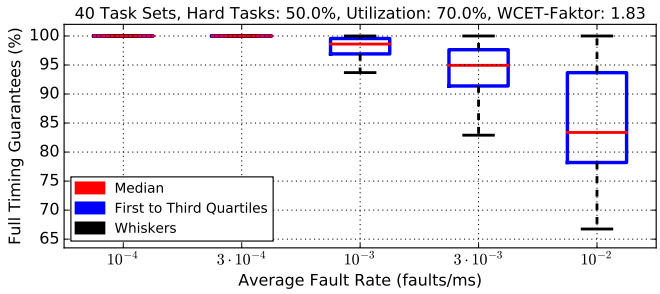

Fig. 5: Percentage of Time with *Full Timing Guarantees* for task sets with critical utilization under different fault rates.

utilization of 71% and 56% for WCET-Factors of 1.83 and 2.83, respectively. For higher utilization values OA is able to schedule more task sets than EDF-VD. In addition, the gap between EDF-VD and OA in the acceptance rate for a given utilization seems reasonable if we consider that OA does not drop any tasks when only *limited timing guarantees* are provided and that the scheduling overhead of EDF is in general larger than the overhead of fixed-priority scheduling.

Figures displaying the acceptance rates depending on the percentage of *timing strict* tasks and the impact of the set size are presented in the Appendix.

7.2 System State Analysis

Figure 3 shows an acceptance rate of 44.4% for task sets with 10 tasks, 50% tasks in $\mathbf{T}^A_{\text{hard}}$, WCET-Factor of ≈ 1.83, and $U^N_{sum} = 70\%$, which means $U^A_{sum} \approx 128.1\%$. EDF-VD [3] was able to guarantee schedulability for 53.7% of these task sets. As both algorithms can only provide schedulability for roughly 50% of the task sets and the acceptance rate is decreasing fast around 70% utilization we assume those sets have a *critical* utilization. We analysed the system state, regarding the percentage of time, where we can give *full timing guarantees* and *limited timing guarantees* for 40 of these *critical* sets that are schedulable according to Algorithm 1.

We used QEMU emulators under *Real-Time Executive for Multiprocessor Systems (RTEMS)* [1] version 4.11 where the used kernel is enhanced by the pending patch #2772 [11], enabling only one processor. The chosen board support package was *RealView Platform Baseboard Explore for Cortex-A9*. For each testing instance, the system ran for one hour under different fault rates, i.e., on average 10^{-4}, $3 \cdot 10^{-4}$, 10^{-3}, $3 \cdot 10^{-3}$ and 10^{-2} faults per millisecond (f/ms). If the executed instance of τ_i is faulty, the corresponding WCET C^N_i becomes C^A_i. If the system can give *full timing guarantees* or only *limited timing guarantees* is decided by the system monitor presented in Section 6. The results, i.e., the percentage of time the system was running with *full timing guarantees*, are shown in Figure 5. The median of those 40 sets is colored red. The blue box represents the interval from the first to the third quartile, while the black whiskers show the minimum and maximum of all of the data.

At a fault rate of 10^{-4} and $3 \cdot 10^{-4}$ f/ms the system always provides *full timing guarantees*. When the fault rate is increased to 10^{-3} and $3 \cdot 10^{-3}$ the median value decreases to 98.6% and 94.9%, respectivaly, while the third quartile is at 99.6% and 97.6%, respectivaly, and the first quartile is at 96.91% and 91.4%, respectivaly. If the fault rate is increased further to 10^{-2} we got 93.7%, 83.4%, and 78.2% for third quartile, median, and first quartile, respectively. This shows

that even for higher fault rates under an, in general, difficult setting we are still able to provide *full timing guarantees* for a reasonable percentage of time. However, for some of the task sets the percentage of time where *full timing guarantees* can be given drops faster as can be seen by the comparatively long lower whiskers for $3 \cdot 10^{-3}$ and 10^{-2}.

8 Conclusion and Extensions

We provide the definition of a *System with Dynamic Real-Time Guarantees* to model real-time task sets in an uncertain or faulty execution environment. We present a schedulability test, a way to find an optimal assignment of fixed-priorities for such systems and show how to monitor the system state. The evaluations provide good supports to our claim that uncertain and faulty execution environments can be reasonably handled without any online adaptation if certain properties can be provided offline. We showed that, if the fault rate and task settings are given, the percentage of time where the system only provides *limited timing guarantees* can be approximated. For a concrete system, this can be used to decide whether additional online adaptation might be needed.

As we only consider implicit- and constrained-deadline task sets under preemptive scheduling on a uniprocessor, we plan to extend our model to cover arbitrary deadlines, non-preemptive scheduling, and multiprocessor platforms.

Acknowledgement: This paper has been supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/), and the priority program "Dependable Embedded Systems" (SPP 1500 - http://spp1500.itec.kit.edu).

References

[1] Rtems: Real-time executive for multiprocessor systems. http://www.rtems.com/, 2013.
[2] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991.
[3] S. K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *J. ACM*, 62(2):14, 2015.
[4] R. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 2005.
[5] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
[6] E. Bini, T. H. C. Nguyen, P. Richard, and S. K. Baruah. A response-time bound in fixed-priority scheduling with arbitrary deadlines. *IEEE Trans. Computers*, 58(2):279–286, 2009.
[7] E. Bini, A. Parri, and G. Dossena. A quadratic-time response time upper bound with a tightness property. In *RTSS*, pages 13–22, 2015.
[8] U. Bordoloi, B. Tanasa, M. Tahoori, P. Eles, Z. Peng, S. Shazli, and S. Chakraborty. Reliability-aware instruction set customization for asips with hardened logic. In *RTCSA 2012*.
[9] A. Burns and R. Davis. Mixed criticality systems-a review. Technical report, University of York, 2016. 7th edition.
[10] J.-J. Chen, W.-H. Huang, and C. Liu. k2Q: A quadratic-form response time and schedulability analysis framework for utilization-based analysis. In *Real-Time Systems Symposium (RTSS)*, 2016.
[11] K.-H. Chen. #2772 ticket: Enhancement for more general real-time model. http://devel.rtems.org/ticket/2772, 2016.
[12] R. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *IEEE Real-Time Systems Symposium*, pages 398–409, 2009.
[13] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.
[14] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.
[15] R. Ernst and M. D. Natale. Mixed criticality systems - A history of misconceptions? *IEEE Design & Test*, 33(5):65–74, 2016.
[16] A. Esper, G. Nelissen, V. Nélis, and E. Tovar. How realistic is the mixed-criticality real-time system model? In *RTNS*, pages 139–148, 2015.
[17] R. Garg, N. Jayakumar, S. Khatri, and G. Choi. A design approach for radiation-hard digital electronics. In *Design Automation Conference, 2006*.
[18] O. Gettings, S. Quinton, and R. I. Davis. Mixed criticality systems with weakly-hard constraints. In *RTNS*, pages 237–246, 2015.
[19] Z. A. H. Hammadeh, S. Quinton, and R. Ernst. Extending typical worst-case analysis using response-time dependencies to bound deadline misses. In *EMSOFT*, pages 10:1–10:10, 2014.
[20] P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *The 51st Annual Design Automation Conference, DAC*, pages 131:1–131:6, 2014.
[21] P. Kumar and L. Thiele. Quantifying the effect of rare timing events with settling-time and overshoot. In *RTSS*, pages 149–160, 2012.
[22] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium'89*, pages 166–171, 1989.
[23] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2:237–250, 1982.
[24] F. Many and D. Doose. Scheduling analysis under fault bursts. In *RTAS*, pages 113–122, 2011.
[25] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *ACM MICRO*, 2003.
[26] S. Quinton, M. Hanke, and R. Ernst. Formal analysis of sporadic overload in real-time systems. In *DATE*, pages 515–520, 2012.
[27] S. Quinton, M. Negrean, and R. Ernst. Formal analysis of sporadic bursts in real-time systems. In *DATE*, pages 767–772, 2013.
[28] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News*.
[29] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, 2002.
[30] A. Thekkilakattil, R. Dobrin, and S. Punnekkat. Fault tolerant scheduling of mixed criticality real-time tasks under error bursts. In *The International Conference on Information and Communication Technologies*. Elsevier Procedia Computer Science, December 2014.
[31] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, pages 239–243, 2007.
[32] W. Xu, Z. A. H. Hammadeh, A. Kröller, R. Ernst, and S. Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *ECRTS*, pages 247–256, 2015.

Appendix

Example: abnormal execution does not necessarily lead to a deadline miss (Figure 6): The tasks are characterized by C_i^N and $D_i = T_i$ with $C_i^A = 2 \cdot C_i^N$. Three faults occur, represented by the ↯, and lead to a prolonged execution time for the first two jobs of τ_1 and the first jobs of τ_2. However, $\tau_3 \in \mathbf{T}^A_{\text{soft}}$ and $\tau_4 \in \mathbf{T}^A_{\text{hard}}$ still meet their deadlines and thus aborting τ_2 and τ_3 to ensure the timingness of τ_4 would be unnecessary.

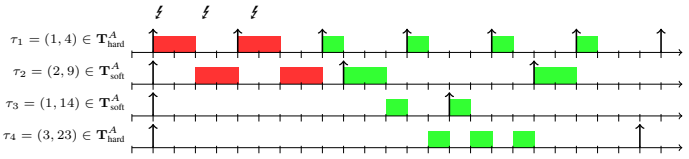

Fig. 6: Faults will not necessarily lead to deadline misses.
Tasks characterization: (C_i^N, T_i), with $C_i^A = 2 \cdot C_i^N$ and $D_i = T_i$.
Three occurring faults (↯) lead to prolonged execution for the red colored jobs of τ_1 and τ_2 due to error recovery. However, τ_4 still meets its deadline.

Proof of Theorem 1: Exact Schedulability Test for Constrained Deadlines. As TDA is an exact schedulability test for any preemptive fixed priority scheduling schema if the critical instant theorem holds, 1) follows immediately. If no faults occur, all tasks are executed in normal mode and the situation is identical to the case that all tasks have only one possible execution time

In 2), we only have to test the tasks in $\mathbf{T}^A_{\text{hard}}$ with TDA, as we only want bounded tardiness for $\tau_i \in \mathbf{T}^A_{\text{soft}}$; this will be tested in the next step. However, tasks in Θ^A_{soft} can contribute

workload to the Worst Case Response Time of tasks in $\mathbf{T}^A_{\text{hard}}$. The tasks in $\mathbf{T}^A_{\text{soft}}$ do not have hard real-time constraints anymore, but they are executed with the same priority as in the normal mode and thus can be handled as hard real-time tasks in the analysis, as they contribute the same workload as tasks in $\mathbf{T}^A_{\text{hard}}$ would. The possibility that these tasks may miss their deadlines does not have impact on the analysis, as TDA tests every task individually and we only care about the workload those tasks contribute if they are executed, and not about the concrete execution order of higher priority tasks or if the tasks meat or miss their deadline. The worst case for $\tau_k \in \mathbf{T}^A_{\text{hard}}$ will happen when it is released together with all higher priority tasks, all subsequent jobs of these tasks will be released as early as possible, and all tasks are executed in abnormal mode. Using C^N_i instead of C^A_i in the analysis would only decrease the workload generated by tasks in $hp(\tau_k)$ or the task itself.

For the tasks in $\mathbf{T}^A_{\text{soft}}$, only bounded tardiness has to be provided. If $U^A_{sum} \leq 1$, for the critical instant of each task the total workload in one hyperperiod[3] is smaller than or equal to the hyperperiod length. Thus the latest possible time for a job to finish is one hyperperiod after its release if $U^A_{sum} \leq 1$. This leads to an upper bound on the Worst Case Response Time for all tasks in $\mathbf{T}$ and thus to a bounded tardiness.

If $U^A_{sum} > 1$, the maximum amount of workload in one hyperperiod, due to jobs that arrive in the hyperperiod but are not fully executed in the hyperperiod, is larger than the length of the hyperperiod. Let this extra amount of workload be $\gamma > 0$. As for each value β a number of hyperperiods m with $\gamma \cdot m > \beta$ exists, the tardiness is not bounded.

This are the 3 conditions we have to match for a *System with Dynamic Real-Time Guarantees* (Definition 1). □

Proof of Lemma 1. Assume $\varepsilon > 0$ but very small and two tasks $\tau_1 \in \mathbf{T}^A_{\text{soft}}$ with $C^N_1 = 1,\ C^A_1 = 1+\varepsilon,\ T_1 = D_1 = 4$ and $\tau_2 \in \mathbf{T}^A_{\text{hard}}$ with $C^N_2 = 3,\ C^A_2 = 4,\ T_1 = D_1 = 6$ that are scheduled according to DM, i.e., $P(\tau_1) < P(\tau_2)$. In normal mode both tasks will meet their deadlines with WCRTs of $R^N_1 = 1$ and $R^N_2 = 4$. In the abnormal mode, the total utilization $U^A_{sum} = \frac{1+\varepsilon}{4} + \frac{4}{6} = \frac{22+6\cdot\varepsilon}{24} < 1$ for small values of $\varepsilon > 0$, which leads to bounded tardiness for τ_1 in the abnormal mode. For the abnormal mode, $R^A_2 = 2\cdot(1+\varepsilon)+4 > 6$ and thus τ_2 will miss its deadline.

If the priorities of τ_1 and τ_2 are switched, U^A_{sum} remains the same and thus τ_1 has bounded tardiness in abnormal mode. Both tasks are schedulable in normal mode as $R^N_1 = 4$ and $R^N_2 = 3$. In abnormal mode the WCRT of τ_2 is $4 < 6$. □

Proof of Lemma 2. Assume $\varepsilon > 0$ but very small and two tasks $\tau_1 \in \mathbf{T}^A_{\text{soft}}$ with $C^N_1 = 1,\ C^A_1 = 1+\varepsilon,\ T_1 = D_1 = 3$ and $\tau_2 \in \mathbf{T}^A_{\text{hard}}$ with $C^N_2 = 3,\ C^A_2 = 3+\varepsilon,\ T_1 = D_1 = 6$. Assume these two tasks to be scheduled according to Criticality Monotonic, i.e., $P(\tau_1) > P(\tau_2)$. In ormal mode we get $R^N_1 = 4$ and $R^N_2 = 3$ and thus τ_1 will not meet its deadline.

If the priorities of τ_1 and τ_2 are switched, τ_1 and τ_2 will both meet their deadlines in normal mode, i.e., $R^N_1 = 1$ and $R^N_2 = 4$. In abnormal mode τ_2 will still meet its deadline, as $R^N_2 = 2\cdot(1+\varepsilon)+3+\varepsilon = 5+3\cdot\varepsilon < 6$. Here τ_1 has bounded tardiness, as $U^A_{sum} = \frac{1+\varepsilon}{3} + \frac{3+\varepsilon}{6} = \frac{5+3\cdot\varepsilon}{6} < 1$. □

[3]The hyperperiod of a task set is the least common multiple of the periods of all tasks in the set.

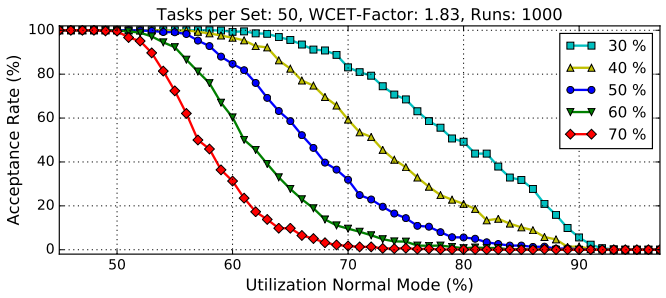

Fig. 7: Acceptance Rate for Percentages of *timing strict* tasks. The acceptance rate drops earlier if the percentage of hard tasks is higher.

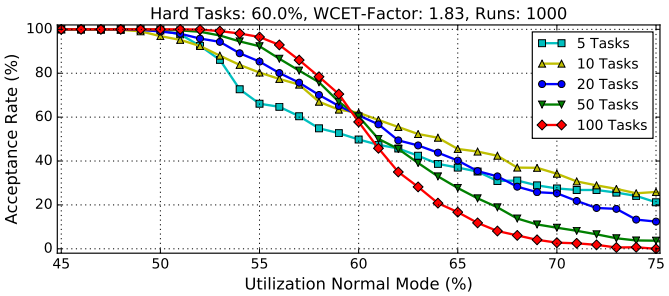

Fig. 8: Acceptance Rate for Different Set Sizes.
Larger task sets starts decreasing later but will decrease faster. Only the task sets with 5 tasks behave a bit differently due to the randomness effects.

Algorithm 1: Feasible Priority Assignment: $\mathbf{T}^A_{\text{hard}}$ and $\mathbf{T}^A_{\text{soft}}$ are ordered according to Deadline Monotic order in a preprocessing step and the ordered task sets are the input for *Find Assignment*, which tries to assign the $\tau_t \in \mathbf{T}^A_{\text{hard}}$ with the largest relative deadline to the lowest priority by assuming that all other tasks have higher priority. If τ_t can be assigned, it is given the lowest priority and removed from $\mathbf{T}^A_{\text{hard}}$. If not, $\tau_t \in \mathbf{T}^A_{\text{soft}}$ with the largest relative deadline is tested. If it can be assigned, it is given lowest priority and removed from $\mathbf{T}^A_{\text{soft}}$. If not, *NOT POSSIBLE* is returned. This procedure is continued until either all priorities are assigned or for one priority no task can be assigned. To keep the pseudo code short we do not take care of the case that either $\mathbf{T}^A_{\text{hard}}$ or $\mathbf{T}^A_{\text{soft}}$ will be empty at some point during the algorithm. In that case only the lowest priority task of the not empty set will be tested.

Evaluations

We analysed the schedulability under the Optimal Assignment (OA) with relation to the percentage of timing strict tasks and the size of the task set.

Schedulability under OA related to the percentage of timing strict tasks (Figure 7). When we consider different rates for the percentage of *timing strict* tasks and only display the interesting utilization interval $[45\%, 95\%]$. The acceptance rate drops earlier when the percentage of *timing strict* tasks is higher. This is due to the fact, that we have to give hard real-time guarantees for a higher percentage of tasks.

Schedulability under OA with relation to different set sizes (Figure 8). We analysed the impact of different set sizes and only the interesting utilization interval $[45\%, 75\%]$ is displayed. If only the sets with 10, 20, 50 and 100 tasks are consider, the curve for the larger sets starts decreasing later but decrease faster and vice versa. Only the sets with 5 tasks behave a bit different due to the randomness of the input.