Do Nothing, but Carefully: Fault Tolerance with Timing Guarantees for Multiprocessor Systems devoid of Online Adaptation

Georg von der Brüggen, Lea Schönberger, and Jian-Jia Chen
TU Dortmund University, Germany
{georg.von-der-brueggen, lea.schoenberger, jian-jia.chen}@tu-dortmund.de

***Abstract*—Many practical real-time systems must be able to sustain several reliability threats induced by their physical environments that cause short-term abnormal system behavior, such as transient faults. To cope with this change of system behavior, online adaptions, which may introduce a high computation overhead, are performed in many cases to ensure the timeliness of the *more important* tasks while no guarantees are provided for the *less important* tasks. In this work, we propose a system model which does not require any online adaption, but, according to the concept of dynamic real-time guarantees, provides *full timing guarantees* as well as *limited timing guarantees*, depending on the system behavior. For the normal system behavior, timeliness is guaranteed for all tasks; otherwise, timeliness is guaranteed only for the *more important* tasks while bounded tardiness is ensured for the *less important* tasks. Aiming to provide such dynamic timing guarantees, we propose a suitable system model and discuss, how this can be established by means of partitioned as well as semi-partitioned strategies. Moreover, we propose an approach for handling abnormal behavior with a longer duration, such as intermittent faults or overheating of processors, by performing task migration in order to compensate the affected system component and to increase the system's reliability. We show by comprehensive experiments that good acceptance ratios can be achieved under partitioned scheduling, which can be further improved under semi-partitioned strategies. In addition, we demonstrate that the proposed migration techniques lead to a reasonable trade-off between the decrease in schedulability and the gain in robustness of the system. The presented approaches can also be applied to mixed-criticality systems with two criticality levels.**

***Index Terms*—scheduling, mixed-criticality, fault tolerance, fault recovery, multiprocessor, real-time**

I. Introduction

Undeniably, the majority of practical real-time systems must be able to sustain several reliability threats, especially if the system is safety-critical and hard real-time characteristics must be satisfied, as prevalent in the automotive and aerospace sector. More precisely, the proper system functioning must be maintained at any point in time, comprising not only functional but also temporal correctness, i.e., a delivered result must be correct and, moreover, be obtained previous to a specified deadline. In order to ensure these properties, manifold hardware as well as software techniques have been developed so far by means of which such systems' reliability can be increased when so-called *soft errors* or *transient faults* occur, e.g., spatial isolation of certain components, hardware redundancy, remapping of logical system functionalities onto a subset of hardware resources, monitoring, and re-execution of erroneous software jobs [16]. However, these strategies are not necessarily sufficient or even applicable in all cases, since i) not every technique is fruitful with respect to each type of faults, and ii) online adaption performed in the course of fault-recovery may lead to uncertain execution behavior.

When addressing the issue of fault tolerance in real-time systems, a distinction must be made between hardware and software faults, whereat we focus on the former and do not explicitly consider software faults in this work. Regarding hardware faults, a distinction must be drawn de novo, namely, between *permanent*, *transient*, and *intermittent* faults. While permanent faults compromise a hardware component in such a way that it is entirely broken and needs to be replaced, e.g., a blown fuse or burnt cable, transient faults typically occur as bursts provoked, for instance, by power supply jitters or electromagnetic interference (EMI) due to atmospheric effects [17] and therefore only affect the functionality of a component for a limited interval of time. By way of illustration, consider a memory cell suffering bit flips due to EMI. Following from this, the hardware itself is not damaged but only provides wrong data, which can be simply overwritten [19]. If a system component is affected by an intermittent fault, its condition steadily alternates between proper functionality and malfunction. In contrast to permanent faults, intermittent faults never disappear entirely, but rather switch from an active to an inactive state and back, e.g., a loose electrical contact [19].

In this work, we focus on handling transient faults as well as other rare events via software-based techniques in the first place, while also considering abnormal behavior that occurs for a limited interval of time, such as, e.g., intermittent faults or overheating processors, later on. More precisely, considering transient faults, we assume that, in order to overcome a temporary execution uncertainty caused by a compromised hardware component, all instances (*jobs*) of related applications (*tasks*) are (at least partially) re-executed. As a consequence, the worst-case execution time (WCET) of each affected job is prolongated by a certain amount of time depending on the applied fault-recovery routine, e.g., complete re-execution [24] or checkpoint-based recovery [12]. Subsequently, we term this kind of execution behavior *abnormal execution* (with *abnormal WCET*), in contrast to the *normal execution* (with *normal WCET*) covering a task instance's plain, fault-free worst-case execution time in addition to the amount of time required for fault-detection.

Since transient faults can be considered as rare events rather than common cases, it is quite pessimistic to only take a task's abnormal WCET into account during the system design process. Thus, a technique must be developed to allow the sporadic occurrence of abnormal execution behavior without leading to an over-dimension of resources and without endangering the system safety. For this purpose, it seems sensible to divide the overall set of tasks into two distinct categories, namely, into *more important* and *less important* tasks. As *more important* tasks, we denote those tasks which must not under any circumstances miss their particular deadlines, since a deadline miss may cause hazardous consequences. Regarding the *less important* tasks, however, a small number of deadline misses can be tolerated, subject to the condition that these only occur in the fault case and, moreover, the amount of time the respective tasks perform normal execution is considerably longer than their abnormal executions.

While the origin is different, we assume that the WCETs are also prolongated if hardware components are affected by abnormal behavior for a limited interval of time. This is the case when, for instance, the CPU clock speed is decreased to reduce the temperature of an overheating processor.

A. Fault Tolerance and Mixed-Criticality

To model a system exhibiting these characteristics and behavior, the well-known mixed-criticality model [26] can be adopted, where a vector of possible worst-case execution times is associated with each task, out of which the actual WCET of a job is chosen depending on the current system mode. A classical approach in mixed-criticality systems is to neglect or even abort the jobs of less important tasks for the benefit of the more important ones' timeliness. This typically happens in the course of a so-called *mode change* from a lower-priority to a higher-priority mode, whereat the system does not return from a higher-priority mode to a lower-priority mode in many cases (for more details on mixed-criticality systems see [8]). However, steadily providing limited or even no service to less important tasks after the occurrence of a transient fault is not desirable, owing to the fact that these tasks are of some importance nonetheless, and their computation results can still be useful despite their lateness (please refer to Ernst and Di Natale [15], Esper et al. [16], and Burns and Davis [8, Section 6] for more details). Picking up this idea, new scheduling approaches for mixed-criticality systems have emerged that give at least certain reduced timing guarantees for less important tasks e.g., [3], [7], [14], [23], [25], [28]. Nevertheless, the majority of publications still require to perform adaptions at run-time as, e.g., modifying task priorities, execution times, or deadlines, decreasing the release rate of certain tasks, etc., or focus on uniprocessor systems only. A detailed overview about the state-of-the-art methods can be found in [8].

B. Dynamic Real-Time Guarantees

In order to provide a reduced level of timing guarantees for less important tasks in case of fault occurrence, the *Systems with Dynamic Real-Time Guarantees* (SDRTG) model was proposed by von der Brüggen et al. in [28], which, unlike other methods, does not require any online adaption. This is motivated by the following three evident reasons: i) online adaptions are quite costly in terms of resources, ii) online modifications of priorities, inter-arrival times of task instances, etc. are not supported by most systems, and iii) enabling online adaption requires additional configuration effort during the system design. Accordingly, in contrast to the classical mixed-criticality model, a SDRTG does *neither* perform a mode change *nor* modify any task properties if a fault occurs. In fact, simply no reaction is triggered. Notwithstanding, since one or more task instances need to be re-executed due to the faulty system component and therefore execute for an abnormal WCET, the system is described as *abnormal* in this case, while it is indicated as *normal* otherwise. These terms, however, do *not* denote different system modes as known from traditional mixed-criticality systems, but rather an observed execution behavior.

As a consequence, a SDRTG provides the following characteristics (*dynamic timing guarantees*), which are verified offline at system design time:

- *Full timing guarantees*: The system is fault-free and all tasks are executed normally, i.e., all task instances in the system meet their related deadlines.
- *Limited timing guarantees*: In case a transient fault (burst) occurs, all more important (also denoted *timing strict*) tasks meet their deadlines, while bounded tardiness is guaranteed for all less important (also termed *timing tolerable*) tasks.
- The system returns from providing *limited timing guarantees* to *full timing guarantees* after a certain interval of time, i.e., once all previous faults do not have an impact on the system anymore.

C. Multiprocessor Scheduling

As a prerequisite for providing dynamic timing guarantees in the context of multiprocessor platforms, it is necessary to choose a suitable scheduling paradigm. With respect to multiprocessor systems, three approaches are commonly followed, namely, partitioned, semi-partitioned, as well as global scheduling. Under partitioned scheduling, each task is statically allocated to a specific processor, i.e., none of its released instances must ever be executed elsewhere. On each processor, the actual schedule is determined by means of a uniprocessor scheduling policy. Global scheduling approaches, in contrast, enable each task to migrate freely between processors, such that the highest-priority task instances among all task instances in the system are executed in any point in time. These approaches are combined in the shape of semi-partitioned scheduling algorithms, which partition the tasks and allocate them to particular processors, but, nevertheless, allow a certain degree of migration, e.g., in predefined time slots or depending on specified resource constraints. A comprehensive survey on multiprocessor scheduling can be found in [11].

The main advantage of partitioned approaches is the fact that the multiprocessor scheduling problem is reduced to a set of uniprocessor scheduling problems as soon as the task partitioning is completed, while global scheduling is more costly due to task migration and online adaptions. However, determining an optimal task partition is NP-hard in the strong sense, owing to the underlying bin-packing problem. It has been shown by Brandenburg and Gul [6] that by means of semi-partitioned strategies, the schedulability of partitioned approaches can be surpassed at the cost of an only slightly increased online overhead. Accordingly, we merely employ partitioned and semi-partitioned scheduling strategies here.

D. Contributions

The approach proposed in this work aims to provide *dynamic timing guarantees* for multiprocessor systems with mixed-criticality characteristics under transient faults without any online adaption. Our contributions are:

- We propose the *Multiprocessor Systems with Dynamic Real-Time Guarantees* (MSDRTG) model in Sec. II-A as well as a related schedulability test in Sec. II-B.
- In Sec. III, we provide two scheduling approaches for MSDRTG, namely, a partitioned (cf. Sec. III-A) as well as a semi-partitioned one (cf. Sec. III-B).
- Moreover, we discuss how processors suffering from abnormal execution behavior over a limited interval of time, e.g., intermittent faults or processor clock speed drops due to overheating, can be compensated by means of task migration and introduce the concepts of *full compensation* and *partial compensation* in Sec. IV.
- We assess the developed techniques by means of comprehensive evaluations in Sec. V.

II. System Model

Consider a set of n independent sporadic (or periodic) real-time tasks $\mathbf{T} = \{\tau_1, \ldots, \tau_n\}$ to be scheduled on a multiprocessor platform with m homogeneous processors under a (semi-) partitioned fixed-priority preemptive scheduling policy. Each task releases an infinite number of task instances (*jobs*), and is defined by $\tau_i = (C_i^N, C_i^A, D_i, T_i)$ where D_i denotes the task's relative deadline and T_i its inter-arrival time or period. All deadlines are assumed to be constrained, i.e., $D_i \leq T_i$ for all tasks τ_i. To each task τ_i, two distinct worst-case execution times are associated, namely, the normal WCET C_i^N as well as the abnormal WCET C_i^A. For instance, when considering the mitigation of transient faults, C_i^N is the plain worst-case execution time and an supplementary error detection (fault-free case), and the abnormal WCET C_i^A is composed of a fault-recovery routine as well as the normal WCET. We assume that $C_i^A \geq C_i^N$ for all tasks τ_i. To fulfill its timing requirements, each task τ_k must be able to finish up to C_i^N (C_i^A) time units previous to its deadline under normal (abnormal) system behavior. The utilization of a task τ_i under normal system behavior, denoted as *normal utilization*, is identified by $U_i^N = \frac{C_i^N}{T_i}$, whereas its utilization under abnormal system behavior, referred to as *abnormal utilization*, is specified as $U_i^A = \frac{C_i^A}{T_i}$. Accordingly, the system utilization is determined by $U_{sum}^N = \sum_{\{\tau_i \in \mathbf{T}\}} U_i^N$ under normal behavior, i.e., in the fault-free case, and by $U_{sum}^A = \sum_{\{\tau_i \in \mathbf{T}\}} U_i^A$ under abnormal behavior, i.e., if a transient fault (burst) occurs. The maximum tardiness of τ_i under normal and abnormal system behavior is denoted as E_i^N and E_i^A, respectively.

Each task is either a *timing strict* task (also known as *more important* or *hard task*) or a *timing tolerable* task (also referred to as *less important* or *soft task*). The partition of the task set $\mathbf{T}$ into the related subsets $\mathbf{T}_{hard}^A$ and $\mathbf{T}_{soft}^A$ is assumed to be given. Regarding each processor p in the system, its allocated tasks are identified as $\mathbf{T}_p$ and scheduled according to a fixed-priority order P_p. For each task $\tau_i \in \mathbf{T}$, the set of higher-priority tasks on the same processor is described by $hp(\tau_k)$.

A. Multiprocessor Systems with Dynamic Real-Time Guarantees

Having specified the considered task model, we henceforth define the Multiprocessor Systems with Dynamic Real-Time Guarantees (MSDRTG) model as well as its respective properties, before establishing a suitable schedulability test.

In a system $\mathbf{S}$, consider a set of n tasks $\mathbf{T} = \{\tau_1, \ldots, \tau_n\}$ with each task $\tau_i \in \mathbf{T}$ being either timing strict, i.e., $\tau_i \in \mathbf{T}_{hard}^A$, or timing tolerable, i.e., $\tau_i \in \mathbf{T}_{soft}^A$, so that $\mathbf{T}_{hard}^A \cap \mathbf{T}_{soft}^A = \emptyset$ and $\mathbf{T}_{hard}^A \cup \mathbf{T}_{soft}^A = \mathbf{T}$. The task set $\mathbf{T}$ is partitioned onto a set of m homogeneous processors, in such a way that $\mathbf{T}_j \cap \mathbf{T}_k = \emptyset$ for all $j \neq k$ with $j, k \in \{1, \ldots, m\}$ and $\mathbf{T}_1 \cup \mathbf{T}_2 \cup \cdots \cup \mathbf{T}_m = \mathbf{T}$. Each set of tasks $\mathbf{T}_p$ allocated to a processor p with $1 \leq p \leq m$ is identified as *subsystem* of $\mathbf{S}$ and scheduled according to a fixed-priority task order P_p. For each $\tau_i \in \mathbf{T}_p$, it must hold that no task instance can begin its execution before all previously arrived instances released by the same task are completed. Moreover, no task instance of any $\tau_i \in \mathbf{T}_p$ must ever be aborted.

Definition 1 (Multiprocessor System with Dynamic Real-Time Guarantees)**.** *A system $\mathbf{S}$ with subsystems $\mathbf{T}_1, ..., \mathbf{T}_m$ is an MSDRTG if and only if each subsystem $\mathbf{T}_p$ satisfies the characteristics of an System with Dynamic Real-Time Guarantees [28], i.e., if under normal system behavior, all subsystems provide full timing guarantees, and under abnormal system behavior at least one affected subsystem provides limited timing guarantees for a bounded interval of time.*

A subsystem $\mathbf{T}_p$ provides *full timing guarantees* if hard real-time constraints are satisfied for each task τ_i allocated to processor p. More precisely, under normal system behavior, all instances of each task $\tau_i \in \mathbf{T}_p$ meet their hard relative deadlines, i.e., $E_i^N = 0 \, \forall \tau_i \in \mathbf{T}_p$. In a subsystem $\mathbf{T}_p$ providing limited timing guarantees, service level guarantees may be downgraded for some timing tolerable tasks $\mathbf{T}_{p,soft}^A \subseteq \mathbf{T}$, so that all $\tau_i \in \mathbf{T}_{p,soft}^A$ have (at least) bounded tardiness, i.e., $0 \leq E_i^A < \gamma_i \; \forall \; \tau_i \in \mathbf{T}_{p,soft}^A$ for a fixed γ_i. However, each instance of each $\tau_i \in \mathbf{T}_{p,hard}^A$ meets its hard relative deadline irregardless, i.e., $E_i^A = 0 \; \forall \; \tau_i \in \mathbf{T}_{p,hard}^A$. An MSDRTG $\mathbf{S}$ provides full timing guarantees if all subsystems $\mathbf{T}_p$ provide full timing guarantees, and limited timing guarantees if at least one subsystem provides limited timing guarantees.

B. Schedulability Test

Owing to the fact that an MSDRTG $\mathbf{S}$ as defined above is composed of a set of subsystems $\{\mathbf{T}_1, \ldots, \mathbf{T}_m\}$, each of which comprises a set of tasks scheduled according to an individual fixed-priority order, the schedulability test for uniprocessor

SDRTG proposed in [28] can be adapted, which is based on Time Demand Analysis [21] (TDA). TDA is an exact schedulability test for fixed-priority preemptive uniprocessor scheduling of constrained-deadline task sets, by means of which the schedulability of a task τ_i is determined under a given priority order, assuming that the schedulability of all higher-priority tasks is already ensured.

Definition 2 (Time Demand Analysis [21]). *A constrained-deadline task τ_i is schedulable if and only if the following equation holds:*

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

A contemplated task set is schedulable under fixed-priority preemptive scheduling according to a given priority order if the above condition holds for each task $\tau_i \in \mathbf{T}$.

Based on the schedulability test for SDRTG in [28], the schedulability test for MSDRTGs is given as follows:

Theorem 1 (Exact Schedulability Test for MSDRTGs with Constrained Deadlines under Partitioned Scheduling). *A task system $\boldsymbol{S}$ with a given partition of a task set $\boldsymbol{T}$ into subsets $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_m$, a given partition of $\boldsymbol{T}$ into $\boldsymbol{T}^A_{hard}$ and $\boldsymbol{T}^A_{soft}$, and a given priority order P_p for each subset $\boldsymbol{T}_p \in \{\boldsymbol{T}_1, \ldots, \boldsymbol{T}_m\}$ is an MSDRTG if and only if the following conditions hold:*

1) *Each subsystem $\boldsymbol{T}_p$ can be scheduled according to Time Demand Analysis [21] under the given priority order P_p and normal system behavior, i.e., $C_i = C_i^N \; \forall \tau_i \in \boldsymbol{T}_p$.*
2) *All $\tau_i \in \boldsymbol{T}^A_{p,hard}$ can be scheduled according to TDA under the given priority order P_p and abnormal system behavior, i.e., $C_i = C_i^A \; \forall \tau_i \in \boldsymbol{T}_p, \forall \boldsymbol{T}_p \in \boldsymbol{T}$.*
3) *For each subsystem $\boldsymbol{T}_p$, it holds that $U^A_{p,sum} \leq 1$.*

Since the timing requirements of all timing strict tasks must be satisfied at any point in time and since $C_i^N \leq C_i^A \; \forall \tau_i \in \mathbf{T}$ by definition, only their abnormal WCET must be taken into account when testing the schedulability of an MSDRTG. Moreover, a subsystem utilization of $U^A_{p,sum} > 1$ may be permitted in case of abnormal behavior, provided that $U^N_{p,sum} < 1$ and that the system exhibits normal behavior for a considerably longer amount of time than abnormal behavior. Hence, we omit the condition that $U^A_{p,sum} \leq 1$ in our further discussions as well as in the evaluation.

Note that if the probability of abnormal execution behavior is independent for each job, the probability that a deadline is missed can be upper bounded using the approaches in [10], [29], while in [9] an over-approximation of the deadline miss rate is presented.

III. Multiprocessor Scheduling

Having provided the formal specifications of an MSDRTG as well as a suitable schedulability test, we henceforth elucidate, how to obtain the actual task partition and priority assignment. Furthermore, we discuss, how to increase the number of tasks that are feasibly schedulable as an MSDRTG, namely, by means of task migration (cf. Sec. III-B).

A. Partitioned Scheduling

Adopting the partitioned scheduling paradigm, two distinct problems must be considered: In a first step, the overall task set $\mathbf{T}$ must be partitioned so that each task $\tau_i \in \mathbf{T}$ is statically allocated to a specific processor. Thereon, a priority order must be specified for each subsystem $\mathbf{T}_p$.

An intuitive solution to the first subproblem is to apply the *deadline-monotonic partitioning algorithm* by Baruah and Fisher [2], in the course of which all $\tau_i \in \mathbf{T}$ are initially pre-ordered increasingly with respect to their relative deadline, i.e., $D_i \leq D_{i+1}$. Thereafter, each task is assigned to the first processor that satisfies the scheduling condition, i.e., Theorem 1 for an MSDRTG. This algorithm can be easily modified using other heuristics, such as arbitrary-fit (AF), worst-fit (WF), best-fit (BF), etc., and other task set pre-orders.

In order to address the second subproblem, namely, the establishment of a priority order, the *Feasible Priority Assignment Algorithm* for (uniprocessor) System with Dynamic Real-Time Guarantees [28] can be adopted for each subsystem $\mathbf{T}_p$, which is based on Audsley's Optimal Priority Assignment [1]. Owing to the fact that if a feasible priority assignment exists, a feasible priority assignment in which the tasks in $\mathbf{T}_{p,hard}$ and $\mathbf{T}_{p,soft}$ are internally arranged according to a deadline-monotonic order exists as well (as shown in [28]), only two tasks must be considered on each priority level, namely, the one (among the tasks that have not yet been assigned a priority) with the longest relative deadline in $\mathbf{T}_{p,hard}$ and the one (among the tasks that have not yet been assigned a priority) with the longest relative deadline in $\mathbf{T}_{p,soft}$. Resulting from this, the time required to retrieve a feasible priority order is decreased substantially. Nevertheless, this implies that for each attempt of allocating a new task τ_t to a particular processor, the priority assignment algorithm must be executed once again, since even if τ_t cannot be scheduled together with all $\tau_i \in \mathbf{T}_p$ under the priority order P_p, the set $P_p \cup \{\tau_t\}$ may be schedulable under another priority order. Hence, the priority assignment on each processor may change whenever a new task is allocated to that processor.

B. Task Migration

When following a partitioned scheduling approach, it may be the case that no further tasks can be joined with a subsystem $\mathbf{T}_p$, but, notwithstanding, some spare capacity is left on the respective processor p. In this event, it seems sensible to allow the not yet allocated tasks to split into so-called *subtasks*, which are executed on more than one processor. More precisely, we attempt to fill each processor to the maximum by assigning the largest possible task shares. As a consequence, task sets can be scheduled as an MSDRTG exhibiting a higher system utilization than feasible under merely partitioned techniques.

Nonetheless, this idea entails additional challenges. In particular, it is necessary to decide which tasks to share, how to compute the spare capacity of a processor, i.e., how to derive the largest possible worst-case execution time of a subtask, and how to specify its deadline. In addition, it must be determined, which priority to assign to each individual subtask and how to ensure the correct execution order of a sequence of subtasks.

Contemplate a task $\tau_t \in \mathbf{T}$ that cannot be allocated to any processor under a given partitioned technique. In this case, a certain task τ_s, which can be either the task τ_t that could not be assigned successfully or a task that is already part of the considered subsystem, i.e., a $\tau_s \in \mathbf{T}_p$, may be shared between at least two processors such that τ_s is not executed by more than one processor at the same time. As a consequence of this sequential execution, a release of a subtask of the so-called *shared task* τ_s on a processor $p+1$ must not take place any earlier than the respective subtask executed on processor p is finished. To ensure this property, it seems reasonable to execute each subtask as early as possible, i.e., as soon as it is released. For this reason, we always assign the highest priority in the subsytem to each subtask of τ_s. Owing to the fact that the remaining capacity on each processor does not suffice to execute τ_s completely, all processors maintaining one share of τ_s are filled to maximum capacity after the splitting operation, except presumably the one covering the last subtask which indeed represents a special case. In contrast to the remaining processors, this particular one can still affiliate a subtask of another shared task. Since the next task instance cannot be released before the previous one is finished, the execution order of the first shared task cannot be perturbated anyway. Hence, we endow the latter one with the highest priority in the respective subsystem. Subsuming the aforesaid, a precise definition of the term *shared task* shall be given.

Definition 3 (Shared Task)**.** *A task $\tau_s \in \boldsymbol{T}$ 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 by more than one processor at the same time, but is successively passed on to the next processor as soon as its execution budget on the considered one 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 one. In this case, the later allocated subtask is preferred in terms of priority, since the previously assigned task segment is the last segment of the related task.*
- *Concerning all calculations involving the shared task's worst-execution time, C_s^A is considered regardless of the actual system behavior.*

Each share of τ_s, denoted as subtask, is treated as an individual task $\tau_{s,p} = (C^N_{s,p}, C^A_{s,p}, D_s, T_s)$.

Having clarified how to specify the deadlines of each subtask, how to assign its priority, and how to ensure the correct execution order of a sequence of subtasks, we henceforth discuss how to compute the maximum amount of time a subtask can execute on a particular processor p, i.e., its worst-case execution time, which was assumed to be given up to now. Consider a task τ_s to be assigned to processor p as well as the set of previously allocated tasks $\mathbf{T}_p$ with a priority order P_p. Availing the method proposed in [18], it is possible to compute the maximum amount of time τ_s can be executed on the respective processor without causing a task $\tau_i \in \mathbf{T}_p$ to miss its deadline. The minimum value derived for any $\tau_i \in \mathbf{T}_p$ determines the worst-case execution time $C_{s,p}$ of subtask $\tau_{s,p}$ on p. Accordingly, a subtask of length $C_{s,p}$ is split off and the worst-case execution time budget of τ_s, i.e., the remaining workload to be distributed to processors with spare capacity, is reduced by $C_{s,p}$. This operation is repeated until either no further workload needs to be distributed or the system is deemed to be unschedulable. Please note that we always refer to the abnormal WCET C_s^A throughout the task splitting.

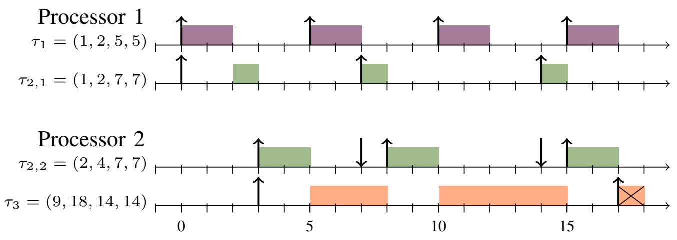

Fig. 1: Task τ_3 misses its deadline due to the release jitter of subjob $\tau_{2,2}$ on processor 2 resulting from the execution of subjob $\tau_{2,1}$ on processor 1.

When a shared task τ_s is migrated from processor p to processor $p+1$, the so-called *release jitter* of each subtask must be investigated as well, i.e., the difference between its best- and worst-case response time. An example is given in Fig. 1, outlining 3 tasks being distributed to two processors, where task τ_1 and τ_3 are statically allocated to processor 1 and 2, respectively. Here, the first subtask $\tau_{2,1}$ of the shared task τ_2 is always executed on processor 1, while its second subtask is always executed on processor 2. Due to the additional workload produced by the higher-priority task τ_1 in the interval $[0; 2]$, the first instance of $\tau_{2,1}$ has a larger response time than its second and third instance. Resulting from this, $\tau_{2,2}$ is not released exactly periodically on processor 2, but with a release jitter of 2 time units, which, in turn, leads to a deadline miss of τ_3, indicated by the cross. This can be avoided by releasing $\tau_{2,2}$ exactly periodically or sporadically, such that its maximum workload in any interval of length 14 comprises 4 time units and, as a consequence, τ_3 always meets its deadline. Owing to the fact that each subtask (except possibly the last one) of a shared task is always executed under the highest priority in the respective subsystem, its best- and worst-case response time do not differ. Other factors potentially inducing jitters - although on a smaller scale - are the time required for task preemption as well as the migration time. However, we assume the preemption time to be sufficiently small to be neglected, whereas we assume that for a given subtask the related migration time is always constant. Accordingly, we can consider all subtasks as periodic tasks without any release jitter in our analysis. In a conflicting case, release enforcement techniques can be used as, e.g., explained in [4], [27].

As our subject of discussion, the question remains, which task to choose as a shared task. In fact, two diverging strategies can be pursued: either the task τ_t which could not be allocated under a partitioned approach without making the system unschedulable as an MSDRTG, or an already assigned task $\tau_i \in \mathbf{T}_p$. While the first case can be handled as elucidated above, the latter one can be dealt with as proposed in [20]. More precisely, the highest-priority task in $\mathbf{T}_p$ is chosen to be shared between processors, owing to the fact that the highest-priority task on each processor typically has a short relative deadline, whereas the task τ_t to be assigned usually has a larger deadline[1] as well as a larger WCET. Moreover, it is well known that assigning a higher priority to a task with a shorter period or deadline is in general favorable [22].

Accordingly, after applying a task allocation strategy of choice until its point of failure, we attempt to assign each remaining task $\tau_t \in \mathbf{T}$ by identifying a processor p on which τ_t can be scheduled along with $\mathbf{T}_p \setminus \left\{\tau_p^h\right\}$, where τ_p^h is the highest-priority task on processor p. Thereon, $\tau_s = \tau_p^h$ is divided into multiple subtasks and shared across a number of processors applying the previously explained method until the workload of τ_s is completely distributed. As soon as each $\tau_i \in \mathbf{T}$ is assigned to one (or more) processor(s), the algorithm terminates successfully. Otherwise, the task set $\mathbf{T}$ is declared to be unschedulable as an MSDRTG.

Further strategies for increasing the number of schedulable task sets can be applied as well, such as, e.g., removing previously assigned tasks τ_r from a subsystem in order to allow a feasible allocation of the currently considered task τ_t, while reconsidering τ_r later on. A comprehensive survey including conceivable approaches can be found in in [11]. We renounce further discussion due to space limitations.

IV. Compensating Faulty Processors

So far, we proposed a system model by means of which dynamic timing guarantees, i.e., full timing guarantees for timing strict tasks and limited timing guarantees for timing tolerable tasks, can be given without any online adaption in case one or more system components are affected by a transient fault. Moreover, we suggested a number of scheduling strategies by means of which such an MSDRTG can be established.

Henceforth, we introduce a compensation technique for components evincing abnormal behavior for a limited interval of time, aiming to achieve additional tolerance with respect to, for instance, intermittent faults or a decreased CPU clock frequency. We term a system component, i.e., a subsystem $\mathbf{T}_p$, *corrupted* if it exhibits abnormal execution behavior.

If this is the case, a certain subset of tasks scheduled on the corrupted subsystem $\mathbf{T}_p$ must be migrated to other processors to satisfy their timing requirements. Since timeliness is guaranteed for all tasks $\tau_i \in \mathbf{T}_{p,hard}$ under abnormal system behavior by definition, these tasks may remain on the corrupted processor even if the subsystem exhibits abnormal behavior for a longer interval of time, as long as for each $\tau_i \in \mathbf{T}_p$ a correct result can be obtained within the respective abnormal WCET C_i^A. For all tasks $\tau_i \in \mathbf{T}_{soft}^A$, bounded tardiness is already guaranteed if $U_{p,sum}^A \leq 1$ holds, but, anyhow, we remove this condition, assuming that the intervals in which the system exhibits abnormal behavior are significantly shorter than those under normal behavior. As a consequence, neither timeliness nor bounded tardiness can be ensured for any $\tau_i \in \mathbf{T}_{soft}^A$; hence their migration would be beneficial.

For this migration process, it is necessary to determine a specific order in which the tasks are migrated away from $\mathbf{T}_p$. However, we cannot decide which tasks to favor by means of their particular function or purpose, owing to the fact that all timing tolerable tasks are considered as equally important in a SDRTG. Nevertheless, it is rather appropriate to factor in another property when taking this decision. More precisely, we classify the tasks in $\mathbf{T}_{p,soft}$ into three categories:

- Tasks that anyway meet their hard deadline under abnormal system behavior, denoted as $\mathbf{T}_{p,soft}^{guar}$. These can be neglected within the migration process.
- Tasks for which no timeliness but at least bounded tardiness can be guaranteed under abnormal system behavior, identified as $\mathbf{T}_{p,soft}^{bd}$.
- Tasks for which no guarantees can be given under abnormal system behavior[2], referred to as $\mathbf{T}_{p,soft}^{unbd}$.

Depending on the number of corrupted subsystems as well as on their particular task sets, two levels of compensations may be possible. Actually, a corrupted processor can be *fully compensated* if the system maintains the characteristics of an MSDRTG after the migration process. Otherwise, it can be *partially compensated* if all $\tau_i \in \mathbf{T}_{hard}^A$ meet their hard deadlines under abnormal system behavior, whereas at least bounded tardiness can be guaranteed for each $\tau_i \in \mathbf{T}_{soft}^A$, provided that the destination processor(s) of the task migration are not affected by an intermittent fault.

Regarding the migration process, we begin with the tasks in $\mathbf{T}_{p,soft}^{unbd}$, for which guarantees can be provided neither under full nor under partial compensation. Concerning this subset, we make the following observations: Since the task utilization is an increasing function with respect to the priority, only tasks with a priority lower than any $\tau_i \in \mathbf{T}_{p,hard}$ can be in $\mathbf{T}_{p,soft}^{unbd}$. Moreover, if a task τ_j is in $\mathbf{T}_{p,soft}^{unbd}$, all tasks having lower priority than τ_j are in $\mathbf{T}_{p,soft}^{unbd}$ as well, while each $\tau_i \in \mathbf{T}_{p,soft}^{bd}$ has a higher priority than each $\tau_j \in \mathbf{T}_{p,soft}^{unbd}$. Accordingly, $\mathbf{T}_{p,soft}^{unbd}$ can be computed easily. As soon as $\mathbf{T}_{p,soft}^{unbd}$ has been determined, we try to assign each $\tau_j \in \mathbf{T}_{p,soft}^{unbd}$ to a non-corrupted processor.[3] If τ_j cannot be allocated to such a processor or no such processor exists, we search for a processor q where timeliness can still be guaranteed for all tasks $\tau_i \in \mathbf{T}_{q,hard}$, while bounded tardiness is ensured for all tasks $\tau_i \in \mathbf{T}_{q,soft} \cup \tau_j$. If at least one task in $\mathbf{T}_{p,soft}^{unbd}$ cannot be assigned either way, the corrupted processor(s) *cannot* be compensated. If, in contrast, migration of at least one task $\tau_j \in \mathbf{T}_{p,soft}^{unbd}$ leads to bounded tardiness for some of the tasks in $\mathbf{T}_{q,soft} \cup \tau_j$ the corrupted processor(s) can be *partially compensated*. Otherwise, if all non-corrupted subsystems exhibit the characteristics of an System with Dynamic Real-Time Guarantees after the migration of $\mathbf{T}_{p,soft}^{unbd}$, we continue to migrate the tasks in $\mathbf{T}_{p,soft}^{bd}$ in deadline-monotonic until either the characteristics of an MSDRTG are restored for

[1]This depends on the order in which the tasks are partitioned. Nevertheless, even for an inverted deadline-monotonic order, the unassigned task τ_t most commonly has a relative deadline that is not considerably shorter than the one of the highest-priority task $\tau_p^h \in \mathbf{T}_p$.

[2]This is only possible since we dropped the condition that $U_{p,sum}^A \leq 1$.

[3]Tasks in $\mathbf{T}_{p,soft}^{unbd}$ are considered in deadline-monotonic order during the migration process.

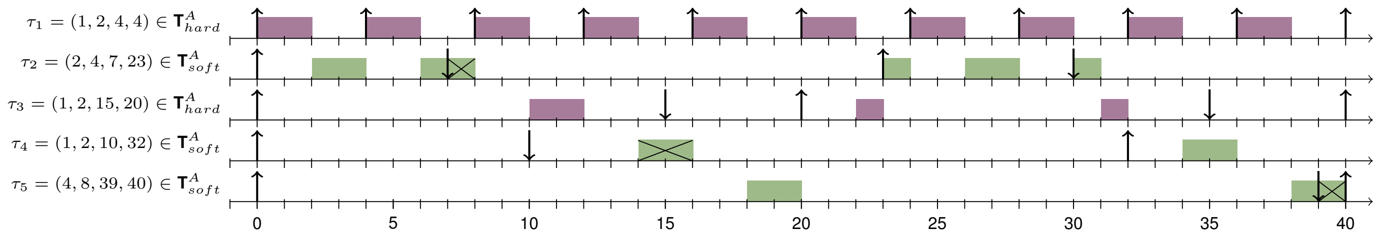

Fig. 2: Migration example for $\mathbf{T}_p$ with $\mathbf{T}_{p,hard} = \{\tau_1, \tau_3\}$, $\mathbf{T}_{p,soft} = \{\tau_2, \tau_4, \tau_5\}$, and $U^A > 1$. First, $\tau_5 \in \mathbf{T}^{unbd}_{p,soft}$ is migrated, resulting in $U^A \leq 1$. Thereon, τ_2 is migrated since $\tau_2, \tau_4 \in \mathbf{T}^{bd}_{p,soft}$ and τ_2 has higher priority. Hence, τ_4 meets its deadline.

the system $\mathbf{S}$ (in this case, the corrupted processor(s) can be *fully compensated*) or one task $\tau_j \in \mathbf{T}^{bd}_{p,soft}$ cannot be assigned to any processor q in such a way that the System with Dynamic Real-Time Guarantees property is maintained for $\mathbf{T}_p \cup \tau_j$. Please note that migrating a small number of tasks $\tau_j \in \mathbf{T}^{bd}_{p,soft}$ from a corrupted subsystem $\mathbf{T}_p$ to another one, may already reduce the workload on $\mathbf{T}_p$ sufficiently to provide *full compensation*, so that further migration is unnecessary.

Concerning the actual migration, we assume that all instances of a task τ_i that is migrated from a subsystem $\mathbf{T}_p$ to a subsystem $\mathbf{T}_q$ are terminated on $\mathbf{T}_p$ previous to the migration and restarted on $\mathbf{T}_q$ afterwards. More precisely, the release of the first instance of τ_i on $\mathbf{T}_q$ occurs in the same moment in which its next release on $\mathbf{T}_p$ would have taken place. This indeed implies that all jobs terminated on $\mathbf{T}_p$ are lost, but since it can be assumed that abnormal behavior over a longer interval of time is rare compared to transient faults, this can be tolerated for the benefit of the system robustness.

By way of illustration, consider Figure 2 which portrays a corrupted subsystem under abnormal behavior comprising 5 distinct tasks. Under abnormal system behavior, it holds for the system utilization that $U^A_{sum} \approx 1.036 > 1$, for which reason we are able to categorize the tasks as follows: $\tau_5 \in \mathbf{T}^{unbd}_{p,soft}$, $\tau_2, \tau_4 \in \mathbf{T}^{bd}_{p,soft}$, and $\tau_1, \tau_3 \in \mathbf{T}^A_{hard}$. If τ_5 can be successfully migrated to another processor, the corrupted processor is *partially compensated*. If, moreover, τ_2 can be migrated, τ_4 meets its deadline under abnormal system behavior and the System with Dynamic Real-Time Guarantees property is restored for $\mathbf{T}_p$. Finally, the corrupted subsystem is *fully compensated* if τ_5 and τ_2 can be migrated, such that the system $\mathbf{S}$ exhibits the characteristics of an MSDRTG thereafter.

Although enabling full or partial compensation increases the robustness and thus the safety of an MSDRTG, a compromise must be made, since this leads to a smaller number of schedulable task sets. In the course of our evaluations in Sec. V, we will examine this trade-off more thoroughly.

Unfortunately, our proposed method is not applicable to recover from permanent faults. This follows from the fact that a system component affected by a permanent fault is entirely inoperable, for which reason not only all timing tolerable tasks but, in addition, all timing strict tasks need to be migrated to another subsystem. In this event, the migration itself leads to manifold problems for timing strict tasks, e.g., regarding the problem of ensuring their timeliness during the migration process. Therefore, addressing this issue is beyond the scope of this work.

V. Evaluation

In the following, we discuss the results of our comprehensive evaluations, in the course of which we analyzed the schedulability of randomized synthetic task sets as MSDRTGs under different processor assignment, task splitting, and compensation techniques. The evaluation setup is described in Section V-A while the results are presented in Section V-B.

A. Experiment Setup

We randomly generated implicit-deadline task sets, i.e., $D_i = T_i \; \forall \tau_i$. The number of processors m and the number of tasks n in the set differs depending on the analyzed setting, i.e., $m = 4$, 8, or 16, and $n = 40$, 80, or 160. For a given m, n, and total utilization U^N_{sum}, we generated tasks according to the UUniFast method [5]. For m processors, the values of U^N_{sum} ranged from $2\% \times m$ to $100\% \times m$ with steps of $2\% \times m$. For each setting and each utilization value, 1000 randomly generated task sets were evaluated under 16 scheduling strategies. We also evaluated if full or partial compensation was possible for each task set, assuming that one processor was corrupted. The task periods were drawn randomly according to a log-uniform distribution with two orders of magnitude as suggested by Emberson et al. [13], i.e., $\log_{10} T_i$ was a uniform distribution over $[1ms - 100ms]$. The WCET under normal system behavior was set according to the utilization, i.e., $C^N_i = U_i \cdot T_i$, and 50% of the tasks were randomly chosen to be in $\mathbf{T}^A_{hard}$; the remaining tasks were part of $\mathbf{T}^A_{soft}$.

In terms of fault-recovery, we considered one re-execution, two re-executions, and checkpointing of a task, leading to the following ratios between C^N_i and C^A_i:

- Re-Execution: $C^A_i \approx 1.83 \cdot C^N_i$ as $\frac{2.2}{1.2} \approx 1.83$. One fault detection with an assumed overhead of 20% is performed at the end of the normal execution. If a fault is detected, the task is completely re-executed.
- Two Re-Executions: $C^A_i \approx 2.83 \cdot C^N_i$ as $\frac{3.4}{1.2} \approx 2.83$. Two re-executions and two fault detections, one after the normal execution and one after the first re-execution.
- Checkpointing: $C^A_i \approx 1.14 \cdot C^N_i$ as $\frac{1.6}{1.4} \approx 1.14$. The occurrence of a fault is tested at multiple checkpoints during the normal execution. We assumed a total overhead of 40% for the detection and that 20% of the task have to be re-executed.

We used the same ratio values, denoted *WCET-factors*, for both $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$. Since tasks with an abnormal utilization over 100% cannot be scheduled on one processor by default, such

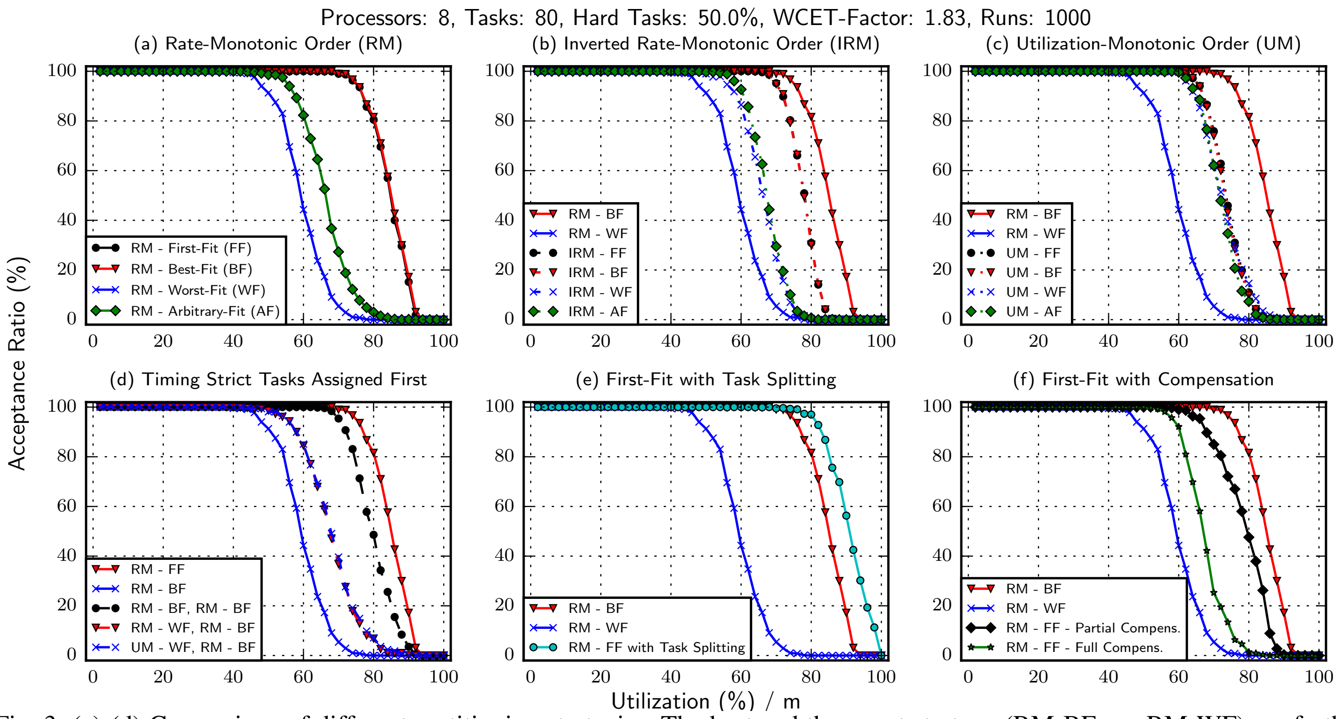

Fig. 3: (a)-(d) Comparison of different partitioning strategies. The best and the worst strategy (RM-BF ans RM-WF) are further compared to (e) our semi-partitioned approach with highest-priority task splitting, and (f) our compensation techniques.

tasks were discarded during the random generation and a new task was drawn instead.

B. Results

In our evaluations, we tried to allocate tasks to processors for different combinations of task pre-orders and partitioning strategies. During the task pre-ordering, $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ were *not* sorted separately. We considered the following pre-orders:

1) Rate-Monotonic Order (RM): The tasks were sorted increasingly with respect to their period T_i.
2) Inverted Rate-Monotonic Order (IRM): Tasks with longer period T_i were allocated first.
3) Utilization-Monotonic Order (UM): Tasks with higher utilization U_i^N were allocated first.

We considered four well-known assignment strategies for partitioned task scheduling:

1) First-Fit (FF): Testing order based on the processor ID.
2) Best-Fit (BF): Processors are considered in decreasing order with respect to their utilization.
3) Worst-Fit (BF): Processors are tested in increasing order with respect to their utilization.
4) Arbitrary-Fit (AF): Processors are tested in random order.

Under these assignment strategies, all tasks are always allocated to the first possible processor. The three pre-orders combined with the four assignment strategies led to a total of 12 basic partitioned scheduling approaches. The results for a setup with 8 processors, 80 tasks, and a WCET-factor of 1.83, i.e., one re-execution, can be found in Figure 3(a)-(c). The labels indicate the considered pre-order and the applied assignment strategy, e.g., RM-FF for rate-monotonic pre-order with first-fit assignment strategy.

In Figure 3(a), the assignment strategies are compared under a rate-monotonic preorder. RM-FF and RM-BF performed nearly identical with a slight advantage for RM-BF. Both acceptance ratios start dropping noticeably at $80\% \times m$, whereas RM-WF performed worst and the acceptance ratio breaks down $25\% \times m$ earlier than for RM-BF and RM-FF. The reason is that under a worst-fit approach the utilization is distributed equally while under the first-fit and best-fit strategies the processors are filled as densely as possible. Therefore, a single task with a long period and high utilization can easily lead to a case in which no processor has sufficient remaining capacity when the worst-fit strategy is combined with a rate-monotonic preorder. RM-AF performs slightly better than RM-WF because the randomness of the approach sometimes prevented the aforementioned worst-case scenario. Since RM-BF and RM-WF performed best and worst, they serve as reference values for all other approaches in the following subfigures of Figure 3.

With respect to the inverted rate-monotonic (IRM) order, IRM-FF and IRM-BF as well as IRM-WF and IRM-AF performed similar, as shown in Figure 3(b). While IRM-FF and IRM-BF were slightly worse than RM-BF, IRM-WF and IRM-AF accepted more task sets than RM-WF. With a utilization-monotonic (UM) pre-order, all strategies led to a nearly identical acceptance ratio, especially between RM-BF and RM-WF, as portrayed in Figure 3(c).

Furthermore, we tested three partitioned approaches where the tasks in $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ were assigned separately, namely:

1) RM-BF+RM-BF: Initially, the tasks in $\mathbf{T}^A_{hard}$ and, thereon, the tasks in $\mathbf{T}^A_{soft}$ were partitioned using the RM-BF approach.
2) RM-WF+RM-BF: First, RM-WF was applied for $\mathbf{T}^A_{hard}$, then RM-BF for $\mathbf{T}^A_{soft}$.
3) UM-WF+RM-BF: The tasks in $\mathbf{T}^A_{hard}$ were assigned

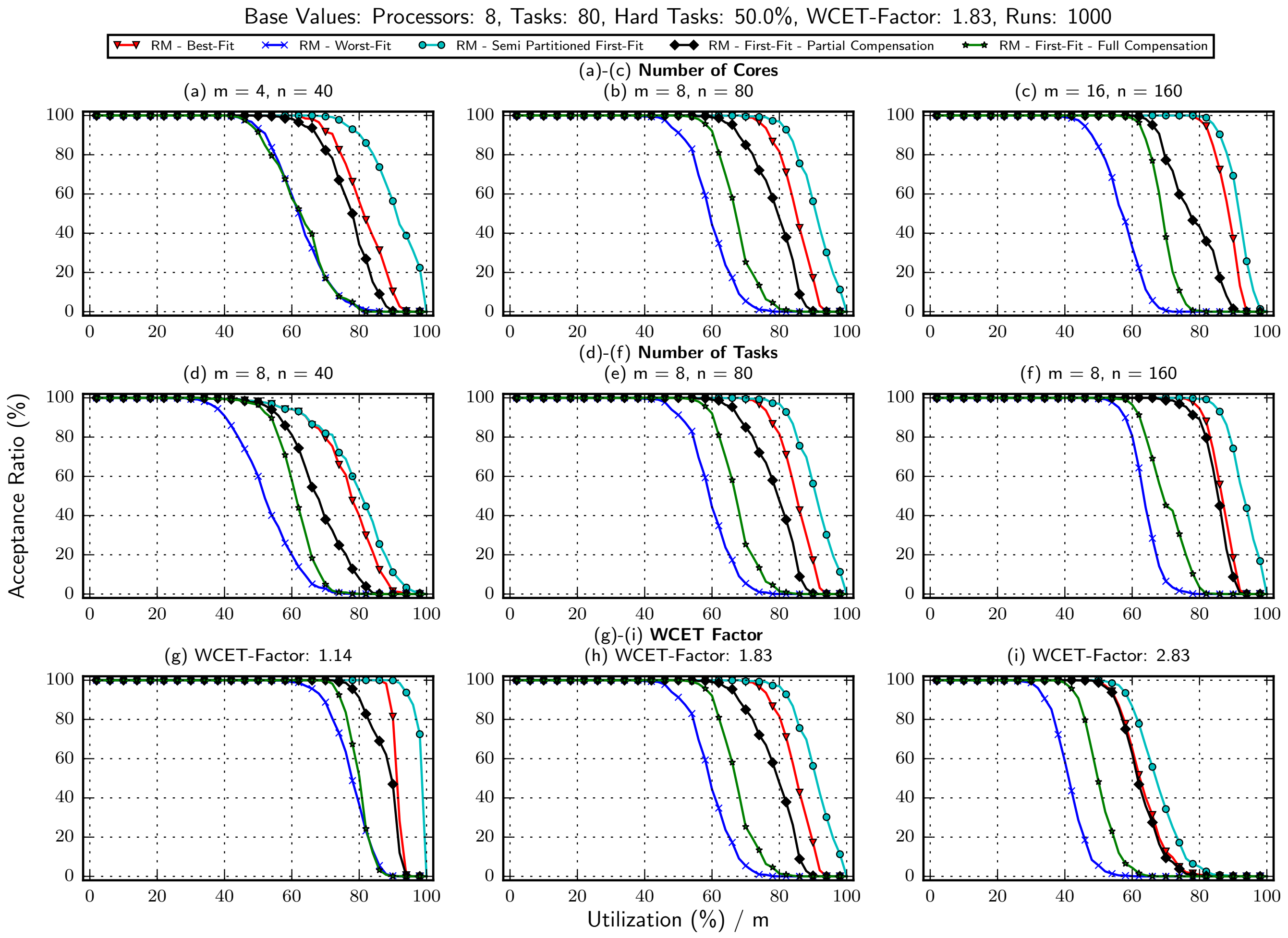

Fig. 4: Impact of: (a)-(c) the number of processors, (d)-(f) number of tasks, and (g)-(i) WCET on the acceptance ratio.

according to UM-WF, thereafter $\mathbf{T}^A_{soft}$ via RM-BF.

The results are shown in Figure 3(d). RM-BF+RM-BF surpasses the other two approaches but is still outperformed by RM-BF. The two other approaches performed slightly better than RM-WF.

Combining Figures 3(a)-(d), we conclude that RM-BF is superior among the considered partitioned scheduling approaches, while RM-WF performed worst. All other applied preorders and partitioned strategies as well as assigning $\mathbf{T}^A_{hard}$ and $\mathbf{T}^A_{soft}$ separately led to an acceptance rate higher than of RM-WF and a lower than of RM-BF.

Aiming to analyze the benefit of semi-partitioned scheduling with respect to the schedulability, we implemented a rate-monotonic first-fit strategy with highest-priority task splitting, denoted as RM-FF-TS. It can be seen in Figure 3(e) that RM-FF-TS was superior to RM-FF in the evaluation, i.e., the acceptance ratio of RM-FF-TS drops roughly $6\% \times m$ later than for RM-BF. Even for a utilization of $98\% \times m$, some task sets were still schedulable as an MSDRTG.

Finally, we assess how providing full or partial compensation of one corrupted processor affects the schedulability. To be more precise, we considered all cases individually, in which one processor was corrupted, and determined if the respective processor could be compensated. If full (partial) compensation was possible for all processors, the system provided full (partial) compensation. In Figure 3(f), the resulting detriment can be observed for a setting in which we applied a RM-FF for both the initial partition and the compensation. While for full compensation the loss was nearly $20\% \times m$, the loss for partial compensation was $10\% \times m$. This means, *full* and *partial compensation* are often achieved for the expense of $60\% \times m$ and $70\% \times m$ system utilization in this setting.

Since the relation between the considered scheduling approaches was similar in all settings, we focused on analyzing the impact of the different parameters on the schedulability. We show the acceptance ratio for the RM-BF and RM-WF approach as well as for the semi-partitioned scheduling approach and the compensation techniques in Figure 4. We analyzed the effect of three different parameters:

1) **Number of processors** in Figure 4(a)-(c): The acceptance ratio increased, when a larger number of processors is considered. This result was not unexpected, since for a constant average processor utilization and a constant average number of tasks per processor, a larger number of processors results in more possibilities to allocate the tasks. Only for RM-WF, increasing the number of processors had no positive effect.
2) **Number of tasks** in Figure 4(d)-(f): Increasing the number of tasks in the systems for a constant number of processors increased the acceptance ratio as well,

since the average task utilization is decreased and smaller tasks can usually be easier allocated. The gap between the acceptance ratios for systems with full and partial compensation became larger for a larger number of tasks.

3) **WCET-Factor** in Figure 4(g)-(i): As expected, increasing the WCET-factor led to a decrease in schedulability.

Summarizing the aforementioned evaluation results, it becomes evident that a rate-monotonic best-fit approach leads to a good acceptance ratio under partitioned scheduling when designing Multiprocessor Systems with Dynamic Real-Time Guarantees. If semi-partitioned scheduling techniques are considered, namely, an approach with highest priority task splitting, this acceptance ratio can be further increased. The analysis of the presented compensation techniques showed a decreased acceptance ratio, compared to the best partitioned strategy. Nevertheless, the trade-off between acceptance ratio and reliability seems reasonable, which means that intermittent faults can be compensated for MSDRTGs.

VI. Conclusions

In this paper, we addressed the problem of multiprocessor real-time scheduling under uncertain execution behavior which may occur with low probability, due to, for instance, transient faults, as well as within intervals of bounded length, e.g., owing to intermittent faults or decreasing CPU clock speed. We introduced the *Multiprocessor Systems with Dynamic Real-Time Guarantees* model, which provides *dynamic timing guarantees*, i.e., full timing guarantees for timing strict tasks and limited timing guarantees for timing tolerable tasks, without the necessity of any online adaption in case of fault-occurrence. We clarified how such a system can be established under partitioned scheduling and exploited semi-partitioned scheduling techniques to further increase the schedulability. Moreover, we introduced the concept of *full and partial compensation* to enhance the system reliability in the case that one or more processors suffer from abnormal execution behavior during a longer interval of time. Our evaluations show that reasonable acceptance ratios can be achieved for partitioned and semi-partitioned scheduling. Not least, we unveiled that improving the system robustness by applying the proposed compensation techniques entails a tolerable trade-off between acceptance ratio and reliability.

Acknowledgments

This work has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of the priority program "Dependable Embedded Systems" - SPP1500 Get-SURE.

References

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

[2] Sanjoy Baruah and Nathan Fisher. The Partitioned Multiprocessor Scheduling of Sporadic Task Systems. In *Real-Time Systems Symposium*, RTSS '05, pages 321–329, Washington, DC, USA, 2005.

[3] Sanjoy K. Baruah, Alan Burns, and Zhishan Guo. Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In *ECRTS*, 2016.

[4] Riccardo Bettati and Jane W.-S. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *ICDCS*, pages 452–459, 1992.

[5] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[6] Björn B. Brandenburg and Mahircan Gul. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Real-Time Systems Symposium*, 2016.

[7] A. Burns and S. Baruah. *Towards A More Practical Model for Mixed Criticality Systems*, pages 1–6. 2013.

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

[9] Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. Analysis of deadline miss rates for uniprocessor fixed-priority scheduling. In *The 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Hakodate, Japan, 2018.

[10] Kuan-Hsun Chen and Jian-Jia Chen. Probabilistic schedulability tests for uniprocessor fixed-priority scheduling under soft errors. In *12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017, Toulouse, France, June 14-16, 2017*, pages 1–8, 2017.

[11] 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.

[12] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping 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.

[13] Paul Emberson, Roger Stafford, and Robert 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.

[14] Jeremy P. Erickson. Managing tardiness bounds and overload in soft real-time systems, 2014.

[15] Rolf Ernst and Marco Di Natale. Mixed criticality systems - A history of misconceptions? *IEEE Design & Test*, 33(5):65–74, 2016.

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

[17] 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.

[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] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2007.

[20] 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.

[21] 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.

[22] 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.

[23] D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46, Nov 2016.

[24] F. Many and D. Doose. Scheduling analysis under fault bursts. In *RTAS*, pages 113–122, 2011.

[25] Risat Mahmud Pathan. Improving the Quality-of-Service for Scheduling Mixed-Criticality Systems on Multiprocessors. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[26] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, 2007.

[27] Georg von der Brüggen, Jian-Jia Chen, Wen-Hung Huang, and Maolin Yang. Release enforcement in resource-oriented partitioned scheduling for multiprocessor systems. In *International Conference on Real-Time Networks and Systems*, RTNS '17, 2017.

[28] 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.

[29] Georg von der Brüggen, Nico Piatkowski, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. Efficiently approximating the probability of deadline misses in real-time systems. In *2018 30th EUROMICRO Conference on Real-Time Systems (ECRTS)*, 2018.