FT-Feasibility in Fixed Priority Real-Time Scheduling

Hüseyin Aysan, Radu Dobrin, and Sasikumar Punnekkat
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
{huseyin.aysan, radu.dobrin, sasikumar.punnekkat}@mdh.se

Abstract

Real-time systems typically have to satisfy complex requirements mapped to the timing attributes of the tasks that are eventually guaranteed by the underlying scheduler. These systems consist of a mix of hard and soft tasks with varying criticalities as well as associated fault tolerance (FT) requirements. Often time redundancy techniques are preferred in many embedded applications and hence it is extremely important to devise appropriate methodologies for scheduling real-time tasks under fault assumptions. Additionally, the relative criticality of tasks could undergo changes during the evolution of the system. Hence scheduling decisions under fault assumptions have to reflect all these important factors in addition to the resource constraints.

In this paper we propose a framework for 'FT-feasibility', i.e., to provide a priori guarantees that all critical tasks in the system will meet their deadlines even in case of faults. Our main objective here is to ensure FT-feasibility of all critical tasks in the system and do so with minimal costs and without any fundamental changes in the scheduling paradigm. We demonstrate its applicability in scenarios where the FT strategy employed is re-execution of the affected tasks or an alternate action upon occurrence of transient faults or software design faults. We analyse a feasible set of tasks and propose methods to adapt it to varying FT requirements without modifications to the underlying scheduler. We do so by reassigning task attributes to achieve FT-feasibility while keeping the costs minimised.

1. Introduction

Most embedded real-time applications typically have to satisfy complex requirements which are mapped to the timing attributes of the tasks and taken care by the underlying scheduler. These systems are often characterised by high dependability requirements where fault tolerant techniques play a crucial role towards achieving them. Traditionally such systems found in aerospace, avionics or nuclear domains were built with massive replication and redundancy, with the objective to maintain the properties of correctness and timeliness even in the presence of faults. However, in majority of modern embedded applications, due to space, weight and cost considerations it may not be feasible to provide space redundancy. Such systems often have to exploit time redundancy techniques. At the same time, it is imperative that the exploitation of time redundancy does not jeopardize the timeliness requirements on critical tasks.

Real-time scheduling theory has fairly matured over the past two decades to be able to analyze complex and realistic systems. One of its main research streams, viz., fixed priority scheduling has mainly focussed on the provision of feasibility tests which determine if a given task set is schedulable [12, 17, 7, 9]. Such tests have been developed for increasingly less restrictive computational models, thus allowing tasks to interact via shared resources,have arbitrary deadlines, offsets and account for release jitter as well as operating system overheads. However, the designers are still left with many practical issues such as flexibility, fault tolerance guarantees, which are not comprehensively addressed by any single scheduling paradigm. The daunting task of figuring out which combination of analysis techniques and mechanisms are appropriate within an application context often tempts the industry to shy away from reaping the benefits of real-time scheduling theory.

Despite the significant amount of research results, feasibility tests often provide little or no indication of the changes in task attributes required to achieve a feasible system, nor any indication of the extent to which criticalities of the tasks may be changed without causing deadlines to be missed (in the case of a feasible system). In practice, however, it is useful to know both 'what measures are needed' to make them schedulable, as well as 'how' the system can be scheduled to account for the unexpected fault scenarios. As rightly identified in [18], co-development/integration of real-time and fault tolerance dimensions are extremely important especially taking care that upon interaction, their independent protocols do not invalidate the pre-conditions of each other. In this context, our research focus is to develop a framework and analysis techniques which facilitate such guarantees as well as provide valuable information regarding design choices to the designers.

The need for feasibility analysis which provides guarantees for fault-tolerant real-time task sets under the assumption of a defined failure hypothesis is well known in the past. While incorporating fault tolerance into various real-time scheduling paradigms has been addressed by several researchers, the proposed solutions are either ad hoc or applicable only within restrictive contexts. This paper is an attempt to provide such an analysis taking into consideration both the research perspective as well as some of the industrial requirements and practical issues.

The systems we are concerned with typically consist of a mix of hard and soft real-time tasks, where missing deadlines of any hard task could have a large negative impact on the system, while missing the soft tasks occasionally could be still admissible. In such systems, the fault handling has to be performed in a prioritized (due to resource constraints) way depending on the criticalities of the tasks. Also during the evolution/life time of these systems, the relative criticalities of the tasks could undergo changes and the designer might have the tedious task of making new schedules to reflect such changes. This is especially relevant in the case of 'system of systems' or component based systems where the integrator needs to make judicious choices for assigning/fine-tuning the priorities for scheduling the tasks of subsystems within the global context.

We use the term 'FT-feasibility' of a schedule to indicate how good a schedule is with respect to its ability to meet the deadlines of critical tasks upon faults. In this paper, we explore the FT feasibility concept where the FT strategy employed is the re-execution of the affected task, or execution of an alternate task in the event of transient faults. Our approach is to map the fault-induced additional timing requirements at the task instance level to identify potentially interfering high priority task instances and move them optimally to new execution windows. A method to transform off-line schedules to FPS has been proposed in [4], where the authors derive attributes suitable for FPS for sets of off-line scheduled tasks with completely known parameters. In this paper we are focusing on task executions under FPS in the presence of faults which are non-deterministic by their nature. Our analysis provides new task attributes to obtain FT-feasible schedules whilst minimising any related costs. The main element of cost incurred in our methodology is due to the construction of new task artifacts from original task instances in order to satisfy complex priority inequalities.

This concept of FT-feasibility could also be effectively used for selecting most appropriate schedules based on the criticality of a given set of tasks as against the traditional priority-based approaches, which are often too pessimistic.

We analyse a feasible set of tasks and propose methods to adapt it to new FT requirements without necessitating modifications to the underlying scheduler. The proposed methodology is highly applicable during system evolution (where criticalities and priorities could undergo changes), subsystem integration (as in Electronic Control Units (ECUs) in automotive applications) or in legacy applications (where one needs to preserve the original scheduler and scheduling policy). For example, in the case of two ECUs, developed with pre-assigned priorities for tasks from specified priority bands, one may want to fine-tune and get a better schedule considering the global context during integration.

The remainder of the paper is organised as follows. In the next section, we outline a summary of relevant background research from the literature which specifically addresses the issue of scheduling under fault assumptions. Section 3 presents the system characteristics and task model assumed in our paper followed by Section 4 where we specify the fault assumptions and FT strategies used in the analysis. Section 5 describes a simple example describing the nature of the problem along with a feasible solution. Section 6 describes our proposed methodology followed by a running example in section 7. We evaluate the proposed methodology in Section 8 and we conclude the paper in Section 9 where we offer our conclusions and sketche some of the future research dimensions.

2. Related research

Many researchers have addressed incorporating fault tolerance into various real-time scheduling paradigms, some of which are briefly mentioned below.

Liestman and Campbell [10] investigated a fault tolerant scheduling problem where they tried to schedule primary and alternate versions of a task in the same schedule, under the assumption that the task set consists of harmonic periods. Their goal was to maximize the number of primaries scheduled in the system while guaranteeing at least an alternate to be executed. They showed two methods to achieve this. First method was an on-line re-scheduling mechanism that is run after every successful completion of a primary to make use of the time that was allocated to its alternate. In the second method, a tree of schedules is constructed offline and simple decisions for switching between these schedules are made during runtime whenever primaries fail. The on-line computational overhead imposed by the first method and the need of large memory space to store all possible schedules imposed by the second method are the limitations of this approach. Furthermore, both methods provide recovery for the task instances only if the primaries of these task instances are scheduled. However, task criticalities are not taken into account for selecting which primaries to schedule.

Krishna and Shin [8] used a dynamic programming algorithm to embed backup schedules into the primary schedule, so that hard deadlines of critical tasks will be met in the event of up to a specified number of processor failures.

Pandya and Malek [14] showed that single faults with a minimum interarrival time of largest period in the task set can be recovered if the processor utilization is less than 0.5 under Rate Monotonic (RM) scheduling policy. The FT technique they used was re-executing the failed task for recovering from transient hardware or software faults. Though it improved the intuitive utilization bound of 0.345, its applicability is rather restricted to rate monotonic scheduling only.

Ramos-Thuel and Strosnider [16] used Transient Server approach to handle transient errors which arrive as aperiodic recovery requests. They investigated the spare capacity to be given to the server at each priority level in the task set. They also studied the effect of task shedding to the maximum server capacity and task criticality is used for deciding which task to shed. Ghosh et al. [5] presented a method for guaranteeing that the real-time tasks will meet the deadlines under transient faults, by resorting to reserving sufficient slack in queue-based schedules.

Burns et. al. [1] provided exact schedulability tests for fault-tolerant task sets under specified failure hypothesis where the time redundancy is employed in the form of recovery blocks, re-execution of the affected task, checkpointing schemes(refined analysis by Punnekkat et al. [15]) or forward recovery methods like exception handlers. Two important features of these analysis were that it is applicable for any fixed priority scheduling scheme and that being an exact analysis it can guarantee even task sets with higher utilization factors than was possible by Pandya and Malek's test [14]. Subsequently Lima and Burns [11] extended this analysis in case of multiple faults as well as for the case of increasing the priority of a critical task's alternate upon fault occurrences.

Han et al. [6] extended the *last chance strategy* described by Chetto and Chetto [3] for fixed priority driven preemptive scheduling scheme. They assume primary and alternate versions of each task with imprecise computation model, and aim to guarantee either the primary or alternate version of each task to be executed before deadlines, and try to achieve as many primary executions as possible. A fixed priority driven preemptive scheduling scheme is used to reserve times for the alternates by assigning *notification times* for each alternate. This time is the latest time for a primary to complete its execution successfully. Then, on-line part of the algorithm checks if the primaries are successfully executed before notification times. If not, the corresponding alternates are executed. In the opposite case, the reserved time intervals for these alternates are freed by reconstructing the schedule which results in a significant amount of online computational overhead.

Each of the above works have advanced the field of fault tolerant scheduling within the contexts mentioned above. Some of the disadvantages are restrictive task and fault models, non-consideration of mixed criticality task sets, high computational requirements of complex online mechanisms, and scheduler modifications which may be unacceptable from an industrial perspective.

3. System and task model

In this paper we consider only a uniprocessor system, the results are however equally applicable to distributed/multiprocessor systems, where task allocation to individual processors is performed statically. We assume a task set, $\Gamma = \{\tau_1, \tau_2, .., \tau_n\}$, where each task represents a real-time thread of execution. Each task τ_i is assumed to have a minimum inter-arrival time $T(\tau_i)$, meaning that task arrivals may be either periodic or sporadic. Each task τ_i has a known worst case execution time (WCET) $C(\tau_i)$, an earliest start time $est(\tau_i)$ and a deadline $D(\tau_i)$. We assume that $D(\tau_i) \leq T(\tau_i) \quad for \quad i = 1, 2, \ldots, n$. Often the earliest start times are also referred to as offset and represented by $O(\tau_i)$.

Each task can also have an alternate task $\bar{\tau}_i$ with a worst case execution time (WCET) $\bar{C}(\tau_i)$ with a deadline $\bar{D}(\tau_i)$ equal to the original task deadline $D(\tau_i)$. This alternate can typically be a re-execution of the same task, a recovery block, an exception handler or an alternate with imprecise computations.

We assume that each task τ_i has a priority $P(\tau_i)$ assigned to it. The computational model assumed does not impose any restrictions on the priority assignment algorithm used. This could be Rate Monotonic, Deadline Monotonic or any other fixed priority assignment algorithm. We assume that each task is assigned a unique priority and that a task can be immediately preempted by a higher priority task. At run time, the highest priority task from the set of runnable tasks is allocated the processor time. The criticality of a task could be thought of as a measure of the impact of its correct (or incorrect) functioning on the overall system correctness. The priorities and criticalities need not be the same in the strict sense, especially when one employs different scheduling policies. Let Γ_c represent the subset of critical tasks out of the original task set and Γ_{nc} represent the subset of non-critical tasks, so that $\Gamma = \Gamma_c \cup \Gamma_{nc}$. We use $\bar{\Gamma}_c$ to represent the subset of all the alternates of critical tasks. Our framework permits varying criticality levels for tasks, but to simplify the illustration, we use only binary values for criticality in this paper.

We assume that the original task set is schedulable and that the combined utilization of critical tasks is less than or equal to 0.5. The maximum utilization of the system can never exceed 1 at any instant of time including those of the non-critical tasks. In the event of faults, execution of an alternate of a critical task might call for shedding some non-critical tasks during the overload period. We assume that the scheduler has adequate support for flagging non-critical tasks as unschedulable during such scenarios, along with appropriate error detection mechanisms in the operating system.

4. Fault model and fault tolerance strategy

Our primary concern is providing schedulability guarantees to all the critical tasks in fault-tolerant real-time systems which employ temporal redundancy for error recovery. The basic assumption here is that a large variety of transient and intermittent hardware faults can effectively be tolerated by a simple re-execution of the affected task whilst software design faults could be tolerated by executing an alternate action such as recovery blocks or exception handlers. Both of these situations could be considered as execution of another task (either the primary itself or an alternate) with a specified computation time requirement.

We assume that a fault can adversely affect only one task at a time and is detected before the termination of the current execution of the affected task instance. This would naturally include error detection before any context switches due to release of a high priority task. Although somewhat pessimistic, this assumption is realistic since in many implementations, task errors are detected by acceptance tests which are executed at the end of task execution or by watchdog timers that interrupt the task once it has exhausted its budgeted worst case execution time. In case of tasks communicating via shared resources, we assume that an acceptance test is executed before passing an output value to another task to avoid fault propagations and subsequent domino effects.

Our assumption of at most one fault per task instances, is a much harder assumption than what is found in many previous works such as one fault per hyper period (LCM) or an explicit minimum inter-arrival time requirement between consecutive fault occurrences.

One can envisage many possible variations to the fault model and fault tolerance strategies. Though the present work does not categorically mention each of them, our framework is designed in such a way as to accommodate future anticipated changes in the fault model, fault tolerance strategies as well as probabilistic guarantees [2].

5 Motivating example

Let our task set consists of 2 tasks, A and B, where $T(A) = 3$, $T(B) = 6$, $C(A) = 2$ and $C(B) = 2$, scheduled according to the RM policy (Figure 1) where B is the critical task subject to failures. We also assume that a simple re-execution of the affected task is the fault tolerance strategy.

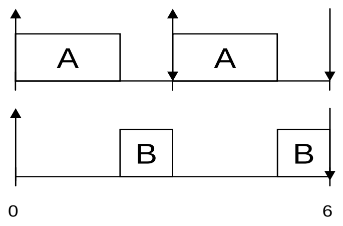

Figure 1. Original task set

To be able to re-execute B upon a fault occurrence, B must complete before $D(B) - C(B)$. In this case, B's new deadline will be 4. One possibility is to assign B a higher priority than A. However, the new priority ordering between A and B will lead to a deadline miss on the first instance of A even if no fault occurs during B's execution (Figure 2).

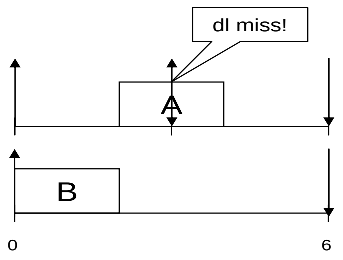

Figure 2. B fault tolerant - A misses deadline

In our approach we propose transforming A's former instances over LCM into 2 artifacts, i.e., A1 and A2. In this case, A1 will be assigned the highest priority followed by B, and finally A2 with the lowest priority. Additionally, we reassign offsets and deadlines to the artifact tasks to ensure the executions within their original feasibility windows (Figure 3).

This new set of attributes will guarantee the task completions before their deadlines in the absence of faults. On the other hand, if a fault occurs during B's execution, B will still have time to re-execute before its deadline at the expense of a possible deadline miss on A2. However, the assumption of fault occurrence and detection at the end of B's execution is highly pessimistic, as well as at run-time the task executions will be most likely shorter than the assumed WCET. In this case, both the re-execution of B and the execution of A2 will most likely meet their deadlines (Figure 4).

The new FT-schedulable task set is presented in table 1.

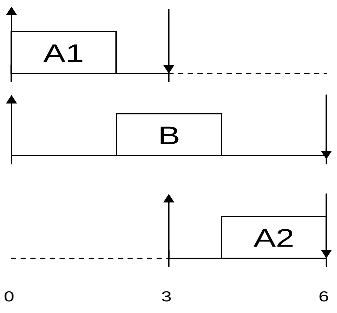

Figure 3. B fault tolerant - no deadline misses

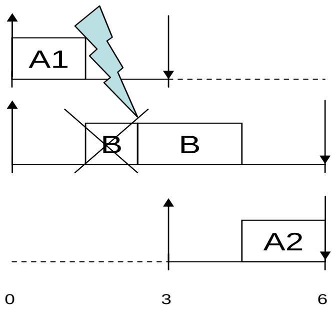

Figure 4. B re-executes - no deadline misses

The associated cost towards achieving FT-schedulability in this example is an increase in the number of tasks by one.

6. Our methodology

We assume a periodic task set schedulable by standard FPS. Each task instance τ_i^j has an original earliest start time $est(\tau_i^j)$ and deadline $D(\tau_i^j)$.

6.1 Overview

We aim to find new deadlines for each critical and non-critical task, such that each critical task instance can be re-executed before its original deadline upon a fault occurrence while the resource allocation to non-critical tasks is maximized, i.e., non-critical tasks execute at the highest possible priority that will not jeopardize the FT feasibility of the critical ones.

Task	T	C	O	D	P
A1	6	2	0	3	3(highest)
A2	6	2	3	6	1
B	6	3	0	6	2

Table 1. FT-feasible FPS tasks

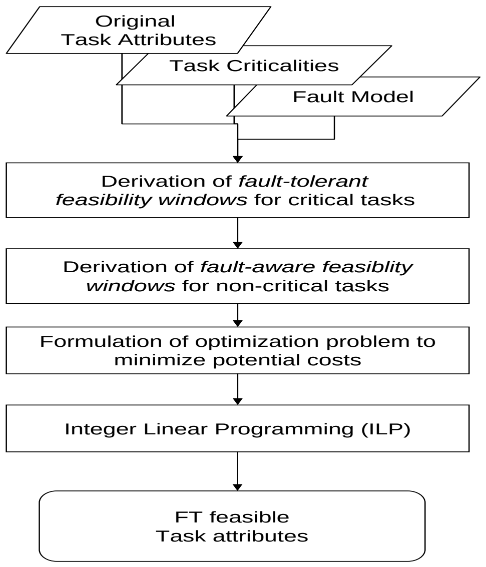
Original Task Attributes
Task Criticalities
Fault Model
Derivation of fault-tolerant feasibility windows for critical tasks
Derivation of fault-aware feasiblity windows for non-critical tasks
Formulation of optimization problem to minimize potential costs
Integer Linear Programming (ILP)
FT feasible Task attributes

Figure 5. Overview of methodology

The major steps of the proposed methodology are shown in Figure 5 and explained below:

1. Input: Original task set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, schedulable by FPS, with earliest start times and deadlines
2. Calculate fault tolerant (FT) deadlines for each critical task instance
   (a) Select subset of critical tasks Γ_c out of the original task set
   (b) Calculate latest start times, $lst(\tau_i^j)$, for each critical task instance τ_i^j under EDF scheduling
   (c) Calculate new FT-deadlines for each critical task instance, $D_{FT}(\tau_i^j)$, based on the latest start times previously derived
3. Calculate fault aware (FA) deadlines for each non-critical task instannce
   (a) Select subset of non-critical tasks Γ_{nc} out of the original task set
   (b) Calculate new FA deadlines for each non-critical task instance, $D_{FA}(\tau_i^j)$, based on the previously derived FT feasibility windows of critical task instances
4. Output: FT and FA feasibility windows,

$$FT\_FW(\tau_i^j) = [est(\tau_i^j), D_{FT}(\tau_i^j)],\ if\ \tau_i^j \in \Gamma_c$$

and

$$FT\_FA(\tau_i^j) = [est(\tau_i^j), D_{FA}(\tau_i^j)],\ if\ \tau_i^j \in \Gamma_{nc}$$

for each task instance $\tau_i^j \in \Gamma$

6.2 Proposed approach

Our goal is to derive feasibility windows for each tasks instance $\tau_i^j \in \Gamma$ to guarantee fault tolerance for the critical tasks. However, depending on the criticality of the original tasks, the feasibility windows we are looking for differ as:

1. fault tolerant (FT) feasibility windows for critical tasks and
2. fault aware (FA) feasibility windows for non-critical ones.

At the same time, we want to maximize the size of the feasibility windows to maximize the flexibility of the tasks. The size of the FA feasibility windows of the non-critical tasks is highly dependent on the size of the FT feasibility windows of the critical ones. Hence, we first calculate FT feasibility windows for the critical tasks, and then, the FA feasibility windows for the non-critical ones.

Derivation of FT deadlines: We consider the subset $\Gamma_c \in \Gamma$ and assume a single criticality level for all critical tasks. We assume $C(\tau_i) = 2WCET(\tau_i)$ for each critical task to account for re-execution upon a fault occurrence per task instance. First, we schedule the set of critical tasks by EDF and derive the completion times of each instance, $finish\_time(\tau_i^j)$. We use EDF to exploit its ability to guarantee task schedulability up to 100% processor utilization. Then, "by mirroring" the EDF schedule, we obtain the latest start times for each task instance such that it can feasibly execute 2 times its WCET before its original deadline. In other words:

$$lst(\tau_i^j) = LCM - finish\_time(\tau_i^k)$$

where

$$k = \frac{LCM}{T(\tau_i)} + j - 1$$

At this point we want to find the *new deadlines*, $D_{FT}(\tau_i^j)$, for the critical tasks instances such that each can be re-executed before its original deadline. For a single task instance without any interference from other tasks, the new deadline is $D_{FT}(\tau_i^j) = lst(\tau_i^j) + C(\tau_i^j)$. However, when calculating the new deadlines, we have to take into account potential interference from other critical tasks, i.e., if the task is preempted after its latest start time.

$\forall\ \tau_i^j, \tau_k^l \in\ \Gamma_c$

$$D_{FT}(\tau_i^j) = D(\tau_i^j) - C(\tau_i^j) - \sum_{k,l=1}^{n} interference(\tau_k^l)$$

where

$$\forall k,l,\ est(\tau_i^j) < lst(\tau_k^l) \leq D(\tau_i^j)$$

and

$$interference(\tau_k^l) = min[2C(\tau_k^l), (D(\tau_i^j) - lst(\tau_k^l))]$$

The FT feasibility windows for each critical task instance are defined as: $\forall \tau_i^j \in \Gamma_c$,

$$FT\_FW(\tau_i^j) = [est(\tau_i^j), D_{FT}(\tau_i^j)]$$

Additionally, we denote the latest finishing time of the alternate task as $\overline{D}_{FT}(\tau_i^j)$.

Derivation of FA deadlines: The underlaying FT mechanism will shed non-critical tasks upon re-executions of critical ones if the non-critical tasks will miss their deadlines. However, the original non-critical task deadlines are not fault-aware in the presence of critical tasks. Hence, a non-critical task execution can be delayed by a critical task re-execution, potentially causing another critical task to miss its deadline.

We aim to provide new fault-aware deadlines to non-critical tasks to protect critical ones from being hit. We attempt to derive new deadlines for each non-critical task instance in order to restrict them from interfering with the critical tasks in case of critical task failure. As a part of recovery action upon failure, the underlaying fault tolerant on-line mechanism checks if there is enough time left for the non-critical task instances to complete before their new deadlines. If not, these instances are not executed.

Here, we can not schedule the tasks in reverse order according to EDF as previously, by simply mirroring the schedule constructed by EDF, since the critical tasks may not have deadlines equal to the end of the periods due to the previously described deadline assignment procedure. Hence, we can no longer benefit form the full processor utilization guarantee provided by EDF.

In our method, we schedule the non-critical tasks by reversed EDF in the remaining slack after the critical tasks are scheduled to execute as late as possible. In some cases, we may fail finding valid FA deadlines on some non-critical task instances. We say that a FA deadline, $D_{FA}(\tau_i^j)$, is *not valid* if $D_{FA}(\tau_i^j) - est(\tau_i^j) < C(\tau_i^j)$. In these cases, we keep the original deadline, and we assign the non-critical task a background priority, i.e., a lower priority than any other critical task, and any other non-critical task with a *valid FA deadline*.

6.3 FPS attribute assignment

Next, we derive FPS attributes such that the original scheduler can be used without further modifications while the critical tasks can feasibly be re-executed or an alternate action can be run upon a fault occurrence.

To do so, we analyze the task set with new deadlines and identify priority relations for each point in time t_k at which at least one task instance is released. We create priority inequalities between instances to reflect the order of execution under EDF. Then we produce a task set with attributes for FPS. As, typically, EDF schedules cannot be translated directly to attributes for FPS, we may have to create new tasks (*artifacts*) of former task instances. The resulting number of FPS tasks is to be minimized.

At this point, our task model, essentially, consists of four types of task instances: critical task instances, Γ_c consisting of primaries Γ_c^{pri} and alternates Γ_c^{alt}, and non-critical task instances with and without valid FA deadlines, $\Gamma_{nc} = \Gamma_{nc}^{FA} \cup \Gamma_{nc}^{non\_FA}$.

Every $t_k \in [0, LCM)$ such that t_k equals the release time of at least one task, we consider a subset $\Gamma_{t_k} \subseteq \Gamma$ consisting of:

1. $\{current\_instances\}_{t_k}$ - instances τ_i^j of tasks τ_i, released at the time t_k: $est(\tau_i^j) = t_k$
2. $\{interfering\_instances\}_{t_k}$ - instances τ_s^q of task τ_s released before t_k but potentially executing after t_k.

$$est(\tau_s^q) < t_k < D(\tau_s^q)$$

where

$$D(\tau_s^q) = \begin{cases} D_{FT}(\tau_s^q), & if\ \tau_s^q \in \Gamma_c^{pri} \\ \overline{D}_{FT}(\tau_s^q), & if\ \tau_s^q \in \Gamma_c^{alt} \\ D_{FA}(\tau_s^q), & if\ \tau_s^q \in \Gamma_{nc}^{FA} \\ D(\tau_s^q), & if\ \tau_s^q \in \Gamma_{nc}^{non\_FA} \end{cases}$$

We derive priority relations within each subset Γ_{t_k}:
$\forall t_k, \forall \tau_i^j, \tau_s^q \in \Gamma_{t_k}$, where $i \neq s$.

1. if $\tau_i^j, \tau_s^q \in \Gamma_c \cup \Gamma_{nc}^{FA}$, or if $\tau_i^j, \tau_s^q \in \Gamma_{nc}^{non\_FA}$

$$P(\tau_i^j) > P(\tau_s^q),\ where\ D(\tau_i^j) < D(\tau_s^q)$$

2. if $\tau_i^j \in \Gamma_c \cup \Gamma_{nc}^{FA}$ and $\tau_s^q \in \Gamma_{nc}^{non\_FA}$

$$P(\tau_i^j) > P(\tau_s^q)$$

In tie situations, e.g., when the instances τ_i^j and τ_s^q have same deadlines, we prioritize the one with the earliest start times. In case even the earliest start times are equal, we derive the priority inequalities consistently.

Our goal is to provide tasks with fixed offsets and fixed priorities. When we solve the priority inequalities derived so far, it may happen that we have to assign different priorities to different instances of the same task, in order to reenact the EDF schedule. These cases cannot be expressed directly with fixed priorities and are the sources for *priority assignment conflicts*. We solve this issue by splitting the task with the inconsistent priority assignment into a number of new periodic tasks with different priorities. The instances of the new tasks comprise all instances of the original task. Our goal is to find the splits which yield the smallest number of FPS tasks.

An optimization issue is which task to split to yield the least number of artifacts. Since a priority assignment conflict involves at least two different tasks, there is typically a choice of which task to split. For example, if two instances of two tasks, e.g., A and B are involved in a priority conflict, we can split either A or B into their instances to resolve the conflict. For instance, if A has 4 instances over LCM and B 6 instances, the local optimum would be to split A resulting in creating 3 artifacts. However, due to a later conflicting situation, B may have to be split anyway leading to additional 5 artifacts, resulting in 8 total. Hence, the optimum solution would have been to split B at the first conflicting situation to minimize the total number of artifacts.

Hence, in order to minimize the number of artifact tasks, we create an integer linear programming problem from the derived system of priority inequalities to first identify which instances to split, if any, and to derive priorities for the resulting FPS tasks. The flexibility of the ILP solver allows for simple inclusion of other criteria via goal functions.

6.4 ILP problem representation

A linear programming (LP) problem consists of a linear goal function in a number of variables and a set of linear inequality relations of the variables. LP solving searches a value assignment for all variables (solution) that optimizes (minimizes or maximizes) the given goal function under the given constraints. If the values of a solution have to be integral the problem is called an integer linear programming (ILP) problem.

The aim of the given attribute assignment problem is to find a task set, i.e., a minimum number of tasks together with their priorities, that fulfills the priority relations of the sequences of the schedule. As mentioned above, each task of the task set is either one of the original tasks or an artifact task created from one of the instances of an original task selected for splitting.

The problem is translated into an ILP problem, because we are only interested in integral priority assignments and solutions. In the ILP problem the goal function G to be minimized computes the number of tasks to be used in the

FPS scheduler

$$G = N + \sum_{i=1}^{N}(k_i - 1) * b_i + \sum_{i=1}^{N}\sum_{j=1}^{n} b_i^j a$$

here N is the number of original tasks, n is the number of instances of τ_i over LCM, k_i is the number of instances of task τ_i, b_i is a binary integral variable that indicates if τ_i needs to be split into its instances and $b_i^j a$ is a binary variable that indicates if the alternate of the critical task instance τ_i^j can be executed at the same priority as it primary.

The constraints of the ILP problem reflect the restrictions on the task priorities as imposed by scheduling problem. To account for the case of priority conflicts, i.e., when tasks have to be split, the constraints between the original tasks, including task re-executions, are extended to include the constraints of the artifact tasks. Thus each priority relation $P(\tau_i^j) > P(\tau_p^q)$ between two tasks is translated into an ILP constraint:

$$p_i + p_i^j > p_p + p_p^q,$$

where the variables p_i and p_p stand for the priorities of the FPS tasks representing the original tasks or alternates τ_i and τ_p, respectively, and p_i^j, p_p^q stand for the priorities of the artifact tasks τ_i^j and τ_p^q (in case it is necessary to split the original tasks or to run an alternate at a different priority). Although this may look like a constraint between four tasks (τ_i, τ_i^j, τ_p, τ_p^q) it is in fact a constraint between two tasks – for each task only its original (τ_i resp. τ_p) or its artifact tasks (τ_i^j resp. τ_p^q) can exist in the FPS schedule. In case the priority relation involves task re-executions, e g., $P(\tau_i^j a) > P(\tau_p^q)$ the resulting constraint is:

$$p_i^j a > p_p + p_p^q,$$

where $\tau_i^j a$ represents the alternate execution of τ_i^j. Our goal is to be able to re-execute a task instance without changing its priority.

A further set of constraints for each task τ_i ensure that only either the original tasks or its instances (artifact) are assigned valid priorities (greater than 0) by the ILP solver. All other priorities are set to zero.

$$\begin{array}{rcl} p_i & \leq & (1 - b_i) * M \\ \forall j : p_i^j & \leq & b_i * M \end{array}$$

On the other hand, both primaries and alternates can coexist at different valid priorities. Moreover, a primary can be an original task instance or an artifact. The last set of constraints aims to yield same priorities for both of them. Otherwise, the alternate will be assigned a different priority than its primary.

$$\left|(p_i + p_i^j) - p_i^j a\right| \leq b_i^j a * M$$

In these constraints M is a large number, larger than the total number of instances and alternates in the original task set. The variable b_i for task τ_i, which also occurs in the goal function, is the binary variables that indicates if τ_i has to be split, i.e., b_i allows only a task or its artifact tasks to assume valid priorities. On the other hand, the variable $b_i^j a$ ia a binary variable that indicates if the alternate of τ_i^j can be scheduled at the same priority as its primary. Since the goal function associates a penalty for each b_i and $b_i^j a$ that has to be set to 1, the ILP problem indeed searches for a solution that produces a minimum number of task splits. The constraints on the binary variables complete the ILP constraints:

$$\forall i, j : \; b_i, b_i^j a \leq 1$$

The solution of the ILP problem yields the total number of tasks as the result of the goal function. The values of the variables represent a priority assignment for tasks and artifact tasks that satisfies the priority relations of the scheduling problem.

6.5 Periods and offsets

Since the priorities of the FP tasks have been assigned by the LP-solver, we can now focus on the assignment of periods and offsets. Now we have a set of tasks with FPS attributes, Γ_{FPS}. Based on the information provided by the LP-solver, we assign periods and offsets to each task $\tau_i \in \Gamma_{FPS}$, in order to ensure the run time execution under FPS within their respective FT feasibility windows, as following:

$$\begin{array}{l} for \quad 1 \leq i \leq nr\_of\_tasks\_in \; \Gamma_{FPS} \\ \qquad T(\tau_i) = \dfrac{LCM}{nr\_of\_instances(\tau_i)} \\ \qquad O(\tau_i) = est(\tau_i^1)) \end{array}$$

7. Example

We illustrate our method by an example. Let us assume we have a task set schedulable by RM as described in table 2 and Figure 6.

Let us now assume B and C are the critical tasks. In this example, RM priority assignment can not guarantee fault tolerance on every critical task instance, e.g., if the first 2 instances of B are hit by a fault and have to be re-executed, C

Task	T	C
A	3	1
B	4	1
C	12	3

Table 2. Original task set

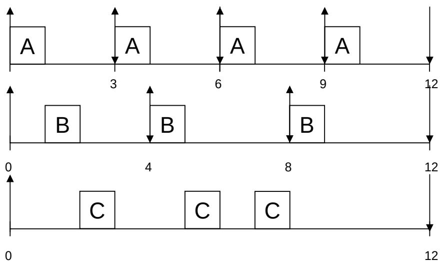

Figure 6. Original RM schedule

will not be able to re-execute without missing its deadline if a fault occurs during its execution. In our method we derive FPS attributes to guarantee fault tolerance on each critical task instance by first deriving FT feasibility windows for the critical tasks. We do so by first scheduling the critical tasks by EDF "in the mirror" (Figure 7). The dashed blocks represent the re-execution of the critical tasks instances upon a fault.

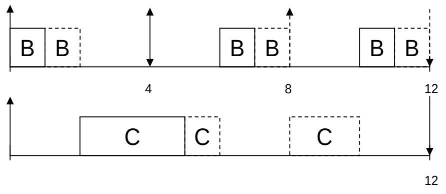

Figure 7. EDF schedule for critical tasks and re-executions

The FT feasibility windows for the critical tasks are presented in Figure 8.

At this point we derive FA feasibility windows for non-critical task instances (in our case, for the instances of A), by schedule them by "ED as late as possible" together with the critical ones (Figure 9). Based on the derived FT and FA feasibility windows for the critical and non-critical tasks respectively, we analyze the sets of current and interfering instances for each release time in the tasks set and we derive priority relations between the instances as described in Section 6.3. The resulting priority inequalities are presented in Figure 3.

Next, we formulate the optimization problem as described in section 6.4. The LP solver provides us a set of fault tolerant tasks suitable for FPS to which we assign periods and offsets as described in section 6.5. The resulting task set is presented in table 4.

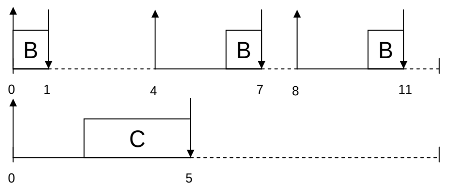

Figure 8. FT feasibility windows for B and C

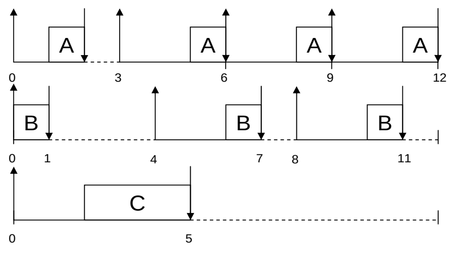

Figure 9. FA feasibility windows for A

In our example, since the utilization is already 100% without any faults, the LP solver yields a solution consisting of 9 tasks, i.e., 8 from the original tasks instances, and one additional consisting of the alternate task belonging to C that has to be executed at a lower priority than C.

The resulting task set is directly schedulable by the original scheduler while the critical tasks can tolerate one fault per instance. Moreover, non-critical tasks can be scheduled up to full processor utilization in the absence of faults. In case of a fault, however, the non-critical tasks will be suspended by the underlaying scheduler until the faulty task has been re-executed.

8. Evaluation

In real-time systems where both critical and non-critical tasks co-exist, missing a single deadline of a critical task instance can result in more severe consequences than missing several deadlines of non-critical task instances. Based on this point of view, we define our primary success criteria as the percentage of successfully met critical deadlines in our evaluation. Meeting the deadlines of non-critical task instances is assumed to be the secondary success criteria and amount of deadline misses of such tasks can be seen as the cost of meeting more critical deadlines.

In this section we evaluate the performance of our method in comparison with rate monotonic (RM) schedul-

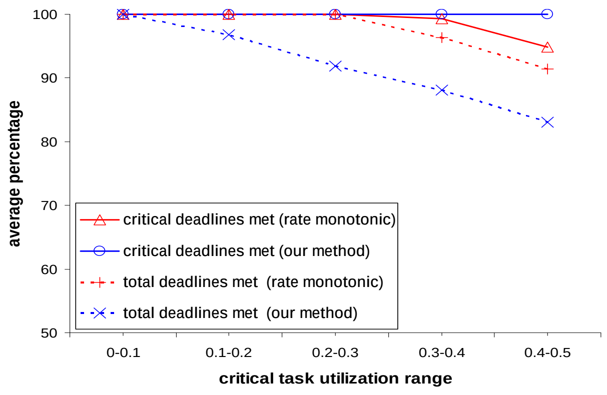

Figure 10. Average percentage of successfully met deadlines (Total utilization is between 0.6 and 0.7)

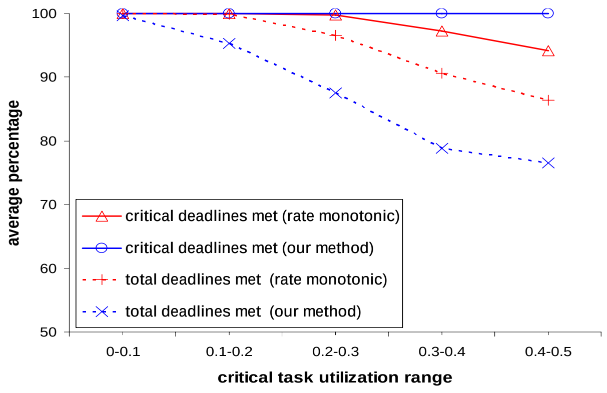

Figure 11. Average percentage of successfully met deadlines (Total utilization is between 0.7 and 0.8)

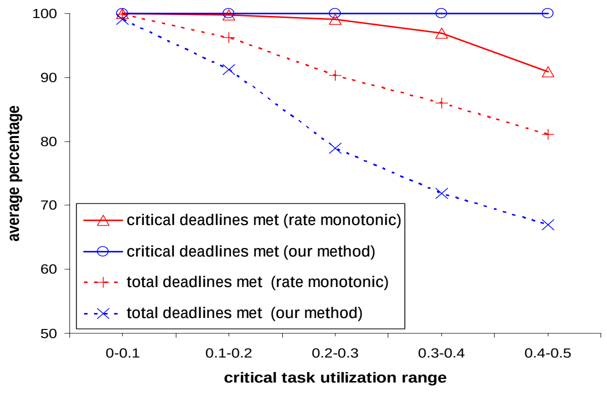

Figure 12. Average percentage of successfully met deadlines (Total utilization is between 0.8 and 0.9)

t_k	$\left\{ \begin{matrix} current \\ inst. \end{matrix} \right\}_{t_k}$	$\left\{ \begin{matrix} intf. \\ inst. \end{matrix} \right\}_{t_k}$	inequalities
0	$A^1, B^1, \overline{B}^1, C^1, \overline{C}^1$	None	$P(B^1) > P(A^1)$ $P(\overline{B}^1) > P(C^1)$ $P(A^1) > P(C^1)$
3	A^2	C^1	$P(C^1) > P(A^2)$
4	$B^2, \overline{B}^2$	$A^2, C^1, \overline{C}^1$	$P(C^1) > P(A^2)$ $P(A^2) > P(B^2)$ $P(B^2) > P(\overline{C}^1)$ $P(\overline{B}^2) > P(\overline{C}^1)$
6	A^3	$B^2, \overline{B}^2\overline{C}^1$	$P(B^2) > P(A^3)$ $P(B^2) > P(\overline{C}^1)$ $P(\overline{B}^2) > P(\overline{C}^1)$
8	$B^3, \overline{B}^3$	$A^3, \overline{C}^1$	$P(A^3) > P(B^3)$ $P(\overline{C}^1) > P(B^3)$ $P(\overline{C}^1) > P(\overline{B}^3)$
9	A^4	$B^3, \overline{B}^3, \overline{C}^1$	$P(B^3) > P(A^4)$ $P(\overline{C^1}) > P(B^3)$ $P(\overline{C^1}) > P(\overline{B}^3)$

Table 3. Derivation of inequalities

$\overline{\tau_i}$	T	C	O	D	P
A1	12	1	0	3	7
A2	12	1	3	6	5
A3	12	1	3	9	2
A4	12	1	9	12	0
B1	12	1	0	1	8 (highest)
B2	12	1	4	7	4
B3	12	1	8	11	1
C	12	3	0	5	6
$\overline{C}$	12	3	0	10	3

Table 4. FT FPS Tasks

ing policy upon occurrence of faults. We conducted a number of simulations on synthetic task sets as the lack of a priori knowledge about when the faults occur and the resulting task interactions make the comparison procedure rather complex to be performed mathematically. We simulated the worst case scenario where every critical task instance is hit by a fault at the end of its execution and then re-executed.

2000 task sets were generated where the total number of tasks in every task set is 10 and the number of critical tasks is varying randomly from 1 to 10. The LCM is chosen randomly between 20 and 200 time units which seems sufficient to compare the behavior of two approaches. One reason of choosing the constant value 10 for the number of tasks is related to the LCM range. With this constant value and the given LCM range, we are able to create tasks sets with a wide range of total utilization from 0.5 to 1 even when the LCM is selected as minimum. Furthermore, the limited number of tasks increases the traceability of the scheduling decisions made by the approaches under obser-

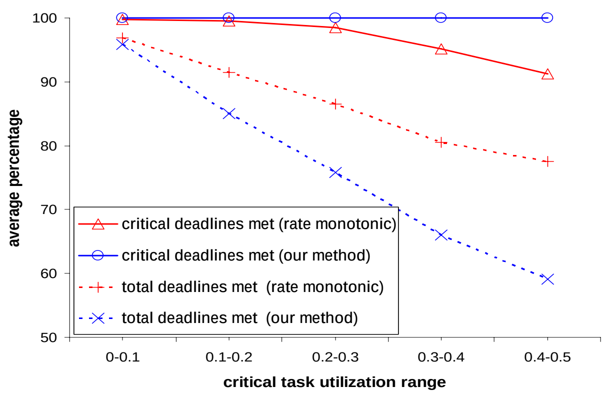

Figure 13. Average percentage of successfully met deadlines (Total utilization is between 0.9 and 1)

vation. After finding the LCM, task periods are randomly chosen among the divisors of LCM. Randomization is realized by Mersenne Twister pseudorandom number generator with 32-bit word length [13]. Total processor utilizations of the task sets were kept within intervals of 0.1 for every group of 500 task sets starting from the range 0.6-0.7. Within each group, processor utilizations of the critical tasks were also kept within intervals of 0.1 for every subgroup of 100 task sets varying between the range 0-0.1 and 0.4-0.5. The average execution time of our implementation to create FT feasible task attributes is around 100 milliseconds on a 1GHz PC when a task set generated as described above is used as an input.

Figures 10 to 13 show the average percentage of successfully met deadlines with respect to critical task utilization. Each figure shows a different range of total CPU utilization starting from the range 0.6-0.7. As the CPU utilization increases, it can be seen that the success of our method also increases with the cost of missing more non-critical deadlines.

In the processor utilization range 0.6-0.7, our method starts to give better results than RM when critical task utilization is above 0.3 (Figure 10). In the range 0.8-0.9 this threshold decreases to 0.2 (Figure 12). When the processor utilization is between 0.9 and 1 (Figure 13), critical task instances scheduled by RM start to miss their deadlines even the critical task utilization very low while our method still guarantees meeting all the critical deadlines.

9. Conclusions and future work

We have presented a framework which allows the system designer to schedule a set of real-time tasks with mixed criticalities and fault tolerance requirements. Our main contribution is the methodology which eliminates shortcomings of the earlier works and schedules tasks of mixed criticalities and their alternates with a performance level equivalent to online algorithms like EDF, but in an fixed priority-based system incurring much less overheads. The proposed method can guarantee that all the critical tasks (primaries or alternates upon faults) will meet their deadlines provided their combined utilization is less than 1. Additionally our methodology can schedule the non critical tasks in a fault-aware manner to achieve the best possible utilization of the system.

Our ongoing work attempts to incorporate more complex fault models and provision of probabilistic guarantees [2] to non critical tasks. Another research direction is to map the related works in our framework to compare and formally prove that this framework is capable of performing at least as good as all of them. We are also working on formalizing an FT-feasibility index which can distinguish different schedules in terms of feasibility and associated costs to help the designer in choosing the optimal schedule.

References

[1] A. Burns, R. I. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. *Euromicro Real-Time Systems Workshop*, pages 29–33, June 1996.

[2] A. Burns, S. Punnekkat, L. Strigini, and D. Wright. Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems. In *Proceedings of DCCS-7,IFIP International Conference on Dependable Computing for Critical Applications, California*, January 1999.

[3] H. Chetto and M.Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[4] R. Dobrin, G. Fohler, and P. Puschner. Translating offline schedules into task attributes for fixed priority scheduling. In *Proc. 22nd IEEE Real-Time Systems Symposium*, pages 225–234, Dec. 2001.

[5] S. Ghosh, R. Melhem, and D. Mosse. Enhancing real-time schedules to tolerate transient faults. *Proceedings Real-Time Systems Symposium*, December 1995.

[6] C.-C. Han, K. G. Shin, and J. Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEEE Trans. Computers*, 52(3):362–372, 2003.

[7] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal - British Computer Society*, 29(5):390–395, October 1986.

[8] C. Krishna and K. Shin. On scheduling tasks with a quick recovery from failure. *IEEE Transactions on Computers*, 35(5):448–455, May 1986.

[9] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm - Exact characterization and average case behaviour. *Proceedings of IEEE Real-Time Systems Symposium*, pages 166,171, December 1989.

[10] A. L. Liestman and R. H. Campbell. A Fault-Tolerant Scheduling Problem. *IEEE Transactions on Software Engineering*, 12(11):1089–95, November 1986.

[11] G. Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *IEEE Transactions on Computers*, 52(10):1332–1346, October 2003.

[12] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):40–61, 1973.

[13] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[14] M. Pandya and M. Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Trans. on Computers*, 47(10), 1998.

[15] S. Punnekkat, A. Burns, and R. I. Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems*, 20(1):83–102, 2001.

[16] S. Ramos-Thuel and J. Strosnider. The transient server approach to scheduling time-critical recovery operations. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 286–295, December 4-6 1991.

[17] O. Serlin. Scheduling of Time Critical Processes. *Proceedings AFIPS Spring Computing Conference*, pages 925–932, 1972.

[18] L. Sha, T. F. Abdelzaher, K.-E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.