Maximizing the Fault Tolerance Capability of Fixed Priority Schedules[*]

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

Abstract

Real-time systems typically have to satisfy complex requirements, mapped to the task attributes, eventually guaranteed by the underlying scheduler. These systems consist of a mix of hard and soft tasks with varying criticality, as well as associated fault tolerance requirements. Additionally, the relative criticality of tasks could undergo changes during the system evolution. Time redundancy techniques are often preferred in embedded applications and, hence, it is extremely important to devise appropriate methodologies for scheduling real-time tasks under failure assumptions.

In this paper, we propose a methodology to provide a priori guarantees in fixed priority scheduling (FPS) such that the system will be able to tolerate one error per every critical task instance. We do so by using Integer Linear Programming (ILP) to derive task attributes that guarantee re-execution of every critical task instance before its deadline, while keeping the associated costs minimized. We illustrate the effectiveness of our approach, in comparison with fault tolerant (FT) adaptations of the well-known rate monotonic (RM) scheduling, by simulations.

1 Introduction

Most embedded real-time applications typically have to satisfy complex requirements, mapped to task attributes and further used by the underlying scheduler in the scheduling decision. These systems are often characterized by high dependability requirements, where fault tolerance techniques play a crucial role towards achieving them. Traditionally, such systems found in, e.g., aerospace, avionics or nuclear domains, were built with high replication and redundancy, with the objective to maintain the properties of correctness and timeliness even under error occurrences. 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, and in particular fixed priority scheduling (FPS), has fairly matured over the past two decades to be able to analyze complex and realistic systems [14, 20, 7, 9]. However, the designers are still left with many practical issues, such as flexibility or fault tolerance guarantees, which are not comprehensively addressed by any single scheduling paradigm. As rightly identified in [21], 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.

Incorporating fault tolerance into various real-time scheduling paradigms has been addressed by several researchers. In [11] and [6], different approaches are presented to schedule primary and alternate versions of tasks to provide fault tolerance. Krishna and Shin [8] used a dynamic programming algorithm to embed backup schedules into the primary schedule. Ramos-Thuel and Strosnider [18] used the Transient Server approach to handle transient errors and investigated the spare capacity to be given to the server at each priority level. They also studied the effect of task shedding to the maximum server capacity where task criticality is used for deciding which task to shed. In [5, 10], the authors 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. Pandya and Malek [16] showed that single faults with a minimum inter-arrival time of largest period in the task set can be recovered if the processor utilization is less than or equal to 0.5 under rate monotonic (RM) scheduling. Burns et. al. [1, 17, 2] provided exact schedulability tests for fault tolerant task sets under specified failure hypothesis. These analysis are applicable for FPS schemes, and, being exact analysis, can guarantee task sets with even higher utilization than guaranteed by Pandya and Malek's test [16]. Lima and Burns [12, 13] extended this analysis in case of multiple faults, as well as for the

[*]This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

case of increasing the priority of a critical task's alternate upon fault occurrences, and in [19] an upper bound for fault-tolerance in real-time systems based on slack redistribution is presented. While the above works have advanced the field of fault tolerant scheduling within specified contexts, each one has some shortcomings, e.g., restrictive task and fault models, non-consideration of task criticality, high computational requirements of complex on-line mechanisms, and scheduler modifications which may be unacceptable from an industrial perspective.

Unlike many previous works, our method guarantees *all primaries' and all alternates' feasible execution, up to 100% utilization, in FPS*, without any on-line computational overhead or major modifications to the underlying scheduler. By doing so, we can successfully recover even in situations where errors occur at the end of the primary task executions. Furthermore, we are able to provide guarantees in worse error scenarios, e.g., assuming *one error per task instance*, as compared to earlier assumptions such as one error per longest task period. Additionally, in case the system load permits, non-critical tasks can feasibly co-exist with critical ones at high priority levels. Our approach targets systems consisting of a mix of hard and soft real-time tasks, where missing a hard task deadline could have a large negative impact on the system, while missing soft task deadlines could be occasionally admissible. In such systems, the error recovery has to be performed in a prioritized (due to resource constraints) way, depending on the task criticalities. Moreover, as the relative task criticalities could undergo changes during the evolution/life time of these systems, 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 task priority assigning/fine-tuning for the subsystem scheduling within the global context.

In our approach, we use the term 'FT-feasibility' of a schedule to indicate whether it is guaranteed to meet the critical task deadlines under specified error assumptions. We assume that the fault tolerance strategy employed is the re-execution of the affected tasks, or execution of alternate tasks in the event of errors. We analyze the error-induced additional timing requirements at the task instance level and derive appropriate task execution windows satisfying these requirements. Based on these windows, and using ILP, we calculate FPS attributes to obtain FT-feasible schedules. In some cases, e.g., when the fault tolerance requirements can not be expressed directly by FPS attributes, we introduce artifacts by splitting tasks into instances to obtain a new task set with consistent FPS attributes. The number of artifacts is bounded by the total number of instances in the schedule within the hyperperiod (LCM). Our method is guaranteed to find a solution, i.e., FT-feasible FPS attributes, under given assumptions, and is optimal in the sense that it minimizes the number of artifacts, which is the main element of cost. In cases the cost may be found too high, e.g., due to extremely large task sets, the proposed methodology allows the end user to selectively choose between the level of FT-feasibility and the number of artifacts. This concept of FT-feasibility could also be effectively used for selecting most appropriate schedules based on the criticality of a given task set, as against the traditional priority-based approaches, which are often too pessimistic. In [4], a method was presented to translate off-line schedules to FPS attributes, assuming the existence of feasibility windows for task instances. In this paper, we derive the FT feasibility windows of the tasks and target FP-based systems directly.

Our methodology is highly applicable in safety critical RT systems design, in legacy applications (where one needs to preserve the original scheduler and scheduling policy), during system evolution (where criticalities and priorities could undergo changes), or during subsystem integration (as in embedded software present in Electronic Control Units) in automotive applications. 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. One can envisage many possible variations to the error model and fault tolerance strategies. Though the present work does not categorically mention each of them, our method is designed in such a way as to accommodate future anticipated changes in the error model and fault tolerance strategies.

The remainder of the paper is organized as follows. In the next section, we present the system characteristics, task model and error scenarios assumed in this paper, together with the FT strategy used in our analysis. Section 3 describes our proposed methodology, illustrated by an example in Section 4. We present evaluation results in Section 5 and conclude the paper in Section 6.

2 System and task model

We assume a periodic task set, $\Gamma = \{\tau_1, \ldots, \tau_n\}$, where each task represents a real-time thread of execution. Each task τ_i has a period $T(\tau_i)$ and a known worst case execution time (WCET) $C(\tau_i)$. We assume that the tasks have deadlines ($D(\tau_i)$) equal to their periods. The task set Γ consists of critical and non-critical tasks where the task criticality could be seen as a measure of the impact of its correct (or incorrect) functioning on the overall system correctness. Each critical task τ_i has an alternate task $\bar{\tau}_i$, where $C(\bar{\tau}_i) \leq C(\tau_i)$ and $D(\bar{\tau}_i) = D(\tau_i)$. The alternate can typically be a re-execution of the same task, a recovery block, an exception handler or an alternate with imprecise computation.

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 set of critical task alternates. While our framework permits varying levels of task criticality, in this paper, to simplify the illustration, we use binary values for criticalities. For each *task instance* τ_i^j we define an *original feasibility window* delimited by its original earliest start time $est(\tau_i^j)$ and deadline $D(\tau_i^j)$ relative to the start of the LCM.

Obviously, the maximum utilization of the original critical tasks together with their alternates can never exceed 100%. This will imply that, during error recovery, execution of non-critical tasks cannot be permitted as it may result in overload conditions. We assume that the scheduler has adequate support for flagging non-critical tasks as unschedulable during such scenarios, in addition to appropriate error detection mechanisms in the operating system.

Our primary concern is providing schedulability guarantees to all critical tasks in fault tolerant real-time systems which employ time redundancy for error recovery. The basic assumption is that the effects of a large variety of transient and intermittent hardware faults can effectively be tolerated by a simple re-execution of the affected task, whilst the effects of software design faults could be tolerated by executing an alternate action, e.g., recovery blocks or exception handlers. Both 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 an error 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, 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 error propagations and subsequent domino effects.

Our proposed approach enables masking of up to *one error per each task instance* which is a worse scenario compared to earlier assumptions such as one error per longest task period, or an explicit minimum inter-arrival time between consecutive error occurrences.

3 Methodology

3.1 Overview

As the original feasibility windows and original priority assignment (if any, e.g., in case of a legacy system) may not express the various FT requirements, our goal is to, first, derive new feasibility windows for each task instance $\tau_i^j \in \Gamma$ to reflect the FT requirements. Then, we assign FPS attributes that ensure task executions within their new feasibility windows, thus, fulfilling the FT requirements.

While executing non-critical tasks in the background can be a safe and straightforward solution, in our approach we aim to provide non-critical tasks a better service than background scheduling. Hence, depending on the criticality of the original tasks, the new feasibility windows we are looking for differ as:

1. *Fault Tolerant* (FT) feasibility windows for critical task instances
2. *Fault Aware* (FA) feasibility windows for non-critical task instances

While critical task instances need to complete within their FT feasibility windows to be able to re-execute feasibly upon an error, the derivation of FA feasibility windows has two purposes: 1) to prevent non-critical task instances from interfering with critical ones, i.e., to cause a critical task instance to miss its deadline, while 2) enabling the non-critical task execution at high priority levels. Since the size of the FA feasibility windows depend on the size of the FT feasibility windows, in our approach we first derive FT-feasibility windows and then FA feasibility windows. Then, we assign fixed priorities to ensure the task executions within their newly derived feasibility windows.

In some cases, however, FPS cannot express all our assumed FT requirements and error assumptions with the same priorities for all instances directly. General FT requirements may require that instances of a given set of tasks need to be executed in different order on different occasions. Obviously, there exists no valid FPS priority assignment that can achieve these different orders. Our algorithm detects such situations, and circumvents the problem by splitting a task into its instances. Then, the algorithm assigns different priorities to the newly generated "artifact" tasks, the former instances. Key issues in resolving the priority conflicts are the number of artifact tasks created, and the number of priority levels. Depending on how the priority conflict is resolved, the number of resulting tasks may vary, i.e., based on the size of the periods of the split tasks. Our algorithm minimizes the number of artifact tasks by using ILP for solving the priority relations. The major steps of the proposed methodology are shown in Figure 1.

3.2 Proposed approach

In this section we use a simple example throughout the description of our approach. Let our task set consist of 2 tasks, A and B, where $T(A) = 3$, $T(B) = 6$, $C(A) = 2$

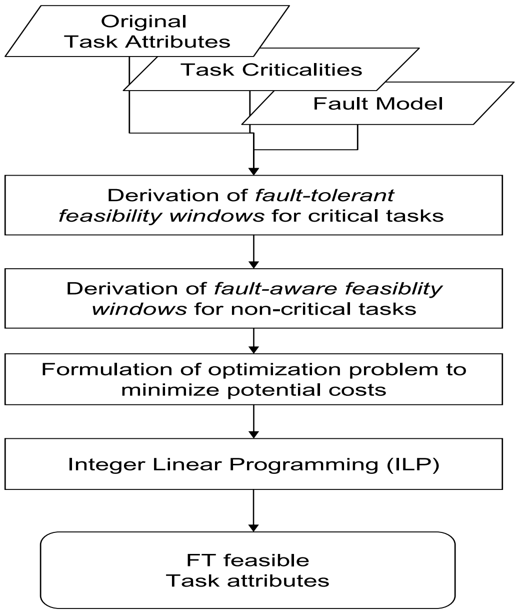

Original Task Attributes
Task Criticalities
Fault Model
Derivation of *fault-tolerant feasibility windows* for critical tasks
Derivation of *fault-aware feasiblity windows* for non-critical tasks
Formulation of optimization problem to minimize potential costs
Integer Linear Programming (ILP)
FT feasible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

Figure 1. Methodology overview

and $C(B) = 2$, scheduled according to the RM policy (Figure 2), where B is the critical task with fault tolerance requirements. Here, the earliest start times and the deadlines are represented by up- and down arrows respectively. We assume that a simple re-execution of the affected task is the fault tolerance strategy.

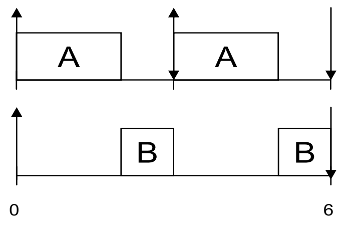


Figure 2. Original task set

To be able to re-execute B upon an error, 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, by doing so, the first instance of A will always miss its deadline, even in error-free scenarios (Figure 3). Moreover, raising the priority of critical tasks may not always ensure fault tolerance in our assumed error scenarios, i.e., one error per task instance, as the processor utilization approaches 100%.

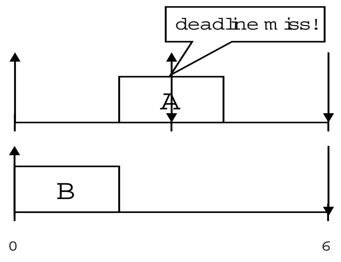


Figure 3. 'B' fault tolerant - 'A' always misses its deadline

3.2.1 Derivation of FT- and FA feasibility windows

The first part of our approach is the derivation of FT and FA feasibility windows for critical and non-critical task instances respectively. Our approach first derives FT deadlines for the primary versions of the critical task instances so that, in case of a critical task error, an alternate version of that instance can be executed before its original deadline. Then FA deadlines for the non-critical task instances are derived so that the provided fault tolerance for the critical ones is not jeopardized. During these steps the goal is to keep the FT and FA deadlines as late as possible in order to maximize the flexibility for the second part of our approach, which is the FPS attribute assignment using an ILP solver.

Derivation of FT deadlines: The aim of this step is to reserve sufficient resources for the executions of the critical task alternates in the schedule. While one can use any method to achieve that, our goal is to provide guarantees in scenarios where the processor utilization can reach 100%. Thus, we choose the approach proposed by Chetto and Chetto [3] to calculate the latest possible start of execution for critical task alternates. Specifically, we select the set of critical tasks Γ_c and their alternates $\bar{\Gamma}_c$ and calculate FT-deadlines for each critical task instance, $D_{FT}(\tau_i^j)$, equal to the latest start time of its alternate $\bar{\tau}_i^j$.In this way we reserve sufficient resources for each critical task instance alternate, assuming that the cumulative processor utilization of the primaries and their alternates does not exceed 100% over LCM. In our example, the FT deadline of B is 4.

Derivation of FA deadlines: We aim to provide FA deadlines to non-critical task instances to protect critical ones from being adversely affected. As a part of recovery action upon errors, the underlying 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.

To derive the FA deadlines, we repeat the process as in

FT deadline derivation, on the set of non-critical tasks, Γ_{nc}, but *in the remaining slack* after the critical task primaries are scheduled to execute as late as possible. We do so due to two reasons: we want to prevent non-critical tasks from delaying the execution of critical primaries beyond their FT deadlines, and to alow non-critical tasks to be executed at high priority levels. In our example the derived FT and FA deadlines are illustrated in Figure 4, where the FA deadlines for the instances of A are 2 and 6 respectively.

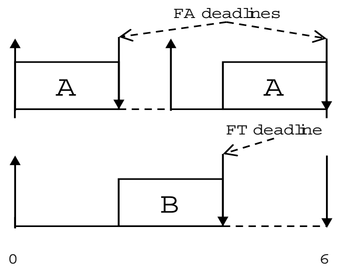


Figure 4. FT and FA deadlines

In some cases, we may fail finding valid FA deadlines for some non-critical task instances. We say that a FA deadline, $D_{FA}(\tau_i^j)$, is *not valid* if $D_{FA}(\tau_i^j) - est(\tau_i^j) < C(\tau_i^j)$. This scenario could occur since the task set consists now of tasks with deadlines less than periods. In these cases, we keep the original deadline, and make sure that the priority assignment mechanism will assign the non-critical task a background priority, i.e., lower than any other critical task, and any other non-critical task with a *valid FA deadline*.

3.2.2 FPS attribute assignment

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 derive priority inequalities between instances to ensure their execution within their derived FT- and FA feasibility windows. By solving the inequalities, our method outputs a set of tasks, Γ_{FPS}, with FPS attributes.

Our task model consists now of four types of task instances: critical task instances consisting of primaries Γ_c and alternates $\bar{\Gamma}_c$, 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 instance, 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

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

We derive priority relations within each subset Γ_{t_k} based on the derived FT and FA deadlines, i.e., the instance with the shortest relative deadline will get the highest priority in each inequality:

$\forall t_k, \forall \tau_i^j, \tau_s^q \in \Gamma_{t_k}$, 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 time. In cases where 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 derived priority inequalities, however, it may happen that different instances of the same task need to be assigned different priorities. These cases cannot be expressed directly with fixed priorities and are the sources for *priority assignment conflicts*. We solve the issue by splitting the tasks with inconsistent priority assignments into a number of new periodic tasks with different priorities. The new task's instances comprise all instances of the original tasks. We use ILP to find the priorities and the splits that yield the smallest number of FT FPS tasks.

3.2.3 ILP formulation

The goal of the attribute assignment problem is to find the minimum number of tasks together with their priorities, that fulfill the priority relations derived so far. 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.

We use ILP since we are only interested in integral priority assignments. In the ILP problem the goal function G to be minimized computes the number of tasks to be used in the FPS scheduler.

$$G = N + \sum_{i=1}^{N}(k_i - 1) * b_i + \sum_{i=1}^{N}\sum_{j=1}^{k_i} \bar{b}_i^j$$

where N is the number of original tasks, k_i is the number of instances of τ_i over LCM, b_i is a binary integral variable that indicates if τ_i needs to be split into its instances and $\bar{b}_i^j$ 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 its primary.

The constraints of the ILP problem reflect the restrictions on the task priorities as imposed by the 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., $P(\bar{\tau}_i^j) > P(\tau_p^q)$ the translated constraint is:

$$\bar{p}_i^j > p_p + p_p^q,$$

where $\bar{\tau}_i^j$ 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 ensures that only either the original tasks or their instances (artifact) are assigned valid priorities (greater than 0) by the ILP solver. All other priorities are set to zero.

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

While both primaries and alternates can coexist at different valid priorities, 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.

$$\left|(p_i + p_i^j) - \bar{b}_i^j\right| \leq \bar{b}_i^j * M$$

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, indicates if τ_i has to be split, i.e., b_i allows only a task or its artifact tasks to be assigned valid priorities. On the other hand, the variable $\bar{b}_i^j$ ia a binary variable that indicates if the alternate of τ_i^j, i.e., $\bar{\tau}_i^j$, can be scheduled at the same priority as its primary. Since the goal function associates a penalty for each b_i and $\bar{b}_i^j$ 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:

$$\forall i, j : \; b_i, \bar{b}_i^j \leq 1$$

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.

3.2.4 Periods and offsets

Once the task priorities ($P(\tau_i)$) have been assigned by the ILP-solver, we can now focus on the assignment of periods ($T(\tau_i)$) and offsets ($O(\tau_i)$). Based on the information provided by the solver, we assign periods and offsets to each task in order to ensure their run time execution under FPS within their respective FT/FA feasibility windows:

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

The final set of tasks executing under FPS is presented in Figure 5. A1 has the highest priority and A2 the lowest. In Figure 5 (a), the tasks execute the worst case scenario, i.e., task execution equal to WCET and errors occurring at the end of the executions. In this case, A2 will be shed by the scheduler due to the system overload. However, at run-time, tasks will most likely execute for less than their WCET's. In such scenarios, B can feasibly re-execute as well as the non critical tasks A1 and A2 can complete before their deadlines (Figure 5 (b)).

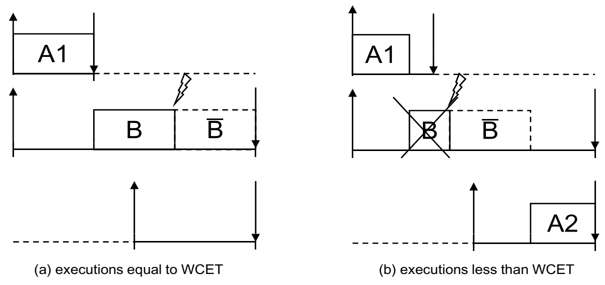


Figure 5. FT feasible taskset

4 Example

We illustrate our method by an example. Let us assume we have a task set schedulable by RM as described in Table

1 and Figure 6.

Task	T	C	P	Criticality
A	3	1	3 (highest)	0 (non-critical)
B	4	1	2	1
C	12	3	1	1

Table 1. Original task set

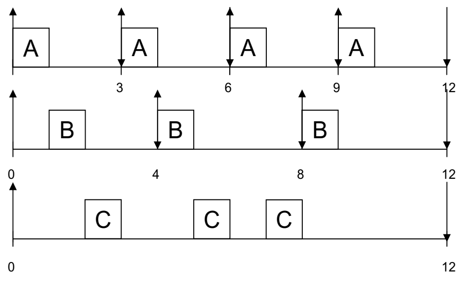


Figure 6. Original RM schedule

Let us now assume B and C are the critical tasks. In this example, RM priority assignment cannot guarantee fault tolerance on every critical task instance, e.g., if all instances of B are hit by faults and need to be re-executed, the primary version of C will always miss its deadline (Figure 7).

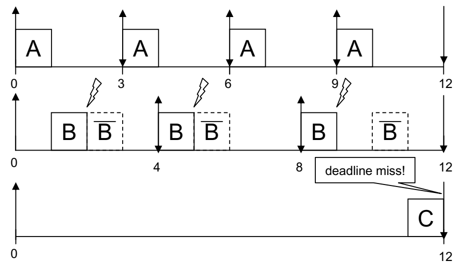


Figure 7. RM schedule in presence of errors - C misses its deadline

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 calculating the latest possible start of execution for critical tasks and alternates (Figure 8). As previously mentioned, the earliest start times and the deadlines are represented by up- and down arrows respectively. The dashed blocks represent the re-execution of the critical tasks instances. Accordingly, the FT feasibility windows for the critical tasks are presented in Figure 9.

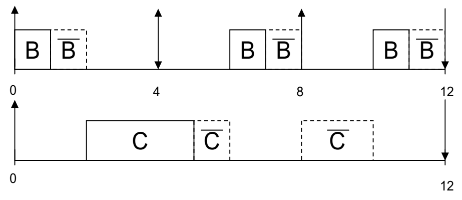


Figure 8. Latest possible executions for critical tasks and alternates

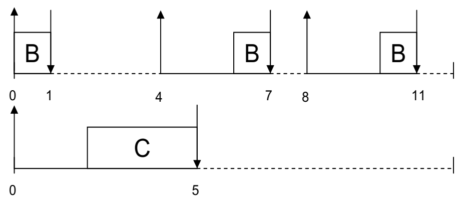


Figure 9. FT feasibility windows for critical tasks (B and C)

At this point, we derive FA feasibility windows for non-critical task instances (in our case, for the instances of A), by scheduling them *as late as possible* [3], *together* with the critical ones and associated FT feasibility windows. The resulting FA feasibility windows are shown in Figure 10.

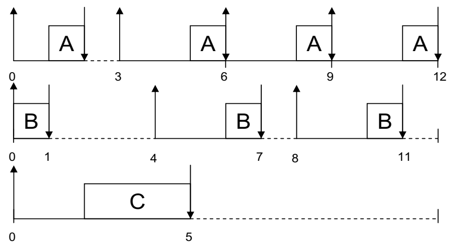


Figure 10. FA feasibility windows for the non-critical task (A)

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 task set and we derive priority relations between the instances as described in Section 3.2.2. The resulting priority inequalities are presented in Table 2.

Next, we formulate the optimization problem. The terms

t_k	$\left\{ \begin{array}{c} current \\ inst. \end{array} \right\}_{t_k}$	$\left\{ \begin{array}{c} intf. \\ inst. \end{array} \right\}_{t_k}$	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 2. Derivation of inequalities

in the ILP goal function, i.e.,

$$G = N + \sum_{i=1}^{N}(k_i - 1) * b_i + \sum_{i=1}^{N}\sum_{j=1}^{k_i}\overline{b}_i^j$$

are:

$$\begin{aligned} N &= 3 \\ \sum_{i=1}^{N}(k_i - 1) * b_i &= 3 * b_A + 2 * b_B + 0 * b_C, \; and \\ \sum_{i=1}^{N}\sum_{j=1}^{k_i}\overline{b}_i^j &= \overline{b}_B^1 + \overline{b}_B^2 + \overline{b}_B^3 + \overline{b}_C^1 \end{aligned}$$

subject to the constraints derived from the priority inequalities. For example, $P(B^1) > P(A^1)$ is translated into the constraint C1:

$$C1 : p_B + p_B^1 > p_A + p_A^1$$

The LP solver provides a set of fault tolerant tasks suitable for FPS to which we assign periods and offsets, as described in section 3.2.4 (Table 3).

In our example, since the utilization is already 100% even in error-free scenarios, 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 FPS schedule is shown in figure 11 in the scenario where every critical task instance is re-executing due to errors. Note that, in this scenario, the non-critical tasks A1-A4 are not executed by the scheduler due to the overload situation. However, one can see that in an error-free

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

Table 3. Fault-tolerant FPS Tasks

scenario, the non critical tasks will be executed at higher priorities than the critical ones (e.g., A1 has the next highest priority). The resulting task set is directly schedulable

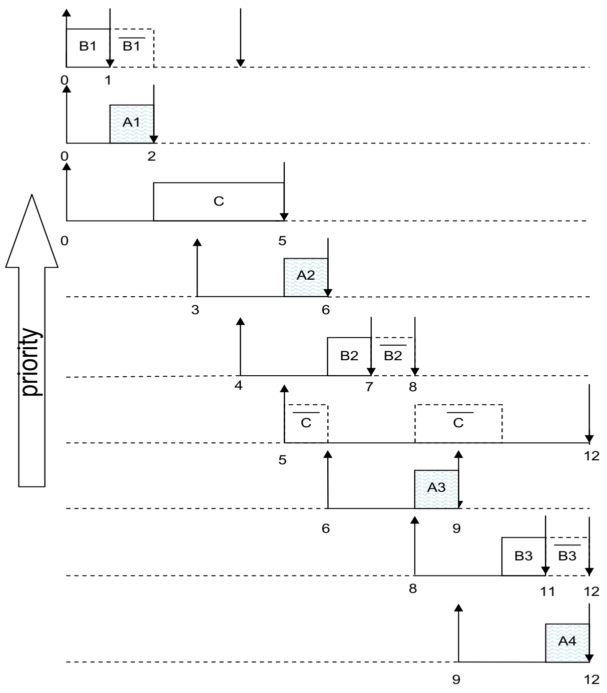
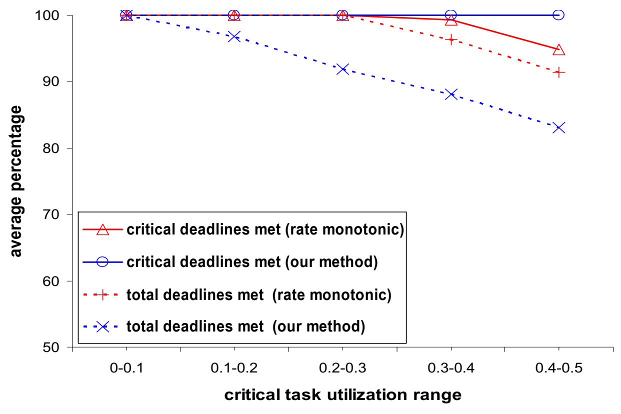
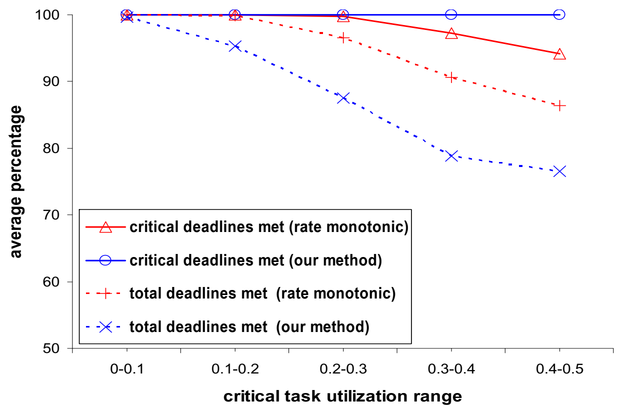


Figure 11. Derived FPS schedule under worst case error occurrences

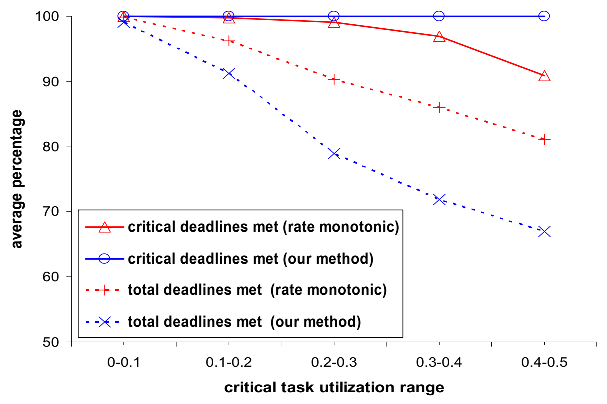
by the original scheduler while the critical tasks can tolerate one error per instance. Our method enables the non-critical tasks to be executed at higher priorities than critical ones, within their derived FA-feasibility windows, without jeopardizing the FT-feasibility of the critical tasks. In case of a critical task failure, however, non-critical tasks will be suspended by the underlying scheduler until the errorneous task has been re-executed.



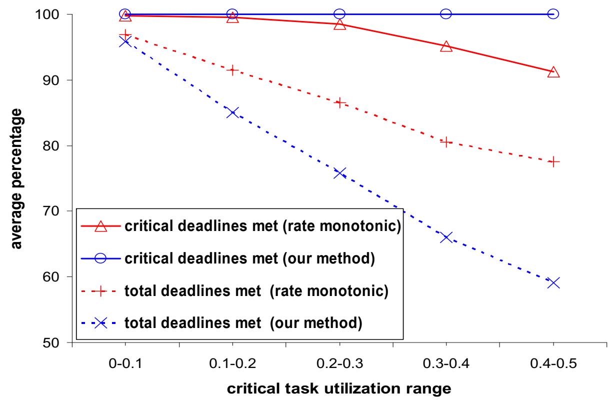
(a) Processor Utilization between 0.6 and 0.7



(b) Processor Utilization between 0.7 and 0.8



(c) Processor Utilization between 0.8 and 0.9



(d) Processor Utilization between 0.9 and 1

Figure 12. Simulation results

5 Evaluation

In this section, we evaluate the performance of our method in comparison with the FT adaptation of RM scheduling policy upon occurrence of errors, where the erroneous tasks are re-executed by the scheduler. We define our primary success criteria as the percentage of critical task re-executions that complete before their deadlines. Meeting the deadlines of non-critical task instances is assumed to be the secondary success criteria. However, in our method we may have to shed non-critical tasks in the favor of critical task re-executions upon failures.

We conducted a number of simulations on synthetic task sets, since the lack of a priori knowledge about when the errors occur and the resulting task interactions would make the comparison procedure rather complex to be performed mathematically. We performed the simulations in the worst case scenario where *every critical task instance is hit by a fault, which is detected at the end of its execution.*

We generated 2000 task sets, 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 total utilization of the task sets varied between 0.5 and 1. After calculating the LCM, task periods were randomly chosen among the divisors of LCM. Randomization was realized by Mersenne Twister pseudorandom number generator with 32-bit word length [15]. 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 sub-group 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 was around 100 milliseconds on a 1GHz PC, when a task set generated as described above was used as input.

Figures 5(a) to 5(d) 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 in the range 0.6-0.7. As the CPU utilization increases, the success of our method increases as well, although 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 5(a)). In the range 0.8-0.9 this threshold decreases to 0.2 (Figure 5(c)). When the processor utilization is between 0.9 and 1 (Figure 5(d)), critical task instances scheduled by RM start to miss their deadlines even when critical task utilization was very low.

In our evaluation we were able to find a feasible solution in all cases, and the results clearly show that our method guarantees the re-execution of every critical task instance before its deadline in the worst case scenario where every critical task instance is hit by an error.

6 Conclusions and future work

In this paper, we presented a methodology which allows the system designer to schedule a set of real-time tasks with mixed criticalities and fault tolerance requirements, in the context of fixed priority based, real-time systems.

Specifically, we proposed a method to analyze a task set, with given criticalities, and derive FPS attributes which guarantee *every critical task instance* to be re-executed upon an error before its deadline, provided the combined utilization of primaries and alternates is less than or equal to 100%. Additionally, our approach enables the execution of non-critical tasks at priority levels higher than the critical ones, in an error-aware manner, thus providing a better service than, e.g., background scheduling, to non-critical tasks.

Our ongoing work aims to incorporate more complex error models, as well as to formalize 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.

7. Acknowledgements

The authors wish to express their gratitude to the reviewers for their helpful comments on the paper.

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. *Dependable Computing for Critical Applications 7, 1999*, pages 361–378, Nov 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] F. Liberato, R. G. Melhem, and D. Mosse. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transactions on Computers*, 49(9):906–914, 2000.

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

[12] 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.

[13] G. Lima and A. Burns. Scheduling fixed-priority hard real-time tasks in the presence of faults. *Lecture Notes in Computer Science*, pages 154–173, 2005.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] R. M. Santos, J. Santos, and J. D. Orozco. A least upper bound on the fault tolerance of real-time systems. *J. Syst. Softw.*, 78(1):47–55, 2005.

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

[21] 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.