Scheduling Fixed-Priority Hard Real-Time Tasks in the Presence of Faults

George Lima[1] and Alan Burns[2]

[1] Distributed Systems Lab (LaSiD),
Department of Computer Science (DCC),
Federal University of Bahia (UFBA),
Salvador, BA, Brazil
gmlima@ufba.br
[2] Real-Time Systems Research Group,
Department of Computer Science,
University of York, York, UK
burns@cs.york.ac.uk

Abstract. We describe an approach to scheduling hard real-time tasks taking into account fault scenarios. All tasks are scheduled at run-time according to their fixed priorities, which are determined off-line. Upon error-detection, special tasks are released to perform error-recovery actions. We allow error-recovery actions to be executed at higher priority levels so that the fault resilience of the task set can be increased. To do so, we extend the well known response time analysis technique and describe a non-standard priority assignment policy. Results from simulation indicate that the fault resilience of the task sets can be significantly increased by using the proposed approach.

1 Introduction

Hard real-time systems are those that *have* to produce correct results within specified deadlines. A flight control system is an example of such a system. Should it fail to produce correct or timely results, an accident may happen. In other words, high costs, in terms of human lives or monetary loss, are usually associated with failures in such a kind of system.

Due to the criticality level of their computation, dealing with hard real-time systems is not simple. In order to provide fault tolerance, the system must be designed making use of redundant components. In order to provide timeliness, the system computation must be organized so that its timing specifications are met. Also, there must be ways of proving the system timeliness given the characteristics of both the system and the environment it is subject to, which must take the presence of faults into account. Preferably, this timeliness checking must be carried out before the system is operational.

Certainly, the use of *active* components, replicated and distributed across the system is of great help to build fault-tolerant real-time systems. In this case, there must be a robust protocol to coordinate the replicas in a timely fashion,

C.A. Maziero et al. (Eds.): LADC 2005, LNCS 3747, pp. 154–173, 2005.

which means that extra computational efforts must be spent even in the absence of faults in order to make the system fault tolerant. An alternative approach is to have *passive* redundant components which can be activated upon error-detection. Since fault scenarios are exceptions, extra computational effort due to fault tolerance can be minimised. Although there is a higher time delay for detecting the error and recovering the system, the latter approach can be effective. Indeed, it is possible to introduce a greater level of flexibility in the system since the redundant component, when activated, may carry out alternative actions to recover or compensate the system from the specific detected error. Also, several modern programming languages allow for error-detection and recovery mechanisms to be programmed at the application level (*e.g.* exception handler) [4] so that the application needs are taken into account. Most importantly, both active and passive redundancy approaches can be jointly employed and in fact they may be designed to complement each other. For example, a transient fault may cause an active component to produce incorrect results. This error can be detected and a passive component can be activated to undo or redo the faulty component actions or even to silence it (*e.g.* shutting it down).

When applying the passive redundancy approach in the context of hard real-time systems a problem to be solved is how to compute the extra time required to execute the component actions when errors are detected. This is the focus of this paper. More specifically, we structured the system as a set of tasks, some of which execute only when errors are detected. Our goals are to: (a) determine whether the execution of system tasks meet their deadlines in the presence of faults; and (b) increase the system fault resilience by providing an appropriate task scheduling mechanism. To do so, we have developed a scheduling mechanism that can be adjusted (by priority assignment) to increase the fault tolerance capacity of the system and have derived a set of equations that are used to verify the system schedulability.

Our approach is based on determining (off-line) which priorities can be assigned to tasks so that more errors can occur without compromising the specified deadlines. As priority represent the urgency of execution, a task that has to be recovered (preferably) should have its recovery actions executing with higher priority. Determining the task priorities off-line is attractive because it provides a simple on-line scheduling criterion (*i.e.* the scheduler, at run-time, needs only to choose the highest priority task to execute). Also, complex and perhaps timing-consume criteria to determine the (best) priority assignments can be used.

The approach described in this paper is an extension of our former work [15], where a less restricted fault model is assumed. Indeed, here we assume that errors may take place at any time instead of considering that errors take place periodically in worst case.

The remainder of this paper is structured as follows. A brief literature review is given in the next section. The computation model is described in Section 3. Then, Section 4 presents some basic concepts on hard real-time scheduling and illustrates the addressed problem. Section 5 derives the schedulability analysis. The problem of searching for a priority assignment that improves the fault re-

silience of the system is addressed in Section 6. Some simulation results are also presented in this section. Then, in Section 7, our final comments are given.

2 Related Work

We identify two branches of work on providing fault tolerance in hard real-time systems, distributed protocols and scheduling. The former deals with coordinating the computation between different nodes when a distributed architecture is necessary [21,20,10]. On the other hand, for hard real-time systems, one has also to be concerned with the (local) computation carried out in the system nodes. Indeed, independently of whether the system is distributed or not, scheduling is a fundamental problem for real-time systems. In this section we sumarise only approaches focused on scheduling.

Scheduling for fault tolerance can further be divided into two categories: those approaches that take into account task replicas running in different nodes and those that focus on the execution of tasks considering only local computation. There are several examples of the former approach [11,2,9,10,16,17]. In this paper we deal with scheduling from the point of view of local nodes with the goal of providing fault tolerance within each node (if the system is distributed). Approaches to doing so usually consider two types of tasks running in the nodes, *primary* and *alternative*. Primary tasks represent the usual computation that needs to be performed in error-free scenarios. Alternative tasks contain actions that must be executed when some error is detected.

One of the first such a mechanism to schedule primary and alternative tasks was described by Liestman *et al.* [13]. This mechanism only deals with periodic tasks, whose periods have to be multiples of each other. The approach presented by Ghosh *et al.* [7] limits the recovery of faulty tasks to re-executing them. Only transient faults can be tolerated (*e.g.* design faults are not considered). An interesting approach to tolerating transient faults which is independent of the schedulability analysis being used has been described by Ghosh *et al.* [6]. However, only the re-execution of faulty tasks as a means of fault tolerance is assumed. Kandasamy *et al.* [8] describe a recovery technique that tolerates transient faults in an *off-line* scheduled distributed system. It is based on taking advantage of task set spare capacity. The amount of spare capacity is distributed over a given period so that task faults can be handled. Although tasks are assumed to be preemptive and their precedence relations are taken into account, only periodic tasks, whose periods are equal to deadlines, are considered.

Recently, an EDF (Earlier Deadline First) based scheduling, which takes the effects of transient faults into account, has been proposed [12]. Its basic idea is to simulate the EDF scheduler and to use slack time for executing task recoveries given a *fault pattern*. Fault patterns, which are the assumed maximum numbers of errors per task, must be known *a priori*. Task recoveries can be modelled as alternative tasks that are released after error-detection. Another EDF based scheduling approach for supporting fault-tolerant systems has been proposed by Caccamo *et al.* [5]. Their task model consists of instance skippable and fault-

tolerant tasks. The former may allow the system to skip one instance once in a while. The latter is not skippable (*i.e.* all instances have to execute by their deadlines) and is composed of a primary and an alternative part. The primary part is scheduled on-line and provides high-quality service while the alternative one is scheduled off-line and provides acceptable services.

The approach presented by Ramos-Thuel *et al.* [19] is based on the *transient server* concept. Its basic idea is to explore the spare capacity of the task set to determine the maximum server capacity at each priority level. A server is an *a priori* created task used to service aperiodic requests. In their approach such requests are the detection of errors. The spare capacity allocated to the server is used for *on-line* dispatching decisions in the case of error occurrences. Although this approach seems interesting since higher priority levels are used to execute alternative tasks, a reasonable way of determining the server periods has not been presented.

A flexible approach that makes use of fixed-priority scheduling and response time analysis has been proposed by Burns *et al.* [3] and Punnekkat [18]. No restriction on alternative tasks is assumed. This approach shows that response time analysis can be straightforwardly adapted to take the execution of alternative tasks into account. Making use of this results, we have recently showed that non-standard priority assignments can be used to increase the fault resilience of the system [15]. Like Burns *et al.* we have restricted the fault model by assuming that there is a minimum time between consecutive errors.

In this paper we extend our former work [15] by removing this restriction on time between consecutive errors. By doing so, we take into consideration more general situations where errors may affect the execution of tasks at any time.

3 Computation Model

We assume that there is a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of n tasks, called *primary tasks*, that must be scheduled by the system in the absence of errors. Any primary task τ_i in Γ has a period, T_i, a deadline D_i ($D_i \leq T_i$), and a worst-case computation time, C_i. Tasks can be periodic or sporadic. For sporadic tasks the period means the minimum inter-arrival time. Each primary task τ_i can have some *alternative tasks* associated with it. Each alternative task corresponds to a given action taken to recover τ_i from a given error. Any alternative task has a worst-case computation time, also called worst-case recovery time. For the sake of simplicity we denote $\overline{\tau}_i$ as the alternative task of τ_i whose worst-case recovery time is the largest one. Also, we assume that all alternative tasks associated with τ_i run at the same priority level. Hence, hereafter we do not include the details of individual alternative task per primary in the description we present. We only need to refer to $\overline{\tau}_i$ as the worst-case alternative task in case of errors in task τ_i.

Primary tasks are scheduled according to some fixed priority assignment algorithm, which attributes a distinct priority to each task τ_i in Γ. We consider n different priority levels $(1, 2, \ldots, n)$, where 1 is the lowest priority level. The alternative tasks of τ_i are assumed to execute at priority levels greater than or

equal to τ_i's priority. We denote the priority of τ_i and $\overline{\tau}_i$ as $\mathsf{pr}(\tau_i)$ and $\mathsf{pr}(\overline{\tau}_i)$, respectively. When a primary task, say τ_i, and an alternative task, say $\overline{\tau}_j$, are ready to execute at the same priority level, we assume that $\overline{\tau}_j$ is scheduled first.

Alternative tasks represent some extra processing that is necessary to recover a task from a given erroneous state caused by a fault. Errors are detected at the task level. When an error interrupts the execution of a task, the system must schedule an appropriate alternative task, which is responsible for carrying out the error-processing procedure and has to finish by the deadline of its primary task. If other errors take place in the alternative tasks, we assume that it is scheduled again for re-execution. We also assume that there is no cost associated with any scheduling of primary or alternative tasks. These costs are assumed to be taken into account by the value of C_i and $\overline{C}_i$, respectively. Further, we assume that all errors are detected by the system and there is no fault propagation in the value domain (*i.e.* faults affect only the results produced by the executing task).

The kinds of fault with which we are dealing are those that can be treated at the task level. Consider for example *design* faults. It may be possible to use techniques such as *exception handling* or *recovery blocks* to perform appropriate recovery actions [3], modelled here as alternative tasks. In addition, one may consider some kind of *transient faults*, where either the re-execution of the faulty task or the execution of some compensation action is effective. For example, suppose that transient faults in a sensor (or network) prevent an expected signal from being correctly received (or received at all) by the control system. This kind of system fault can easily be modelled by alternative tasks, which can be released to carry out a compensation action. However, it is important to emphasise that we are not considering more severe kinds of fault that cannot be treated at the task level. For example, if a memory fault causes the value of one bit to be arbitrarily changed, the operating system may fail, compromising the whole system. Tolerating these kinds of fault requires spatial redundancy (perhaps using a distributed architecture) and is not covered in this paper. Our work fits the engineering approach that uses temporal redundancy at the processor level and spacial redundancy at the system level.

We derive the schedulability analysis of the system as a function of a fault resilience metric, denoted by N_E. The value of N_E represents the assumed maximum number of errors that task set may suffer. The goal of the analysis is to show: (a) whether or not the system is schedulable for a given value of N_E; and (b) whether or not a more resilient system can be built by assigning priorities to (alternative) tasks appropriately.

4 Initial Concepts

The schedulability derived in this paper is based on the well known response time analysis [1]. The basic idea is to compute the worst-case response time of each task in the system, R_i, and compare with its deadline. If $R_i \leq D_i$ for all tasks, then the system is schedulable. In the next two subsections we introduce some basic concepts in response time analysis and show how the effects of faults can

be easily incorporated into the analysis. At the end of the section we illustrate potential advantages of having some alternative tasks running at higher priority levels.

4.1 Fault Free Scenarios

To compute R_i one has to consider the worst-case scenario. This happens when all other higher priority tasks τ_j are released at the same time as τ_i and the execution of τ_i and all higher priority tasks τ_j take C_i and C_j to complete, respectively. In this scenario, the value of R_i is given by C_i plus the sum all $n_j C_j$, where n_j is the maximum number of instance of τ_j that can occur during the execution of τ_i. Note that n_j depends on the response time of τ_i, *i.e.* $n_j = \lceil \frac{R_i}{T_j} \rceil$. Thus, R_i can be written as [1]:

$$R_i = C_i + \sum_{j \in \mathsf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \,, \tag{1}$$

where $\mathsf{hp}(i)$ is the set of tasks that have higher priority than τ_i and $\lceil x \rceil$ returns the smallest integer that is greater than or equal to x.

Since the term R_i appears in both sides of equation (1), it is solved iteratively applying the relation given by equation (2) [1]. The iteration can start with $r_i^0 = C_i$, where r_i^k is the k^{th} approximation to the true value of R_i. The interactions can be halted when $r_i^{k+1} > D_i$ or earlier if $r_i^{k+1} = r_i^k$. In the former case, the task is not schedulable, while the latter means that $R_i = r_i^k$.

$$r_i^{n+1} = C_i + \sum_{j \in \mathsf{hp}(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \,. \tag{2}$$

4.2 Fault Scenarios

If an error occurs, from the point of view of τ_i, the worst case is when this error interrupts the execution the task with which is associated the longest alternative task among all tasks τ_k that can cause the interference in the execution of τ_i (including τ_i itself). Considering N_E errors and the fact that alternative tasks run with the same priority as their primary tasks, equation (1) can be extended as a function of N_E:

$$R_i(N_E) = C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + N_E \max_{\tau_k \in \mathsf{hpe}(i)} \overline{C}_k \,, \tag{3}$$

where $\mathsf{hpe}(i)$ is the set of tasks that have priorities higher of equal to the priority of τ_i. Indeed, for each error occurrence $\overline{C}_k$ time units are incorporated into the computation of R_i.

As an illustration, consider a set of 3 tasks and their alternative tasks shown in Table 1. The values in its last column are the worst-case response times for $N_E = 1$. To illustrate this, the following is what the iterative computation of R_3 looks like:

Table 1. A task set and the derived worst-case response times

Task set						$N_E = 1$
Task	T_i	C_i	$\overline{C}_i$	D_i	$\mathsf{pr}(\tau_i)$	$R_i(1)$
τ_1	13	2	2	13	3	4
τ_2	25	3	3	25	2	8
τ_3	30	5	5	30	1	22

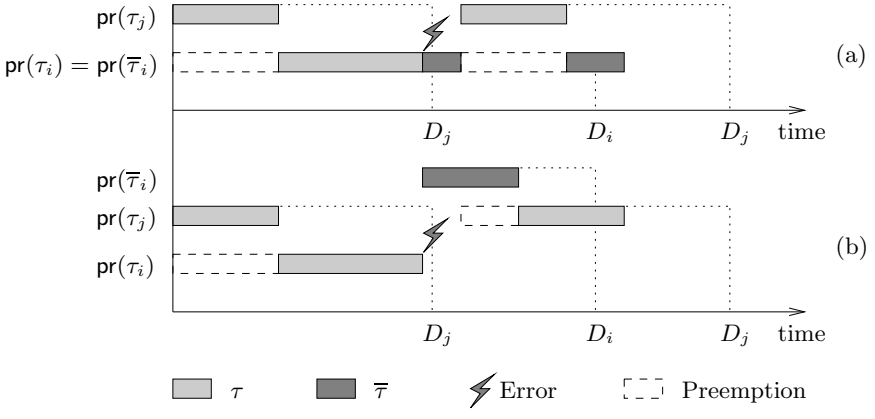

Fig. 1. Priority assignment in fault-scenarios

$$
\begin{aligned}
r_3^0(1) &= 5 \\
r_3^1(1) &= 5 + \left\lceil \frac{5}{13} \right\rceil 2 + \left\lceil \frac{5}{25} \right\rceil 3 + 5 = 15 \\
r_3^2(1) &= 5 + \left\lceil \frac{15}{13} \right\rceil 2 + \left\lceil \frac{15}{25} \right\rceil 3 + 5 = 22 \\
r_3^3(1) &= 5 + \left\lceil \frac{22}{13} \right\rceil 2 + \left\lceil \frac{22}{25} \right\rceil 3 + 5 = 22 \\
R_3(1) &= 22
\end{aligned}
$$

The alternative tasks of Table 1 inherit the priorities of their primary tasks, which are given by some conventional priority assignment (*e.g.* DM - Deadline Monotonic). However, when errors take place, it may be advantageous to execute alternative tasks at higher priority levels. The intuition is that, after an error, tasks (or their recovery/compensation actions) certainly have a shorter period of time to meet their deadlines.

Consider Fig. 1, where the execution time line of two tasks, $\{\tau_i, \tau_j\}$, is shown. The priorities of τ_j and τ_i are assigned by the deadline monotonic approach, *i.e.* $\mathsf{pr}(\tau_j) > \mathsf{pr}(\tau_i)$, since $D_j < D_i$. An error interrupts τ_i just before the end of its execution, as illustrated. In scenario (a), it is assumed that $\overline{\tau}_i$ is executed with priority level $\mathsf{pr}(\tau_i)$ whereas in scenario (b) the priority of $\overline{\tau}_i$ is greater than

the priority of τ_j. As can be seen, in Fig. 1(a) τ_i is not schedulable due to the preemption caused by the execution of the second release of τ_j. This preemption is avoided by executing $\overline{\tau}_i$ at a higher priority level, as illustrated in Fig. 1(b). In the next section, a new set of equations is derived to take into consideration this kind of priority assignment.

5 Schedulability Analysis: A More General Approach

Let Γ be a given task set and $N_E \geq 0$ an integer. The main goal of this section is to develop schedulability analysis to check whether or not Γ is schedulable if up to N_E errors take place during the execution of any task in Γ. Unlike the previous section, we assume here that alternative tasks may have higher priorities than their respective primary tasks. This assumption, as will be seen, makes the analysis much more complex. The problem of determining the priorities of the alternative tasks is postponed to Section 6. Here we focus only on the analysis by assuming that the priority assignment is known.

The strategy to derive the analysis is the following. First, in Section 5.2, we consider that τ_i is fault-free. In this case we say that there are only *external* errors regarding τ_i (*i.e.* errors take place in other tasks but τ_i). Thus, any task (primary or alternative) that has priorities higher than or equal to the priority of τ_i may interfere in its execution. Then, we consider the case of some error in τ_i, *i.e.* there is some *internal* error. This case, addressed in Sections 5.3 and 5.4, is characterised by the fact that the execution of $\overline{\tau}_i$ may interfere in the execution of higher priority tasks. As will be seen, this case is more complicated because there are two phases during the response time of τ_i: before and after the release of $\overline{\tau}_i$. Because of this, the problem of finding what the distribution of errors leads to the worst-case scenario has to be considered. This problem is addressed in Section 5.5. Before describing the analysis, we present some definitions in the next section.

5.1 Definitions

A particular choice of priorities for alternative tasks, named *priority configuration*, is defined as follows:

Definition 1 (Priority configuration). *A priority configuration P_x is a tuple $\langle h_{x,1}, h_{x,2}, \ldots, h_{x,n}\rangle$, where $0 \leq h_{x,i} < i$ and $h_{x,i} = \mathsf{pr}(\overline{\tau}_i) - \mathsf{pr}(\tau_i)$.*

Note that $h_{x,i}$ represents the priority increment for task $\overline{\tau}_i$ in relation to the priority of its primary task τ_i. The definition of $h_{x,i}$ bounds the priority of $\overline{\tau}_i$ from τ_i's priority to the highest priority level. Lower priority levels are not considered. For example, consider $P_x = \langle 0, 0, \ldots, 0\rangle$ a priority configuration. This means that any alternative task executes at the same priority level as the primary task with which it is associated. For $P_x = \langle 0, 0, \ldots, 0, 1\rangle$, all tasks execute at their original priority level apart from $\overline{\tau}_n$, which executes one priority level above its primary task.

Given a priority configuration P_x, the following subsets of Γ regarding the priority of task $\tau_i \in \Gamma$ can be defined:

- $\mathsf{ip}(x,i)$. These are the tasks that may interfere in the response time of τ_i as regards priority configuration P_x if an error occurs. More formally, $\mathsf{ip}(x,i) = \{\tau_j \in \Gamma |\, h_{x,j} + \mathsf{pr}(\tau_j) \geq \mathsf{pr}(\tau_i)\}$.
- $\mathsf{sp}(x,i)$. Tasks that belong to such a subset do not suffer any extra interference when errors interrupt the execution of τ_i as regards priority configuration P_x. This is because their priorities are superior to $\mathsf{pr}(\overline{\tau}_i)$. More formally, $\mathsf{sp}(x,i) = \{\tau_j \in \Gamma |\ \ \mathsf{pr}(\tau_j) > h_{x,i} + \mathsf{pr}(\tau_i)\}$.
- $\mathsf{ipe}(x,i)$. This subset is defined as follows. If $h_{x,i} = 0$, then $\mathsf{ipe}(x,i) = \mathsf{ip}(x,i)$. Otherwise (when $h_{x,i} > 0$), $\mathsf{ipe}(x,i) = \mathsf{ip}(x,i) - \{\tau_i\}$. This subset is particularly useful for modelling cases where errors may interrupt task τ_i since the maximum interference its recovery suffers depends on whether or not $\mathsf{pr}(\tau_i) = \mathsf{pr}(\overline{\tau}_i)$. The meaning of this subset will be clearer later on when we describe the effects of internal errors.

5.2 External Errors

The computation of $R_i^{ext}(x, N_E)$, the worst-case response time of task τ_i due to external errors, is straightforward. This is because $\overline{\tau}_i$ does not need to be considered. In this situation, the worst-case scenario, as for task τ_i, can be described as follows: (a) every task that executes requires its worst-case execution time; (b) errors take place just before the end of the execution of tasks; (c) just before the release of τ_i some alternative task with maximum recovery time among all tasks in $\mathsf{ip}(x,i) - \{\tau_i\}$ is released; and (d) all tasks in $\mathsf{hp}(i)$ are released at the same time as τ_i. Therefore, one has to take into account the time to execute τ_i plus all tasks in $\mathsf{hp}(i)$ and the time to recover the faulty task times the maximum number of errors that may occur over $R_i^{ext}(x, N_E)$. This scenario yields equation (4):

$$R_i^{ext}(x,N_E) = C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{R_i^{ext}(x,N_E)}{T_j} \right\rceil C_j + N_E \max_{\tau_k \in \mathsf{ip}(x,i) - \{\tau_i\}} (\overline{C}_k)\,, \quad (4)$$

It is not difficult to see that if $\overline{C}_i < \max_{\tau_k \in \mathsf{ip}(x,i)}(\overline{C}_k)$, equation (4) gives the worst-case response time of τ_i. Indeed, if only external errors take place, the meaning of equation (4) is similar to equation (3). Also, it is not difficult to see that if some internal error happens, $\overline{C}_i$ cannot cause higher interference than task $\overline{\tau}_k$. Thus, the lemma bellow follows [14]:

Lemma 1. *Let Γ be a fixed-priority set of primary tasks and their respective alternative tasks. Suppose that Γ is subject to faults so that there are at most $N_E \geq 0$ errors during the execution of any task. Also, let P_x be a priority configuration for the alternative tasks. If $\overline{C}_i < \max_{\tau_k \in \mathsf{ip}(x,i)}(\overline{C}_k)$, $R_i^{ext}(x, N_E)$ represents the worst-case response time of τ_i regardless of whether or not the execution of τ_i is interrupted by some error.*

5.3 Internal Errors: Some Intuition

Note that the input parameter N_E does not say much about how the error occurrences are distributed. For example, assume that $N_E = 2$. In this case there are two scenarios that must be considered regarding task $\tau_i \in \Gamma$: (a) an error interrupts the execution of some other task τ_j before the first internal error hits τ_i; or (b) the second error takes place after τ_i suffers the internal error. The problem is that it is not possible to know beforehand which of these scenarios represent the worst-case. Thus, it is convenient to define N_E as follows. For each task $\tau_i \in \Gamma$:

$$N_E = N_i^0 + N_i^1 \tag{5}$$

The terms N_i^0 and N_i^1 stand for the maximum number of errors that may take place before and after (or at) the time the first internal error hits τ_i, respectively. Hence, if $N_E = 2$, the possible combinations are: $N_i^0 = 1$ and $N_i^1 = 1$ for scenario (a); and $N_i^0 = 0$ and $N_i^1 = 2$ for scenario (b). The combination $N_i^0 = 2$ and $N_i^1 = 0$ does not need to be considered since it would imply that all N_E errors are external. Therefore, in general, if $N_E = k$, there are k different scenarios that must be analysed in order to determine which one represents the worst-case.

The worst-case response time of a task τ_i considering the occurrence of some internal error is now a function of P_x, N_i^0 and N_i^1 and is denoted by $R_i^{int}(x, N_i^0, N_i^1)$. The approach to computing its value is divided into two steps. This is because the procedure to calculate $R_i^{int}(x, N_i^0, N_i^1)$ has to take into account two levels of priorities (before and after the first internal error) when $\mathsf{pr}(\overline{\tau}_i) > \mathsf{pr}(\tau_i)$.

It is important to emphasise that the values of both N_i^0 and N_i^1 must be set such that they lead to the worst-case value of $R_i^{int}(x, N_i^0, N_i^1)$. A simple procedure for determining the appropriate values of N_i^0 and N_i^1 is iterative and must evaluate all scenarios. This procedure is explained shortly after the descriptions of the equations that give $R_i^{int}(x, N_i^0, N_i^1)$.

5.4 Internal Errors: The Derivation

Assume that the values of N_i^0 and N_i^1 are known and that at least an internal error takes place at some time t (see Fig. 2). What has to be computed is the maximum time $\overline{\tau}_i$ lasts if it is subject to both other possible errors and the interference due to tasks in $\mathsf{sp}(x, i)$ from t onwards. This time is represented in the figure by $R_i^{int^1}(x, N_i^1)$.

In the worst case there may be N_i^1 errors over the period $R_i^{int^1}(x, N_i^1)$. The first error accounts for $\overline{C}_i$, while the others may cause the release of the recovery of any task in $\mathsf{sp}(x, i) \cup \{\tau_i\}$. The worst case is when all $N_i^1 - 1$ other errors interrupt a task in $\mathsf{sp}(x, i) \cup \{\tau_i\}$ that has the longest recovery time.[1] Therefore, the value of $R_i^{int^1}(x, N_E^1)$ can be computed iteratively by

[1] Note that here a generic situation is assumed. However, in practice one can consider that all errors from t onwards are internal, due to lemma 1.

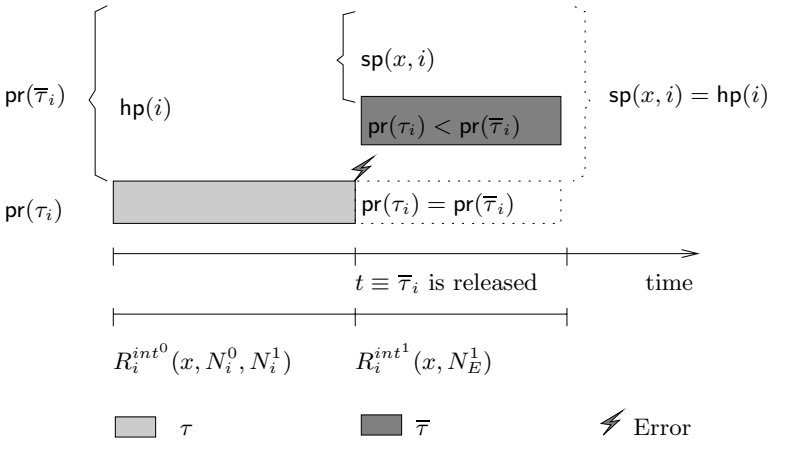

Fig. 2. Illustration of the derivation of R_i^{int}

$$R_i^{int^1}(x, N_i^1) = \overline{C}_i + \sum_{\tau_j \in \mathsf{sp}(i)} \left\lceil \frac{R_i^{int^1}(x, N_i^1)}{T_j} \right\rceil C_j + \left(N_i^1 - 1\right) \max_{\tau_k \in \mathsf{sp}(x,i) \cup \{\tau_i\}} (\overline{C}_k)\,. \tag{6}$$

The computation of $R_i^{int^0}(x, N_i^0, N_i^1)$ is slightly more complex. Let us analyse it considering two cases depending on the values of $\mathsf{pr}(\tau_i)$ and $\mathsf{pr}(\overline{\tau}_i)$:

When $\mathsf{pr}(\tau_i) < \mathsf{pr}(\overline{\tau}_i)$. This means that $\overline{\tau}_i$ executes at a higher priority level. Note that, in this case, knowing $R_i^{int^0}(x, N_i^0, N_i^1)$ is equivalent to knowing the relative earliest possible release time of τ_i so that it suffered the first internal error at time t, as illustrated in Fig. 2.
During $R_i^{int^0}(x, N_i^0, N_i^1)$, τ_i may suffer the preemption of tasks in $\mathsf{hp}(i)$ and possibly the recoveries of tasks in $\mathsf{ip}(i) - \{\tau_i\}$ due to other errors. It is important to note that τ_i has to be removed from the set of tasks that may suffer errors in this phase because, by assumption, the first internal error occurs at time t. Indeed, if there was an earlier internal error, then $\overline{\tau}_i$ would be released earlier and so it would finish earlier. It is clear that this situation does not represent the worst-case scenario.

When $\mathsf{pr}(\tau_i) = \mathsf{pr}(\overline{\tau}_i)$. Unlike the former case, the maximum interference during the period $R_i^{int^0}(x, N_i^0, N_i^1)$ can take place when all errors are internal since both τ_i and its alternative task run at the same priority level. This situation happens, for example, when $\overline{C}_i = \max_{\tau_k \in \mathsf{ip}(x,i)}(\overline{C}_k)$. As a result, instead of considering errors in $\mathsf{ip}(x, i) - \{\tau_i\}$, one should consider errors in the whole $\mathsf{ip}(x, i)$.

In summary, as far as possible errors during $R_i^{int^0}(x, N_i^0, N_i^1)$ are concerned, when $\mathsf{pr}(\tau_i) = \mathsf{pr}(\overline{\tau}_i)$, one has to consider errors in $\mathsf{ip}(x, i) - \{\tau_i\}$. Otherwise, errors in $\mathsf{ip}(x, i)$ should be taken into account. This is the main difference between the cases analysed above. In order to join both cases together in a single equation,

the set $\mathsf{ipe}(x,i)$ can be used. Indeed, errors during the interval $R_i^{int^0}(x, N_i^0, N_i^1)$ may take place in any task in $\mathsf{ipe}(x,i)$.

The equation that gives $R_i^{int^0}(x, N_i^0, N_i^1)$ can now be derived. It has to take into account: the worst-case execution time of τ_i (C_i); the interference due to tasks in $\mathsf{hp}(i)$; and possible recoveries of tasks in $\mathsf{ipe}(x,i)$. Note that some releases of tasks in $\mathsf{sp}(x,i)$ may already have been taken into account when computing $R_i^{int^1}(x, N_i^1)$. This means that one has to take care not to include the same task in $\mathsf{sp}(x,i)$ twice. In other words, one has to subtract for each task in $\mathsf{sp}(x,i)$ and each error occurrence the interference already computed in $R_i^{int^1}(x, T_E)$.

From the description above, equation (7) gives the value of $R_i^{int^0}(x, N_i^0, N_i^1)$. Note that instead of computing the worst-case interference due to tasks in $\mathsf{hp}(i)$, this computation is split as for two complementary subsets, $\mathsf{hp}(i) - \mathsf{sp}(x,i)$ and $\mathsf{sp}(x,i)$. This is to avoid the computation of tasks in $\mathsf{sp}(x,i)$ more than once, as previously mentioned. This is done by subtracting $\lceil \frac{R_i^{int^1}(x,N_i^1)}{T_l} \rceil \overline{C}_l$ for each task $\tau_l \in \mathsf{sp}(x,i)$.

$$R_i^{int^0}(x, N_i^0, N_i^1) = C_i + \sum_{\tau_j \in \mathsf{hp}(i) - \mathsf{sp}(x,i)} \left\lceil \frac{R_i^{int^0}(x, N_i^0, N_i^1)}{T_j} \right\rceil C_j + \sum_{\tau_l \in \mathsf{sp}(x,i)} \left(\left\lceil \frac{R_i^{int}(x, N_i^0, N_i^1)}{T_l} \right\rceil - \left\lceil \frac{R_i^{int^1}(x, N_i^1)}{T_l} \right\rceil \right) C_l + N_i^0 \max_{\tau_k \in \mathsf{ipe}(x,i)} (\overline{C}_k) \,. \quad (7)$$

The final value of $R_i^{int}(x, N_i^0, N_i^1)$ is given by summing up $R_i^{int^0}(x, N_i^0, N_i^1)$ and $R_i^{int^1}(x, N_i^1)$:

$$R_i^{int}(x, N_i^0, N_i^1) = R_i^{int^0}(x, N_i^0, N_i^1) + R_i^{int^1}(x, N_i^1) \,. \quad (8)$$

5.5 Number of Errors

The problem of finding out appropriate values of N_i^0 and N_i^1 is solved iteratively. The idea is to use the schedulability analysis to check which combination of N_i^0 and N_i^1 leads to the worst-case scenario. The algorithm to do so is described in Fig. 3.

The idea of the algorithm is to *distribute* N_E error occurrences during the worst-case response time of a task τ_i. One error hits task τ_i (by assumption). The other $N_E - 1$ errors are considered to be either in $R_i^{int^0}(x, N_i^0, N_i^1)$ or in $R_i^{int^1}(x, N_E)$, depending on which choice gives higher values for $R_i^{int}(x, N_i^0, N_i^1)$. If the task set is unschedulable in some iteration, then the task set does not tolerate N_E errors. Otherwise, the final values of N_i^0 and N_i^1 are given by N^0 and N^1, respectively. At the end of the algorithm, the value of variable R contains the worst-case response time considering internal errors as long as the task set is schedulable in P_x with N_E errors (some of them internal).

```
// This algorithm needs to be executed for each task τ_i ∈ Γ that has:
// (a) R_i^ext(x, N_E) ≤ D_i; (b) pr(τ̄_i) > pr(τ_i); and (c) C̄_i > max_{τ_k ∈ ip(x,i)−{τ_i}}(C̄_k).

(1)  N^0 ← 0; N^1 ← 1; k ← 1
(2)  R ← R_i^int(x, N^0, N^1)
(3)  while k < N_E ∧ Γ is schedulable do
(4)      R^0 ← R_i^int(x, N^0 + 1, N^1)
(5)      R^1 ← R_i^int(x, N^0, N^1 + 1)
(6)      if (R^0 > R^1) then
(7)          R ← R^0
(8)          N^0 ← N^0 + 1
(9)      else
(10)         R ← R^1
(11)         N^1 ← N^1 + 1
(12)     endif
(13)     k ← k + 1
(14) endwhile
(15) if Γ is schedulable then
(16)     // R is the solution for R_i^int(x, N_i^1, N_i^1)
(17)     // where N_i^1 = N^0 and N_i^1 = N^1
(18) else
(19)     // Task set cannot cope with N_E errors
(20) endif
```

Fig. 3. Procedure to determine the values of N_i^0 and N_i^1

Initially, $\mathsf{N}^0 = 0$ and $\mathsf{N}^1 = 1$. The initial value of N^1 accounts for the first assumed internal error. Then, in each iteration, either N^0 or N^1 is increased by 1 depending on which one makes the value of $R_i^{int}(x, \mathsf{N}^0, \mathsf{N}^1)$ bigger. The strategy of increasing the number of errors by one at each time is for the sake of performance. Indeed, equations (6) and (7) are monotonically non-decreasing in function of the number of errors and they are also solved iteratively. Hence, the initial value to solve them for a particular choice of N^0 and N^1 can be the solutions obtained in the previous iteration. For example, let the values of $R_i^{int}(x, 1, 1)$ and $R_i^{int}(x, 0, 2)$ be calculated in iteration it, say. If the task set is schedulable and $\mathsf{it} < N_E$, either $R_i^{int}(x, \mathsf{N}^0 + 1, \mathsf{N}^1)$ or $R_i^{int}(x, \mathsf{N}^0, \mathsf{N}^1 + 1)$ will be computed in the $(\mathsf{it} + 1)^{th}$ iteration. If so, such a computation can start from the previously computed values in iteration it (variable R in the algorithm). The implementation details to do so are not explicitly expressed in the algorithm of Fig. 3 but it can easily be carried out by integrating the algorithm with the iterative procedure that solves equations (6) and (7).

Clearly, the computational effort to calculate $R_i^{int}(x, N_i^0, N_i^1)$ is higher when compared to the computation of $R_i^{ext}(x, N_E)$. Therefore, it is important to observe some aspects related to the need for performing the algorithm of Fig. 3. Indeed, one only needs to calculate $R_i^{int}(x, N_i^0, N_i^1)$ if the following conditions hold:

Table 2. An illustrative task set and the values of worst-case response times (in bold)

Task set					$N_E = 2$ $\langle 0,0,2\rangle$	
Task	T_i	C_i	$\overline{C}_i$	D_i	R_i^{int}	R_i^{ext}
τ_1	13	2	2	13	–	**12**
τ_2	25	3	4	25	–	**17**
τ_3	30	5	5	30	**21**	20

(a) $\forall \tau_i \in \Gamma$: $R_i^{ext}(x, N_E) \leq D_i$; (b) $\mathsf{pr}(\overline{\tau}_i) > \mathsf{pr}(\tau_i)$; and (c) $\overline{C}_i > \max_{\tau_k \in \mathsf{ip}(x,i)-\{\tau_i\}} (\overline{C}_k)$. If condition (a) or (c) does not hold, the computation of $R_i^{int}(x, N_i^0, N_i^1)$ is irrelevant because either the task set is already unschedulable or by lemma 1 it is known that $R_i(x, N_E) = R_i^{ext}(x, N_E)$. Moreover, should $\mathsf{pr}(\overline{\tau}_i)$ equal $\mathsf{pr}(\tau_i)$, any solution of equation (5) can be used. For example, $N_i^1 = 0$ and $N_i^1 = N_E$. This is because under this condition $\mathsf{ipe}(x,i) = \mathsf{ip}(x,i)$.

Should the values of N_i^0 and N_i^1 be determined by the algorithm of Fig. 3 for some task $\tau_i \in \Gamma$, its worst-case response time is given by

$$R_i(x, N_i^0 + N_i^1) = \max\left[R_i^{ext}(x, N_i^0 + N_i^1), R_i^{int}(x, N_i^0, N_i^1)\right] . \tag{9}$$

Otherwise, it is given simply by taking $R_i(x, N_E) = R_i^{ext}(x, N_E)$.

5.6 An Illustrative Example

Consider the task set in Table 2. This task set, with three tasks, is the same as that given in Table 1 but with $\overline{C}_2 = 4$ time units. Assume that the priority of primary tasks are given according to DM and let $P_x\langle 0,0,2\rangle$ and $N_E = 2$. The values of $R_i^{ext}(x,2)$ for each of the three tasks are iteratively calculated by equation (4), similarly to the explanation given in Section 4. The found values are 12, 17 and 20, as indicated in the table.

As can be seen, considering only external errors, the task set is schedulable. However, as $\overline{C}_3 > \max_{\tau_k \in \mathsf{ip}(x,3)-\{\tau_3\}}(\overline{C}_k)$, $R_3^{int}(x, N_3^0, N_3^1)$ needs to be computed since $R_3^{ext}(x, N_E)$ may not give the worst-case response time. In order to do so, the algorithm of Fig. 3 is performed. Firstly, the values of $R_i^{int^1}(x,1)$ and $R_i^{int^0}(x,0,1)$ are calculated (observe that $\mathsf{sp}(x,3) = \emptyset$). We have that $r_3^{int^{1^0}}(x,1) = r_3^{int^{1^1}}(x,1) = 5$ and so $R_3^{int^1}(x,1) = 5$. The value of $R_3^{int^0}(x,0,1)$ equals 10:

$$\begin{aligned}
r_3^{int^{0^0}}(x,0,1) &= 5\,,\\
r_3^{int^{0^1}}(x,0,1) &= 5 + \left\lceil \frac{5}{13} \right\rceil 2 + \left\lceil \frac{5}{25} \right\rceil 3 = 10\,,\\
r_3^{int^{0^2}}(x,0,1) &= 5 + \left\lceil \frac{10}{13} \right\rceil 2 + \left\lceil \frac{10}{25} \right\rceil 3 = 10\,.
\end{aligned}$$

Consequently, $R_3^{int}(x,0,1) = 15$. Then, the iterative procedure starts (lines 3-14 of Fig. 3), where the values of both $R_3^{int}(x,1,1)$ and $R_3^{int}(x,0,2)$ are computed.

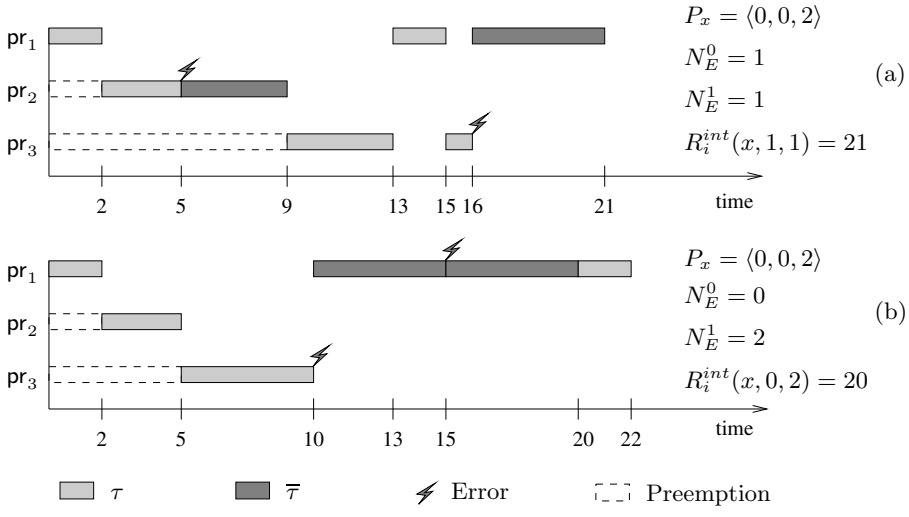

Fig. 4. Two possible fault scenarios for task τ_3 and $N_E = 2$

First, consider $R_3^{int}(x, 1, 1)$, which is obtained by equation (8), *i.e.* $R_3^{int^0}(x, 1, 1) + R_3^{int^1}(x, 1)$. $R_3^{int^1}(x, 1) = 5$ has been computed earlier. The computation of $R_3^{int^0}(x, 1, 1)$ is as follows. Since $R_3^{int^0}(x, 0, 1) = 10$, $r_3^{int^{0^0}}(x, 1, 1) = 10$. Carrying out the iterative procedure, one can see that $R_3^{int^0}(x, 1, 1) = 16$ since

$$r_3^{int^{0^2}}(x, 1, 1) = r_3^{int^{0^3}}(x, 1, 1) = 5 + \left\lceil \frac{16}{13} \right\rceil 2 + \left\lceil \frac{16}{25} \right\rceil 3 + 1 \times 4 = 16\,.$$

Thus, $R_3^{int}(x, 1, 1) = R_3^{int^0}(x, 1, 1) + R_3^{int^1}(x, 1) = 21$. Then, $R_3^{int}(x, 0, 2)$ is computed. The value of $R_3^{int^1}(x, 2)$ is equal to 10 since $r_3^{int^{1^0}}(x, 2) = r_3^{int^{1^1}}(x, 2) = 5 + (2-1)5 = 10$. Also, starting from $r_3^{int^{0^0}}(x, 0, 2) = 10$ since $R_3^{int^0}(x, 0, 1) = 10$, one will find that $R_3^{int^0}(x, 0, 2) = 10$. Hence, $R_3^{int}(x, 0, 2) = R_3^{int^0}(x, 0, 2) + R_3^{int^1}(x, 2) = 20$.

The algorithm stops after the first iteration since $N_E = 2$. As can be seen, the worst-case scenario for τ_3 with $P_x = \langle 0, 0, 2 \rangle$ and $N_E = 2$ is when one error hits τ_2 before the internal error in τ_3. This is because when $N_i^0 = 1$, τ_3 suffers interference from an extra release of τ_1. Fig. 4 illustrates this behaviour, where the two scenarios are presented. Scenario (a) represents the worst case and scenario (b) is when both errors are internal.

6 Priority Assignment and Evaluation

Once one can find the worst-case response time of each task given N_E errors, there is another problem to be solved: determining which priority configuration

P_x leads to the maximum value of N_E without making any task missing its deadline. Our approach to solving this problem is iterative. For the sake of space, only the general idea of the priority configuration search algorithm is presented here. The algorithm is very similar to the one previously published [15]. More details can be found in this or in other publication [14].

6.1 Searching for the Priority Configuration

Let $N_e(x)$ denote the maximum N_E that the task set can cope with in priority configuration P_x. $N_e(x)$ can be implemented as a binary search using the equations previously described in Section 5. The interval of the search can be set to $[0, \min(1, \lfloor \frac{D_i - C_i}{\overline{C}_i} \rfloor)]$, for example. Clearly, no task τ_i can cope with more than $\frac{D_i - C_i}{\overline{C}_i}$ (in the worst case). Note that by definition, no task set is schedulable in P_x if $N_e(x) + 1$ errors take place. The sketch of the algorithm is as follows:

1. Initially, let $P_x = \langle 0, 0, \ldots, 0 \rangle$ and let $N_E = N_e(x)$.
2. Repeat
   (a) Compute $R_i(x, N_E)$, $i = 1, \ldots, n$.
   (b) If the task set is schedulable, make $N_E = N_E + 1$. Save the value of P_x.
   (c) Otherwise
      i. If there is some unschedulable task due to external errors, stop searching priority configuration.
      ii. Look for the lowest priority task $\tau_j \in \mathsf{sp}(x, i)$ so that

$$\left\lceil \frac{R_i^{int}(x, N_i^0, N_i^1)}{T_j} \right\rceil > \left\lceil \frac{R_i^{int^0}(x, N_i^0, N_i^1)}{T_j} \right\rceil . \tag{10}$$

      iii. If there is such τ_j, make $\mathsf{pr}(\overline{\tau}_i) = \mathsf{pr}(\tau_j)$ and $N_E = \max(N_E, N_e(x))$. Otherwise stop searching priority configuration.
3. The last saved priority configuration is returned.

Once $N_e(x)$ is determined, the goal of the algorithm is to find out another priority configuration by raising the priority of some alternative task so that the task set is schedulable with $N_E = N_e(x) + 1$. To do so, one needs to decrease the values of $R_i^{int}(x, N_e(x) + 1)$ by increasing priorities of their alternative tasks if $R_i^{int}(x, N_e(x) + 1) > D_i$. However, these priorities only need to be raised if it is possible to decrease the number of preemption of higher priority tasks τ_j without making them unschedulable. In other words, if there is a task $\tau_j \in \mathsf{sp}(x, i)$ which is executed over the period of time $R_i^{int^1}(x, N_i^0, N_i^1)$ - see condition (10) - then making $\mathsf{pr}(\overline{\tau}_i) = \mathsf{pr}(\tau_j)$ will decrease the worst-case response time of τ_i due to internal errors. Carrying out the described procedure until it is no longer possible to decrease the values of $R_i^{int}(x, N_e(x) + 1) > D_i$, one will find the best priority configuration for fault tolerance purposes [14,15].

6.2 Results of Experiments

Some experiment results are shown in Fig. 5. These results were collected from carrying out the approach for a large number of task sets. The task sets, with 10 tasks each, were randomly generated as follows. The values of worst-case computation time were generated according to an exponential distribution with mean $U/10$, where U is the processor utilisation. The periods and deadlines of tasks were assigned according to a uniform distribution with minimum and maximum values set to 50 and $5,000$, respectively. Deadlines were allowed to

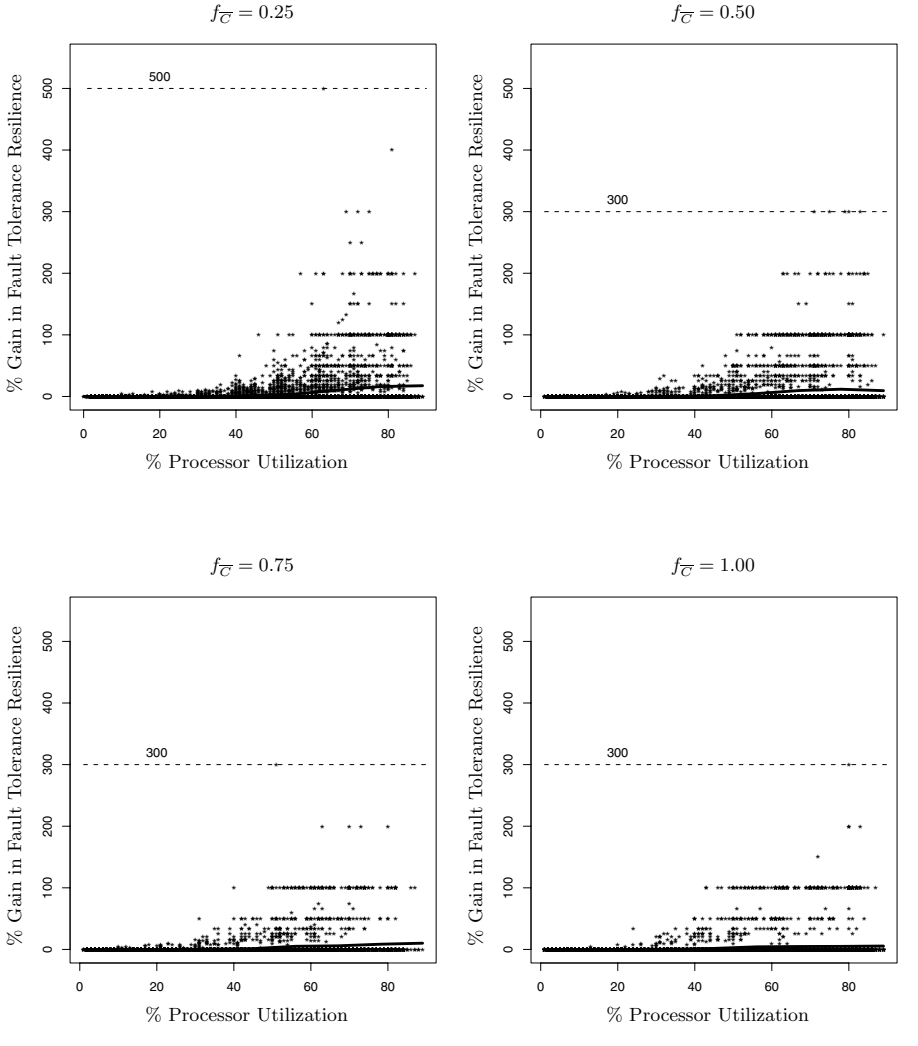

Fig. 5. Improvement in terms of fault resilience measured as obtained increase in N_E

be less than or equal to periods. We used the deadline monotonic algorithm to assign the priorities of primary tasks. We did not consider processor utilisation higher than 0.9 since it is difficult to guarantee the schedulability of the task set under error occurrences (*i.e.* most of the time it is not possible to tolerate even one fault at these higher processor utilisations).

Each one of the graphs represents the obtained gains in terms of increasing in N_E for 18,000 task sets. For each of the four graphs, a different value of recovery factor ($f_{\overline{C}}$) was used. The recovery factor is an input parameter of the simulation and was used to bound the worst-case execution time of the alternative tasks so that the values of $\overline{C}_i$ were generated according to a uniform distribution between 1 and $f_{\overline{C}}C_i$. For example, if $f_{\overline{C}} = 1$, $\overline{C}_i \leq C_i$.

As can be seen from Fig. 5, the gains obtained in terms of fault resilience as measured by N_E can be significant (up to 500% for $f_{\overline{C}} = .25$) and, as expected, is higher for lower values of $f_{\overline{C}}$. We can also observe that there were lower gains for lower processor utilisations. This can be explained by the fact that in these cases there is higher spare time available. This spare time can be used to carry out fault tolerance assuming higher values of N_E. Promoting the priority of alternative tasks for these cases, therefore, has lower impact in fault resilience since it is already high.

As illustration, consider Table 3, which shows the values of the worst-case response time due to external and internal errors of each task of a task set collected from the simulation. The values of the worst-case response time are in bold. It is worth emphasising the fact that in practice one does not need to perform algorithm 3 to compute the values of worst-case response time due to internal errors for all tasks. This is because of the reasons mentioned in Section 5. For example, for $P_x = \langle 0, 0, \ldots, 0 \rangle$ making $N_i^0 = 0$ and $N_i^1 = N_E$ for all tasks suffices. Also, for $P_x = \langle 0, 0, \ldots, 0, 9 \rangle$ the algorithm 3 only needs to be carried out with respect to τ_{10} (by lemma 1). Nevertheless, all values of $R_i^{int}(x, N_i^0, N_i^1)$ are shown in the table for the sake of illustration.

Table 3. Illustration of the improvement in fault tolerance resilience

Task set					$P_x = \langle 0, 0, \ldots, 0 \rangle$ $N_e(x) = 1$		$P_x = \langle 0, 0, \ldots, 0, 9 \rangle$ $N_e(x) = 3$	
Task	T_i	C_i	$\overline{C}_i$	D_i	R_i^{int}	R_i^{ext}	R_i^{int}	R_i^{ext}
τ_1	4016	205	81	4011	**286**	205	448	**1303**
τ_2	4056	304	84	4031	**593**	590	761	**1607**
τ_3	4279	528	46	4034	1083	**1121**	1251	**2135**
τ_4	4363	99	88	4042	**1224**	1220	1400	**2234**
τ_5	4980	9	1	4061	1146	**1233**	1322	**2243**
τ_6	4164	17	2	4138	1164	**1250**	1340	**2260**
τ_7	4341	181	96	4197	**1439**	1431	1631	**2441**
τ_8	4518	90	49	4273	1482	**1529**	1674	**2531**
τ_9	4487	136	112	4305	**1681**	1665	1905	**2267**
τ_{10}	4643	1768	366	4490	**3703**	3449	**4435**	3673

7 Conclusion

In this paper we have presented an approach to increasing the fault tolerance resilience of hard real-time task sets in the context of fixed priority scheduling. The priorities of tasks are determined off-line so that the system can cope with more errors during their execution. To do so, a new framework to analyse the system under fault scenarios and an algorithm to search for the best priority configuration were derived. Both the analysis and priority configuration search algorithm were an extension of our former work [15], which assumed that there is a known minimum time interval between consecutive errors. Here this assumption was removed.

The proposed approach was extensively evaluated by simulation. Results from the experiments indicate that there are benefits in applying our approach. Indeed, for some cases significant gains (up to 500%) in terms of fault resilience was obtained for some cases.

The approach described in this paper takes into consideration the worst-case scenario to derive the priority assignments. It would be interesting to investigate ways of varying priorities dynamically in order to take advantage of spare capacities in the system. This will be part of our future work.

References

1. N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. "Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling". *Software Engineering Journal*, 8(5):284–292, 1993.
2. A. A. Bertossi, L. V. Mancini, and F. Rossini. "Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems". *IEEE Transaction on Parallel and Distributed Systems*, 10(9):934–945, 1999.
3. A. Burns, R. I. Davis, and S. Punnekkat. "Feasibility Analysis of Fault-Tolerant Real-Time Task Sets". In *Proc. of the Euromicro Real-Time Systems Workshop*, pages 29–33. IEEE Computer Society Press, 1996.
4. A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 3rd edition, 2001.
5. M. Caccamo and G. Buttazzo. "Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems". In *Proc. of the 5th Conference on Real-Time Computing and Applications (RTCSA)*, pages 223–231, 1998.
6. S. Ghosh, R. G. Melhem, and D. Mossé. "Enhancing Real-Time Schedules to Tolerate Transient Faults". In *Proc. of the 16th Real-Time Systems Symposium (RTSS)*, pages 120–129. IEEE Computer Society Press, 1995.
7. S. Ghosh, R. G. Melhem, D. Mossé, and J. S. Sarma. Fault-Tolerant Rate-Monotonic Scheduling. *Real-Time Systems*, 15(2):149–181, 1998.
8. N. Kandasamy, J. P. Hayes, and B. T. Murray. "Tolerating Transient Faults in Statically Scheduled Safety-Critical Embedded Systems". In *Proc. of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 212–221, 1999.
9. R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. "The MAFT Architecture for Distributed Fault Tolerance". *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

10. H. Kopetz. *Real-Time Systems Design for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
11. C. M. Krishna and K. G. Shin. "On Scheduling Tasks with a Quick Recovery from Failure". *IEEE Transactions on Computers*, 35(5):448–455, 1986.
12. F. Liberato, R. G. Melhem, and D. Mossé. "Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems". *IEEE Transactions on Computers*, 49(9):906–914, 2000.
13. L. Liestman and R. H. Campbell. "A Fault-Tolerant Scheduling Problem". *IEEE Transaction on Software Engineering*, 12(11):1089–1095, 1986.
14. G. M. A. Lima. *"Fault Tolerance in Fixed-Priority Hard Real-Time Distributed Systems"*. PhD thesis, Deptartment of Computer Science, University of York, 2003.
15. G. M. A. Lima and A. Burns. "An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault Tolerant Hard Real-Time Systems". *IEEE Transaction on Computers*, 52(10):1332–1346, 2003.
16. S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.
17. S. Poledna, A. Burns, A. J. Wellings, and P. Barrett. "Replica Determinism and Flexible Scheduling in Hard Real-time Dependable Systems". *IEEE Transsactions on Computers*, 49(2):100–111, 2000.
18. S. Punnekkat. *"Schedulability Analysis for Fault Tolerant Real-Time Systems"*. PhD thesis, Department of Computer Science, University of York, 1997.
19. S. Ramos-Thuel and J. K. Strosnider. "The Transient Server Approach to Scheduling Time-Critical Recovery Operations". In *Proc. of the 12th Real-Time Systems Symposium (RTSS)*, pages 286–295. IEEE Computer Society Press, 1991.
20. F. Schneider. "Replication Management Using the State-Machine Approach". In Sape Mullender, editor, *Distributed Systems*, chapter 7. Addison-Wesley, 2nd edition, 1993.
21. P. Veríssimo and L. Rodrigues. *Distributed Systems for Systems Architects*. Kluwer Academic Publishers, 2001.