An Effective Schedulability Analysis for Fault-Tolerant Hard Real-Time Systems *

George M. de A. Lima† Alan Burns

Real-Time Systems Research Group
Department of Computer Science
University of York, England
{gmlima,burns}@cs.york.ac.uk

Abstract

We propose worst-case response time schedulability analysis for fault-tolerant hard real-time systems which takes into account the effects of temporary faults. The major contribution of our approach is to consider the recovery of tasks running with higher priorities. This characteristic is very useful since faulty tasks certainly have a shorter period of time to meet their deadlines. Due to its flexibility and simplicity, the proposed approach provides an effective schedulability analysis, where system predictability can be fully guaranteed.

1. Introduction

The ability to tolerate faults is an extremely desired characteristic for hard real-time systems. Nevertheless, the implementation of such systems is far from being simple. It involves both suitable schedulability analysis, which takes the effects of possible faults into account, and fault tolerance mechanisms, which keep the system computation complying with the specification even in the presence of faults. In other words, tasks that may be performed by the fault tolerance mechanism have to be taken into account by the schedulability analysis.

Due to the required fault tolerance/schedulability analysis synergy and the uncertain nature of faults, most papers published up to date have proposed restrictive and/or *ad hoc* solutions to the schedulability analysis. As a result, those solutions impair the applicability and flexibility of the analysis. By contrast, a recent approach [3, 13] based on the well known response time analysis [1] has eliminated these drawbacks by not restricting the way that fault tolerance is carried out and by assuming realistic fault and task models. In this work we generalise this approach by allowing task recovery to execute with higher priority, which increases the flexibility of the analysis. This can be very useful since, after a fault, tasks certainly have a shorter period of time to meet their deadlines. Indeed, data collected from simulation has illustrated that significant gains in terms of reduction of recovery time may be obtained using our approach.

As in [3], we are concerned in this work only with temporary faults. This kind of fault originates either from some disturbances in the external environment (*transient* faults) or from the presence of rarely occurring combinations of conditions in respect of the internal state (*intermittent* faults) [8]. Indeed, since temporary faults are usually more frequent than *permanent faults* [12] and the tolerance of permanent faults requires hardware redundancy (which increase dramatically the cost and complexity of its implementation), a system that tolerates temporary faults is usually more cost-effective.

The remainder of this paper is organized as follows. The next section presents the assumed computational model and introduces our notation. Section 3 describes briefly some concepts regarding the use of response time analysis for fault tolerance purposes. The main result of this paper is presented in section 4. Section 5 describes a simple evaluation of our solution based on simulation. Our approach is compared with related work in section 6. Finally, section 7 offers our conclusion.

*Published in the Proc. of the 13th Euromicro Conference on Real-Time Systems (ECRTS). Delft, The Netherlands. June 2001

†Supported by CAPES/Brazil under grant: BEX1438/98-0.

2. Computational 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 faults. Any primary task τ_i in Γ has a period, T_i, a deadline D_i, and a worst-case computation time, C_i ($D_i \leq T_i$). Tasks can be periodic or sporadic. For sporadic tasks the period means the minimum inter-arrival time. Each primary task τ_i can have m_i *alternative tasks* associated with it, i.e. for all τ_i there exists $\gamma_i = \{\overline{\tau}_i^1, \ldots, \overline{\tau}_i^{m_i}\}$ ($m_i \geq 1$). Each alternative task $\overline{\tau}_i^k$ corresponds to a given action taken to recover τ_i from a given fault. Any alternative task $\overline{\tau}_i^k$ has a worst-case computation time denoted by $\overline{C}_i^k$. We define $\overline{\Gamma} = \gamma_1 \cup \gamma_2 \cup \ldots \cup \gamma_n$. We denote $\overline{\tau}_i$ as the alternative task of τ_i whose worst-case computation time, $\overline{C}_i$, is the largest one, i.e. $\overline{C}_i = max(\overline{C}_i^k)$ ($1 \leq k \leq m_i$). For simplicity we assume that any alternative task in γ_i runs at the same priority level. This assumption is not a restriction of our model and it is useful since we do not need to include unnecessary details in the description we shall present. Thus we only need to refer to $\overline{\tau}_i$ as the worst-case alternative task in case of faults in task τ_i.

Primary tasks are scheduled according to some fixed priority assignment algorithm (e.g. deadline monotonic), which attributes a distinct priority to each task τ_i in Γ. We consider n different priority levels $(1, 2, \ldots, n)$, where 1 and n represent the highest and the lowest priority levels, respectively. Alternative tasks are assumed to execute with the same or higher priorities than their respective primary tasks. We denote the priority of τ_i and $\overline{\tau}_i$ as $\mathrm{pr}(\tau_i)$ and $\mathrm{pr}(\overline{\tau}_i)$, 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. For the simplicity of notation we define two subsets of Γ (others will be defined when necessary): hp(i) represents the tasks whose priorities are greater than τ_i's priority and hpe(i) denotes those tasks whose priorities are greater than or equal to τ_i's priority (hpe(i) = hp(i) $\cup \{\tau_i\}$).

We consider only the occurrence of temporary software faults in a uniprocessor system where recovery techniques such as *recovery blocks*, *exception handling* or *task re-executions* are appropriate [3]. When a fault hits a task, say τ_i, the system must schedule an appropriate alternative task, say $\overline{\tau}_i^k$ ($1 \leq k \leq m_i$), which has to finish by D_i. If other faults hit $\overline{\tau}_i$ during its execution, we assume that the recovery of these successive faults is carried out by re-executions of $\overline{\tau}_i$. We also assume that there is no cost associated with any scheduling of primary or alternative tasks. Also, we assume that all faults are detected by the system and there is no fault propagation (i.e. faults affect only the executing task).

The assumed fault model considers that the fault arrival rate is bounded (i.e there is a minimum interval between two consecutive faults). We denote T_F as the minimum assumed time between faults. Also, we denote the minimum *allowed* time between faults as T_f. This means that any $T_F < T_f$ makes the task set unscheduled by the analysis.

3. Background and Motivation

Briefly, the worst-case standard response time (without considering faults), R_i, of each task τ_i in Γ is calculated by an iterative procedure, whose equation has the form below [1].

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

where the sum term is the interference due to preemption by higher priority tasks. Also, for simplicity, the worst-case blocking time and jitter effects are considered to be null since they are not relevant for the description of the analysis we shall present.

When faults are being considered, extra interference due to fault effects have to be taken into account. This interference represents the worst-case interference of any extra processing performed by alternative tasks when primary tasks fail. In order to derive the expression to take this interference into account, we have to know the maximum number of faults that may take place over the period R_i and the worst-case cost of recovery. As in the worst case the time between faults is T_F, the maximum number of faults over the period R_i is given by $\lceil R_i/T_F \rceil$.

When a fault takes place, say at time t, it may hit any task that is potentially executing at t. Assume that task τ_k is hit by a fault at time t. Thus, any task τ_i ($\tau_k \in$ hpe(i)) will suffer extra interference due to the recovery of τ_k (i.e. $\overline{C}_k$ because in the worst case τ_k fails just before its completion). The longest interference over τ_i happens when $\overline{C}_k$ is maximum among all tasks in hpe(i). Therefore, in the worst case, the response time can be given by

$$R_i = C_i + \sum_{\tau_j \in \mathrm{hp(i)}} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \left\lceil \frac{R_i}{T_F} \right\rceil \max_{\tau_k \in \mathrm{hpe(i)}} (\overline{C}_k) \tag{2}$$

The above equation, derived in [3, 4], is valid only when primary tasks and their respective alternative tasks run with the same priority. However, it would be advantageous in certain situations to give higher priorities to alternative tasks since a faulty task certainly has less time to meet its deadline. As an illustration, consider table 1, where $T_F = 10$ time units, primary task priorities are according to the deadline monotonic algorithm and $\mathrm{pr}(\tau_i) = \mathrm{pr}(\overline{\tau}_i)$ for all τ_i in Γ. As can be seen, τ_3 is considered unschedulable by the analysis carried out by equation (2). On the other hand, it is interesting to note that there is a considerable amount of slack time in higher priority levels that could be used to execute $\overline{\tau}_3$.

Table 1. An unschedulable task set ($\mathbf{T_F} = 10$).

Task	T_i	C_i	$\overline{C}_i$	D_i	R_i
τ_1	13	2	2	13	4
τ_2	25	3	3	25	8
τ_3	30	5	5	30	37

In the sections below we derive a generalisation of equation (2) allowing the priorities of alternative tasks to be raised. The goal of this generalisation is to take advantage of the slack time in higher priority levels to allow task recovery to be carried out.

4. Raising Priorities of Alternative Tasks

As we have seen, it would be advantageous in certain situations to give higher priorities to alternative tasks. In this section we derive an analysis which considers this characteristic. Firstly, we introduce the effects that should be taken into account by the analysis (section 4.1) when alternative tasks execute at a higher priority level. Then, in section 4.2, we derive the equations of the analysis and in section 4.3 some simple but useful properties of the analysis are presented. Finally, using a simple example, we illustrate one of the advantages of using our approach.

4.1. Effects of Raising Priorities upon Faults

In order to understand the effects of raising priorities upon faults consider the set of two tasks in figure 1. Task τ_j has higher priority than τ_i. Suppose that a fault hits τ_i just before the completion of its execution. As can be seen from the figure, $\overline{\tau}_i$ is then selected to execute with higher priority than τ_j. As a result, the response time of τ_j is increased by $\overline{C}_i$ and the response time of τ_i is decreased by C_j. These effects, which are not present when alternative and primary tasks have the same priority, have to be taken into account by the response time analysis.

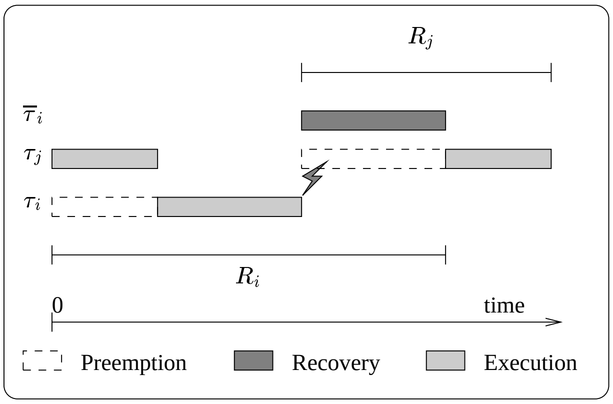

Figure 1. Worst-case scenario for task τ_j

In addition to this, it is important to realize that the worst-case scenario cannot be represented simply by taking the task with the longest recovery time as in equation (2). For example, consider figure 2, which represents a different execution scenario for the same tasks presented in figure 1. Consider that $\overline{C}_j < \overline{C}_i$ and that a fault hits τ_j instead of τ_i. This situation, as can be noted from figure 2, leads to a longer response time for task τ_i when compared to figure 1. This is because task τ_i suffers not only the interference $\overline{\tau}_j$ but also the interference of another activation of task τ_j.

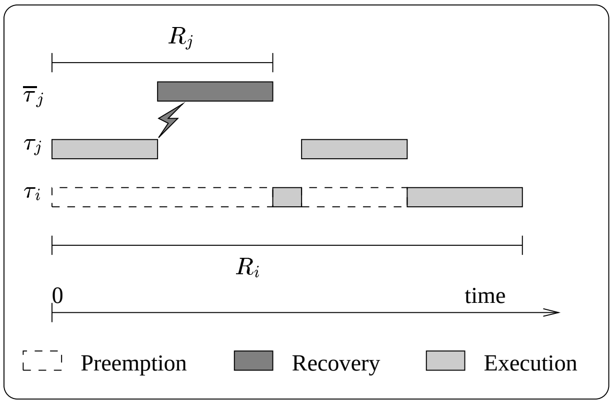

Figure 2. Worst-case scenario for task τ_i

Summing up, the worst-case scenario is no longer determined only by observing the time at which all tasks are released at once. Indeed, this scenario is also related to fault occurrences. For example, from the above figures it can be noted that the worst case for task τ_i is when it is released at the same time as task τ_j and the fault hits τ_j just before its execution. By contrast, the worst case for task τ_j is when it is released at the same time as task τ_i is hit by a fault.

In order to generalise the characteristics illustrated in figures 1 and 2 consider a set of tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ and a given task in Γ, say τ_i. We can define the following subsets of Γ with respect to τ_i:

- intOn(i). These are the tasks that may interfere on τ_i's response time if a fault occurs. More formally:

  $$\text{intOn(i)} = \{\tau_j \,|\, \exists\, \overline{\tau}_j \in \gamma_j\,,\ \text{pr}(\overline{\tau}_j) \geq \text{pr}(\tau_i)\}$$

  Note that all tasks in hpe(i) are in intOn(i) since we are assuming that alternative tasks run with priorities greater than or equal to their respective primary tasks.

- intLess(i). Tasks that belong to such a subset do not suffer any extra interference when faults hit τ_i. This is because their priorities are higher than $\text{pr}(\overline{\tau}_i)$:

  $$\text{intLess(i)} = \{\tau_j \,|\, \forall\, \overline{\tau}_i \in \gamma_i\,,\ \text{pr}(\tau_j) > \text{pr}(\overline{\tau}_i)\}$$

Figure 3 illustrates the relation among these subsets with regard to the priority level of task τ_i.

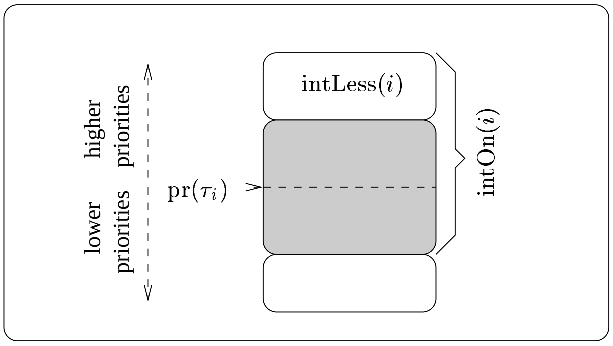

Figure 3. Γ subsets with respect to task τ_i

Note from figure 3 that τ_i does not suffer any interference from tasks in $\Gamma - \text{intOn(i)}$. Thus, when calculating the response time of τ_i, we have to consider only faults hitting tasks in intOn(i).

4.2. The Analysis

As figures 1 and 2 suggest, there are two cases to be considered for analysing the response time of task τ_i: when task τ_i may be hit by faults and when faults hit tasks in $\text{intOn(i)} - \{\tau_i\}$. In the former case, the response time of τ_i is the time necessary for executing both τ_i and its recovery (i.e. $\overline{\tau}_i$). We denote this time as Rc_i, the *worst-case recovery time* of τ_i. By contrast, in the latter case, which we call the *worst-case interference time* on τ_i and denote as Rf_i, the response time of τ_i is determined by the time necessary for executing all tasks in hpe(i) and the recovery of tasks in $\text{intOn(i)} - \{\tau_i\}$. Therefore, the worst-case response time, R_i, is the maximum of Rc_i and Rf_i:

$$R_i = \max(Rc_i, Rf_i) \tag{3}$$

Below we describe respectively the equations to calculate Rc_i and Rf_i.

Worst-Case Recovery Time

If no fault takes place during the execution of τ_i, its response time is given by the standard response time analysis. Let Rc_i^0 represent this time:

$$Rc_i^0 = C_i + \sum_{\tau_j \in \text{hp(i)}} \left\lceil \frac{Rc_i^0}{T_j} \right\rceil C_j \tag{4}$$

Consider that only one fault takes place just before the completion of the execution of τ_i. In this case τ_i needs extra time to execute its alternative task. Let us denote this extra time Rc_i^1. In other words, Rc_i^1 is the time interval from Rc_i^0 to the end of the execution of $\overline{\tau}_i$.

From the start of its execution until Rc_i^0, τ_i suffers preemption due to the execution of all tasks in hp(i). However, from Rc_i^0 to Rc_i^1, τ_i suffers only interference due to the tasks in intLess(i) since $\overline{\tau}_i$ is assumed to be executed at a higher priority level. The figure below illustrates this, where we can note two phases to calculate the worst-case recovery time of τ_i, before and after the fault.

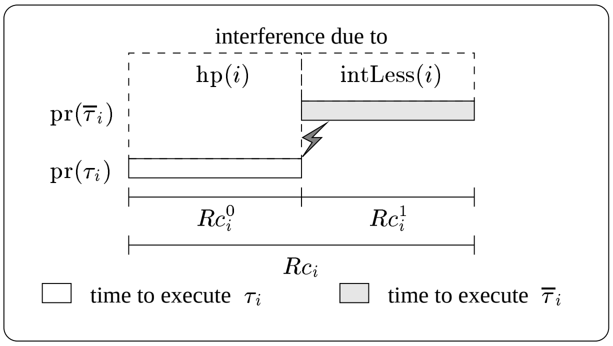

Figure 4. Effects of a single fault during R_i

If more than one fault occurs, the worst case is when all faults hit the task with the maximum recovery cost among all tasks in intLess(i)$\cup\{\tau_i\}$ just before the completion of its execution. Lower priority tasks are not considered since they only start executing after the completion of $\overline{\tau}_i$.[1] Notice that the procedure for calculating Rc_i^1 does not affect the value of Rc_i^0.

Figure 5 illustrates a situation where three faults take place during Rc_i. The first fault interrupts the execution of τ_i just before its completion. Then, $\overline{\tau}_i$ is selected to execute at a higher priority level. The other two faults may hit any task in intLess(i) $\cup\{\tau_i\}$. Therefore, instead of considering only $\overline{C}_i$ as in figure 4, we have to consider $\overline{C}_i$ plus $(N_i-1)\overline{C}_j$, where N_i is the maximum number of faults that may occur and $\tau_j \in$ intLess(i) $\cup\{\tau_i\}$ has the maximum recovery cost.

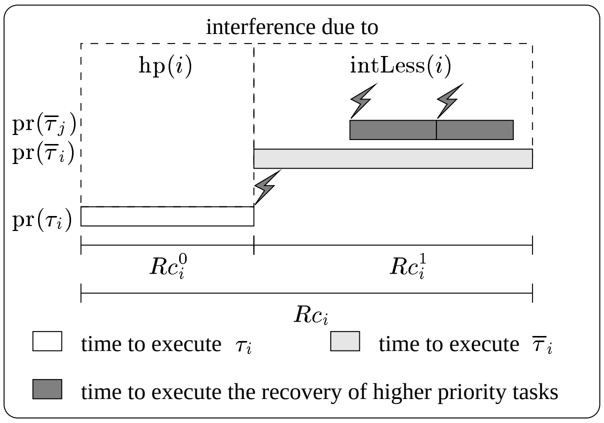

Figure 5. Effects of multiple faults during R_i

As intLess(i) $\subseteq$ hp(i), when calculating Rc_i^1 we have to take care not to compute the interference already taken by Rc_i^0. Hence considering that N_i faults may occur over the response time of τ_i, in the worst case Rc_i^1 can be obtained as follows.

$$\begin{aligned} Rc_i^1 &= \overline{C}_i + (N_i-1) \max_{\tau_j \in \text{intLess(i)} \cup \{\tau_i\}} (\overline{C}_j) + \\ &\sum_{\tau_k \in \text{intLess(i)}} \left(\left\lceil \frac{Rc_i^0 + Rc_i^1}{T_k} \right\rceil - \left\lceil \frac{Rc_i^0}{T_k} \right\rceil \right) C_k \end{aligned} \tag{5}$$

The complete recovery time of τ_i, Rc_i, can be then simply derived by taking the sum of Rc_i^0 and Rc_i^1:

$$Rc_i = Rc_i^0 + Rc_i^1 \tag{6}$$

The maximum number of faults that may occur over the whole recovery time is given as follows.

$$N_i = \left\lceil \frac{Rc_i}{T_F} \right\rceil \tag{7}$$

[1] If tasks may be suspended, one has to consider intOn(i) instead of intLess(i).

Notice that dividing the calculation of Rc_i into two phases is only a practical way to derive its value. It is important to realize that this does not imply that we are assuming that the first fault only occurs after Rc_i^0 time units.

Worst-Case Interference Time

The worst-case interference time, Rf_i, of a task, say τ_i, is determined by the interference due to the preemption of higher priority tasks (hp(i)) plus the interference due to preemption caused by the recovery of faulty tasks in intOn(i) $-\{\tau_i\}$. The former interference can be calculated simply by the standard response time analysis (equation (1)). The interference due to faults is caused by the maximum of the number of faults multiplied by the cost of recovery, which depends on the priority of the faulty task. This is because the maximum number of faults that may occur is determined either by $\lceil Rf_i/T_F \rceil$ when $Rf_i \geq Rc_k$ or by $\lceil Rc_k/T_F \rceil$ when $Rf_i < Rc_k$, where τ_k is the faulty task and belongs to intOn(i) $-\{\tau_i\}$. Note that Rc_k may be greater than Rf_i only when the pr$(\tau_k) <$ pr(τ_i). The following equation gives the value of Rf_i.

$$\begin{aligned} Rf_i &= C_i + \sum_{\tau_j \in \text{hp(i)}} \left\lceil \frac{Rf_i}{T_j} \right\rceil C_j + \\ &\max_{\tau_k \in \text{intOn(i)} - \{\tau_i\}} \left(\left\lceil \frac{\max(Rc_k, Rf_i)}{T_F} \right\rceil \overline{C}_k \right) \end{aligned} \tag{8}$$

4.3. Some Properties of the Analysis

In this section we, most informally, state some simple and quite intuitive properties of the analysis presented in the earlier sections. The main goal of emphasising these properties here is to characterise the situations in which it is worth raising alternative task priorities and also to prove that our approach is a general case of equation (2).

Firstly, observe that allowing an alternative task, say $\overline{\tau}_i$, to run with higher priority does not affect the value of Rf_i. Therefore, there is no reason to raise its priority when $Rf_i \geq Rc_i$ since this would not decrease the value of R_i (see equation (3)). In other words, in the best case the value of R_i can be reduced to Rf_i. On the other hand, the priority assigned to $\overline{\tau}_i$ may affect the value of Rf_j ($\tau_j \in$ intOn(i) $-$ intLess(i)). Because of this one has to observe the schedulability of τ_j (by equation (8)) when raising the priority of $\overline{\tau}_i$. Another interesting but less obvious effect on the value of Rf_j is that when Rc_i decreases as a consequence of raising the priority of $\overline{\tau}_i$, the number of faults considered by equation (7) may drop and consequently the value of Rf_j may decrease. This effect will be illustrated further by a simple example in section 4.4.

Once the condition described above ($Rf_j < Rc_i$) is valid, we have to turn our attention to equation (5) in order to check whether the value of Rc_i^1 can be reduced. It is interesting to note that Rc_i^1 can be reduced when it is possible to decrease (i) the amount of preemption of higher priority tasks or (ii) the interference due to recovery of tasks in intLess(i). Strictly speaking, situations (i) and (ii) are only possible when the following conditions are true, respectively:

(i) There exists at least one task in intLess(i), say τ_k, such that $\lceil Rc_i/T_k \rceil > \lceil Rc_i^0/T_k \rceil$. This means that increasing the priority of $\overline{\tau}_i$ to $\text{pr}(\tau_k)$ level (i.e. shrinking the size of intLess(i)), the preemption caused by the releases of τ_k is eliminated.

(ii) If $\overline{C}_k = \max_{\tau_j \in \text{intLess(i)}}(\overline{C}_j) > \overline{C}_i$, one can eliminate the effects of possible faults on τ_k, which has to be taken into account when computing Rc_i, by making $\text{pr}(\overline{\tau}_i) \geq \text{pr}(\overline{\tau}_k)$.

These conditions, which are direct consequences of equation (5), establish a criterion for searching for the minimum priority to be assigned to alternative tasks in order to obtain reductions in task recovery times. The maximum allowed priority, on the other hand, depends on the schedulability of higher priority tasks as observed before.

Finally, we point out that equation (2) is a particular case of our approach. This is shown by the lemma below.

Lemma 4.1. *If primary and alternative tasks always run in the same priority level, the worst-case response time given by equation* (2) *equals the maximum of* Rc_i *and* Rf_i.

Proof. When alternative task priorities do not change, intOn(i) = hpe(i) and intLess(i) = hp(i). As $Rf_i > Rc_k$ ($\tau_k \in$ hp(i)) since higher priority tasks finishes their execution before lower priority tasks, extra interference on τ_i due to faults on τ_k equals $\lceil Rf_i/T_F \rceil \max_{\tau_k \in \text{hp(i)}}(\overline{C}_k)$. Therefore,

$$Rf_i = C_i + \sum_{\tau_j \in \text{hp(i)}} \left\lceil \frac{Rf_i}{T_j} \right\rceil C_j + \left\lceil \frac{Rf_i}{T_F} \right\rceil \max_{\tau_k \in \text{hp(i)}} (\overline{C}_k) \qquad (9)$$

Also, substituting equation (6) on (5), and after some simple algebra, we can rewrite Rc_i as follows.

$$Rc_i = C_i + \sum_{\tau_j \in \text{hp(i)}} \left\lceil \frac{Rc_i}{T_j} \right\rceil C_j + \overline{C}_i + \left(\left\lceil \frac{Rc_i}{T_F} \right\rceil - 1 \right) \max_{\tau_k \in \text{hpe(i)}} (\overline{C}_k) \qquad (10)$$

Note from equations (9) and (10) that $Rc_i \leq Rf_i$ if $\overline{C}_i \leq \overline{C}_k$. As a result, we can join the equations by considering the same domain for the max function to calculate $R_i = \max(Rc_i, Rf_i)$. This yields equation (2). □

4.4. An Illustrative Example

Consider the example shown in table 2, which illustrates the values of Rc_i and Rf_i for the same task set presented in table 1 when $\overline{\tau}_3$ is allowed to run at a higher priority level. The values of R_i, the maximum of Rc_i and Rf_i, are presented in bold. For the value of T_F equal to 10 time units, as we have seen from table 1, this task set is considered unschedulable ($R_3 > D_3$) by equation (2). If the recovery of task τ_3 (i.e. $\overline{\tau}_3$) is allowed to run even only one priority level above, the task set is considered schedulable. This corresponds to a reduction of Rc_3 by almost 22%. If $\overline{\tau}_3$ runs two priority levels above, this reduction increases to more than 45%.

Table 2. Making a task set schedulable ($\mathbf{T_F} = 10$).

					raising $\overline{\tau}_3$'s priority	
					+1 level	+2 levels
Task	T_i	C_i	$\overline{C}_i$	D	$Rc_i \vert Rf_i$	$Rc_i \vert Rf_i$
τ_1	13	2	2	13	**4**|2	4|**12**
τ_2	25	3	3	25	8|**22**	8|**17**
τ_3	30	5	5	30	**29**|18	**20**|18

It is interesting to note the behaviour of the response time of τ_2. When $\overline{\tau}_3$ runs one priority level above, $R_2 = 22$. This value is dropped to 17 when the priority of $\overline{\tau}_3$ is risen by 2 levels. This effect, caused by the reduction of the maximum number of faults (N_3) was pointed out in section 4.3.

5. Simulation Results

In this section we illustrate, by simulation, the gain in terms of recovery time reduction that can be obtained when alternative tasks are allowed to execute with higher priority. In order to carry out the simulation, random samples of task sets, each one with 10 tasks, were generated. The worst-case computation times of each task set were generated according to an exponential distribution with mean $U/10$, where U is the generated processor utilization bound. 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 be less than or equal to periods. We use the deadline monotonic algorithm to assign the priorities of primary tasks.

We measured the gain of the worst-case recovery time (equation (6)) of task τ_{10} (i.e the lowest priority task) considering that its alternative task was executing at l levels above τ_{10}'s priority ($l = 1, \ldots, 9$). The choice of the lowest priority task gives us the biggest priority range for the simulation. Certainly some upper priority tasks may miss their deadlines due to the execution interference of $\overline{\tau}_{10}$. Because of this we selected those task sets which are schedulable for $\overline{\tau}_{10}$'s priority varying between the lowest and the highest priority. For this simulation $9,600$ task sets were selected.[2] The task sets were grouped by processor utilization ranges $([0.1-0.2), [0.2-0.3), \ldots, [0.8-0.9])$ so that there were $1,200$ tasks sets per range. Then the different recovery time values of $\overline{\tau}_{10}$ executing at priority $10 - l$ were compared to the recovery time of $\overline{\tau}_{10}$ executing with the lowest priority.

In order to find an appropriate value of T_F, we sought for values of T_f considering $\mathrm{pr}(\overline{\tau}_{10}) = p$, for each task set, where $p = 1 \ldots 10$. Then we set T_F to the maximum found T_f. This procedure is important since it keeps the same value of T_F per task set throughout the simulation, allowing a more accurate comparison.

The procedure to find T_f was implemented as a binary search on equation (3) as [13] suggests. Initially, the interval of search is $[max_{\forall i}(C_i), \max_{\forall i}(D_i)]$. This initial interval is appropriate because: (i) T_F cannot assume lower values since faults at this rate or higher will continually disrupt the recovery of the tasks; (ii) any larger value for T_F will imply only one fault within the maximum response time of any task, i.e. if the task set is not schedulable with $T_F = max_{\forall i}(D_i)$, this task set will not be schedulable with faults occurring at any rate.

Figure 6 plots the gains obtained (on average) for each priority level. As expected, the more we increase the priority of $\overline{\tau}_{10}$ the shorter the recovery time is.

[2]Note that a complete evaluation of our approach has to address the problem of choosing alternative task priorities, which is out of the scope of this work.

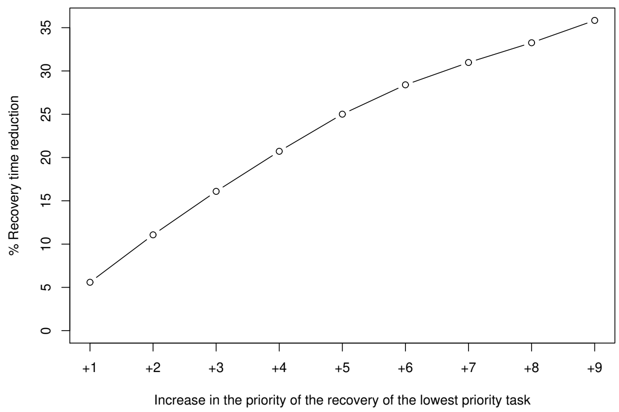

Figure 6. Gain in the recovery time per priority of recovery

6. Related Work

Several scheduling mechanisms for fault-tolerant purposes can be found in the literature. Most of them either assume a restrictive task/fault models and so lack flexibility or impose restrictions on the way in which fault tolerance is carried out. Moreover, those approaches, which are briefly described below, assume that alternative tasks run with the same priorities as their respective primaries. Here we refer only to approaches which deal with temporary faults.

In [10] only periodic tasks are considered; the task periods have to be multiple of each other; and the execution times of alternative tasks have to be shorter than the execution times of their respective primaries.

The approach presented in [6] considers only re-execution of faulty tasks to tolerate temporary faults. As it is based on the Rate Monotonic priority assignment policy [11], its disadvantages are inherited by the proposed mechanism. Another approach that uses similar assumptions has been proposed in [2], which carries fault tolerance by periodically checkpointing tasks in different nodes and so both temporary and permanent faults can be tolerated.

In [7] is described a recovery technique that tolerates temporary 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.

An interesting approach to tolerating temporary faults which is independent of the schedulability analysis being used has been described in [5]. However, it is considered that recovery is carried out only by re-execution of tasks.

Recently, an EDF based scheduling approach, which takes the effects of temporary faults into account, has been proposed [9]. Its basic idea is to simulate the EDF scheduler and to use slack times for executing task recoveries given that a *fault pattern* or the maximum number of faults per task is known *a priory*.

A more flexible scheduling was proposed in [3]. Fault tolerance can be carried out by any error-recovery mechanism since the cost of recovery can be accounted by the schedulability analysis. This approach is represented here by equation (2), which, as we have shown, is a particular case of the analysis we have presented.

7. Conclusion

In this work we have addressed the problem of providing a schedulability analysis for fault-tolerant hard real-time systems regarding temporary faults. We have proposed a suitable solution based on response time analysis that neither impairs the way in which fault tolerance is carried out nor restricts the assumed task/fault models. One important characteristic of our solution is that it allows task recovery to be carried out at higher priority levels. This, as we have seen, has been shown to be a powerful mechanism to increase the flexibility of the analysis, where slack times of higher priority tasks can be better explored. Showing some properties of the analysis, we characterized when it is worth applying our approach.

We have illustrated the advantages of using the described schedulability analyses. By analysing the data collected from simulation we have seen that significant reductions in terms of task recovery time may be obtained by applying the analysis.

Future works will consider heuristics for deciding which higher priority levels to use for general task sets.

Acknowledgement

The authors wish to thank Guillem Bernat and Ian Broster for their comments about this paper.

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 Eng. Journal*, 8(5):284–292, 1993.

[2] A. A. Bertossi and L. V. Mancini. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Systems*, 7(3):229–245, 1994.

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

[4] A. Burns, S. Punnekkat, L. Stringini, and D. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proc. of the 7th International Working Conference on Dependable Computing for Critical Application*, pages 339–356, California, USA, Jan 1999.

[5] S. Ghosh, R. Melhem, and D. Mossé. Enhancing real-time schedules to tolerate transient faults. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, Pisa - Italy, 1995.

[6] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerant rate-monotonic scheduling. In *IFIP International Conference on Dependable Computing for Critical Applications*, Garmisch - Germany, 1997.

[7] N. Kandasamy, J. P. Hayes, and B. T. Murray. Tolerating transient faults in statically scheduled safety-critical embedded sysmes. In *Proc. of the 18th IEEE Symposium on Reliable Distributed Systems*, Lausanne - Switzerland, 1999.

[8] J. C. Laprie. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Spring-Verlag/Wien, 1992.

[9] F. Liberato, R. Melhem, and D. Mossé. Tolerance to mutiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transactions on Computers*, 49(9), September 2000.

[10] L. Liestman and R. H. Campbell. A fault-tolerant scheduling problem. *IEEE Transaction on Software Engineering*, SE-12(11):1089–1095, 1986.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogram in a hard real-time environment. *Journal of ACM*, 20(1):40–61, 1973.

[12] S. McConnel, D. Siewiorek, and M. Tsao. The measurement and analysis of transient erros in digital computer systems. In *Proc. 9th International Conference on Fault-Tolerant Computing (FTCS-9)*, Madison WI, June 1979.

[13] S. Punnekkat. Schedulability analysis for fault tolerant real-time systems. DPhil thesis, Department of Computer Science, University of York, 1997.