An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault-Tolerant Hard Real-Time Systems

George M. de A. Lima and Alan Burns, *Senior Member*, *IEEE*

Abstract—The main contribution of this paper is twofold. First, we present an appropriate schedulability analysis, based on response time analysis, for supporting fault-tolerant hard real-time systems. We consider systems that make use of error-recovery techniques to carry out fault tolerance. Second, we propose a new priority assignment algorithm which can be used, together with the schedulability analysis, to improve system fault resilience. These achievements come from the observation that traditional priority assignment policies may no longer be appropriate when faults are being considered. The proposed schedulability analysis takes into account the fact that the recoveries of tasks may be executed at higher priority levels. This characteristic is very important since, after an error, a task certainly has a shorter period of time to meet its deadline. The proposed priority assignment algorithm, which uses some properties of the analysis, is very efficient. We show that the method used to find out an appropriate priority assignment reduces the search space from $O(n!)$ to $O(n^2)$, where n is the number of task recovery procedures. Also, we show that the priority assignment algorithm is optimal in the sense that the fault resilience of task sets is maximized as for the proposed analysis. The effectiveness of the proposed approach is evaluated by simulation.

Index Terms—Hard real-time systems, fault tolerance, schedulability analysis, priority assignment algorithm.

1 Introduction

Appropriate schedulability analysis schemes are fundamental to the design of predictable hard real-time systems. Response time analysis [1], [9] is one approach that has successfully been used to achieve this goal. In line with this approach, task worst-case response times are efficiently derived due to the fact that tasks have known fixed priorities (given by some priority assignment algorithm). In fact, this scheme provides a good level of flexibility without impairing predictability and has represented a significant step toward the design of flexible and predictable hard-real time systems [20], [21], [4], [16], [17].

Usually, response time analysis has been used for the design of systems on the assumption that there is no error during system execution. The fault-free assumption is, in fact, not realistic. Quoting Laprie [11]:

> Non-faulty systems hardly exist, there are only systems which may have not yet failed.

Response time analysis has recently been extended to cope with the possibility of errors in the system's computation [3]. Modeling the recovery of tasks as *alternative* tasks that should be executed to recover the system from an erroneous state, the authors have shown that fault tolerance based on error-recovery techniques can easily be modeled by response time analysis. However, this approach does not consider appropriate priority assignment schemes to calculate priorities of alternative tasks: They use the same fixed-priority assignment algorithms as those used under the fault-free assumption. Indeed, having a more flexible priority assignment scheme that allows task recovery to be carried out at higher priority levels is very useful since, after errors, tasks certainly have a shorter period of time to meet their deadlines.

The inappropriateness of standard priority assignment approaches for error-recovery can be illustrated as follows (see Fig. 1). The figure represents a set of two hard real-time tasks, $\{\tau_i, \tau_j\}$, where the priority of τ_j is higher than the priority of τ_i. Associated with τ_i is its alternative task, $\overline{\tau}_i$, which should be executed in case of errors in τ_i and must finish by τ_i's deadline, D_i. Consider that an error interrupts the execution of τ_i, which means that the error is raised in τ_i. As can be seen from the scenario illustrated in Fig. 1a, it is not possible to recover τ_i before its deadline because of preemption due to the execution of τ_j. Nevertheless, if there is enough slack time available at some higher priority level where $\overline{\tau}_i$ can be executed, it may finish before D_i, as Fig. 1b illustrates. In this example, the preemption of the second activation of τ_j is avoided by assigning a higher priority to $\overline{\tau}_i$.

The difficulty in deriving an appropriate schedulability analysis to cope with the example illustrated in Fig. 1b comes from the fact that the worst-case scenario is not characterized by the instant when all tasks are released at once as it is in the standard response time analysis. In Fig. 1, for instance, we can note that the response time of τ_j in its second activation (Fig. 1b) is higher than its response time in Fig. 1a due to extra preemption caused by the execution of $\overline{\tau}_i$. To carry out an appropriate response time analysis which takes these characteristics into account, we divide the calculation of task worst-case response times into three main phases. First, we derive, for each task in the task set,

• *The authors are with the Real-time Systems Research Group, Department of Computer Scienc, University of York, York YO10 5DD, UK. E-mail: {gmlima, burns}@cs.ork.ac.uk.*

Manuscript received 10 Oct. 2001; revised 4 Oct. 2002; accepted 3 Dec. 2002. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115168.

0018-9340/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

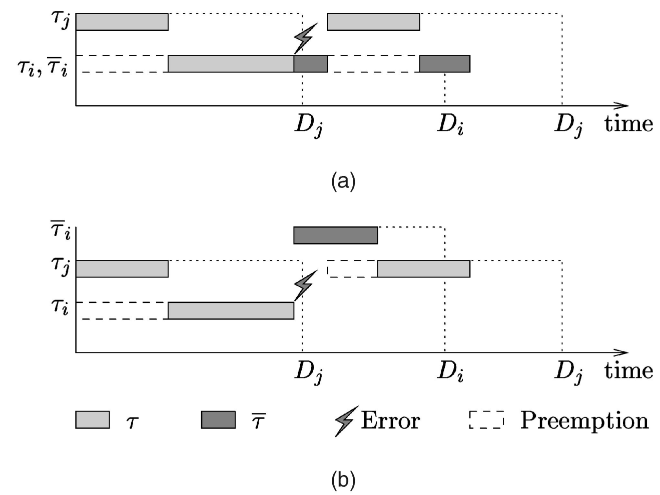

Fig. 1. Two illustrative scenarios: (a) both τ_i and its recovery run at the same priority level; (b) $\overline{\tau}_i$ runs with higher priority.

its worst-case response time, assuming that errors interrupt the execution of other tasks but τ_i (i.e., due to *external* errors). Then, we derive its worst-case response time, assuming that τ_i may be faulty. In other words, this is the time necessary, in the worst-case, to meet task deadlines despite *internal* errors. Finally, the values of worst-case response times due to internal and external errors are used to derive the worst-case response time of tasks. This approach has already been proposed by us [14]. Here, an improved version of this approach is described[1] and a more appropriate notation is used. Also, in this work, we propose an *optimal* priority assignment algorithm to determine the priorities of alternative tasks, a problem that has not been addressed before. By optimal we mean that the algorithm finds the best priority assignment so that the number of errors tolerated (fault resilience) by the task set is maximized and the task set is considered schedulable by the analysis. For example, from Fig. 1, we can see that the priority assignment given in Fig. 1b is better than the one given in Fig. 1a since, in Fig. 1a, the task set is not schedulable. This is due to the fact that, in Fig. 1b, the spare capacity at higher priority levels is being used to recover τ_i from errors, which makes the task set more resilient to faults.

It is important to realize that the schedulability analysis and the search for the optimal priority assignment are interdependent problems and, so, the concept of optimal is relative to the considered analysis, as we now explain. The available spare capacity, necessary to assign priorities to alternative tasks, is only known after carrying out the schedulability analysis, which gives the worst-case response times of tasks. On the other hand, an optimal priority assignment can only be given after discovering the available spare capacity. This dependency cycle suggests an iterative procedure, where priorities and task response times are calculated altogether throughout the iterations. This is the basic idea of our approach.

Carrying out this iterative procedure by brute-force, which tests all possible priority combinations, is not practical since the number of possible priority assignments is too high. For a task set with n alternative tasks, the search space is $O(n!)$, for instance. We solve the problem of assigning priorities to alternative tasks in a very efficient way. The algorithm to do so is iterative, as suggested. However, by establishing a partial order on the alternative task's priorities and using some properties of the derived schedulability analysis, only a few priority configurations need to be examined. Indeed, the proposed assignment algorithm reduces the search space from $O(n!)$ to $O(n^2)$. We have proven that the proposed algorithm finds the optimal priority configuration in the sense that it maximizes the system fault resilience, as seen by the proposed analysis. To the best of our knowledge, the problem of using a nonstandard priority assignment to maximize fault resilience has not been addressed before.

The remainder of this paper is organized as follows: The next section presents the assumed computation model. Then, some initial concepts on the use of response time analysis for fault tolerance purposes are presented in Section 3. Also, in this section, we give the main motivation by presenting an illustrative example. Section 4 presents our approach to carrying out the schedulability analysis. In Section 5, the method for finding out the optimal priority configuration is given and its optimality is proven. The algorithm to implement such a method and a proof of its correctness are given in Section 6. Results from simulation are shown in Section 7. A brief survey on related works is presented in Section 8. Section 9 presents our final comments.

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 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 interarrival 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 ($\mathrm{FP}(\Gamma)$), 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 p_i and $\overline{p}_i$, 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.

[1] Actually, this previous work turned out to be optimistic for some scenarios.

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 reexecution. We also assume that there is no cost associated with any scheduling of primary or alternative tasks. 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], modeled here as alternative tasks. In addition, one may consider some kind of *transient faults*, where either the reexecution 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 modeled by alternative tasks, which can be released to carry out a compensation action. However, it is important to emphasize 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 faults 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 assume in the analysis that there is a minimum time between two consecutive error occurrence, T_E. By finding out the minimum value of T_E such that the system is still schedulable, we are expressing to what extent the system is resilient to the occurrence of errors. This may be important information about the system's fault resilience. For example, if designers are aware of such a value, they may infer how likely the system will be subject to missed deadlines due to faults in a given environment.

It is important to emphasize that the motivation for the approach taken here is to improve fault resilience. To do this, we require a measure of this resilience. The magnitude of T_E is the measure employed in this paper. We assume that the scheduling of a system to reduce T_E will improve the resilience of the system, regardless of the actual fault model employed by the application engineer. Support for the use of this metric comes from Burns et al. [5], who proved that if error arrivals are modeled by a Poison distribution (a usual and often realistic assumption), then the probability of failure during the lifetime of the system is proportional to T_E (i.e., the smaller the value of T_E, the more fault resilient the system).

3 Background and Motivation

Schedulability analysis based on the well-known response time analysis [1] which takes into account the effects of possible faults has already been proposed [3]. In this section, we summarize this result and illustrate its limitations.

The input parameters of this analysis are: the task attributes (T_i, D_i, C_i, and $\overline{C}$); the primary task priorities (p_i), which are given by some fixed-priority assignment algorithm (e.g., deadline monotonic); and the assumed value of T_E. The priorities of alternative tasks are assumed to be the same as their primary tasks ($p_i = \overline{p}_i$).

Consider that no task suffers any error. Under this particular scenario, the worst-case response time of task τ_i is the time necessary to execute τ_i and all tasks τ_j such that $p_j > p_i$. When faults are considered, on the other hand, we have to include in the calculation of the worst-case response time of τ_i the time necessary to recover the faulty task, as we explain below.

Initially, let us consider that only one error may take place during the execution of τ_i. Any task that may be executing concurrently with τ_i (including τ_i itself) may be interrupted by this error. In the worst-case scenario, the error interrupts tasks just before the end of their execution and the faulty task is the one with the longest alternative task among τ_i and all tasks that may preempt the execution of τ_i (i.e., tasks with priority greater than or equal to p_i). Let us say that τ_k is such a task. This means that we have to add the time to recover τ_k (i.e., recovery cost $\overline{C}_k$) to the response time of τ_i. These observations lead to (1), where R_i is the worst-case response time of τ_i, $\mathrm{hp}(i) = \{\tau_j \in \Gamma | \, p_j > p_i\}$ and $\mathrm{hpe}(i) = \{\tau_j \in \Gamma | \, p_j \geq p_i\}$. Since R_i appears on both sides of the equation, its solution is obtained, as usual, iteratively by forming a recurrence relation with $R_i^0 = C_i$. This iterative procedure finishes either when $R_i^{m+1} = R_i^m$ (the worst-case response time of τ_i is found) or when $R_i^{m+1} > D_i$ (τ_i is considered unschedulable).[2]

$$R_i = C_i + \sum_{\tau_j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{\tau_k \in \mathrm{hpe}(i)} \overline{C}_k. \qquad (1)$$

Nevertheless, errors may interrupt the execution of τ_k more than once. As the worst-case period of error occurrence is T_E, the maximum number of errors that may take place during the execution of τ_i is given by $\lceil \frac{R_i(T_E)}{T_E} \rceil$, which leads to (2), originally presented in [18], [3]. As T_E is an input to the analysis, R_i is given as a function of T_E, i.e., $R_i(T_E)$. Also, note that this equation is conservative in the sense that, in practice, some of the errors may not interrupt the task with the largest recovery cost. Indeed, this conservative assumption could only hold if error occurrences were in phase with tasks. Assuming that errors always interrupt the execution of such a task, however, simplifies the worst-case response time computation. In this work, we use the same sort of conservative assumption.

[2] For the sake of simplicity, we are not considering jitter and blocking effects, although they can easily be incorporated into the analysis.

TABLE 1
Illustration of the Limitations of (2)

Task	Task Attributes						
	T_i	C_i	$\overline{C}_i$	D_i	p_i	$R_i(11)$	$R_i(10)$
τ_1	13	2	2	13	3	4	4
τ_2	25	3	3	25	2	8	8
τ_3	30	5	5	30	1	22	**37**

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

Now, consider a simple but illustrative example, presented in Table 1, which gives a task set with three tasks. As can be seen from the solutions of (2) presented in the table, this task set is schedulable for $T_E = 11$ and unschedulable for $T_E = 10$ since τ_3 does not meet its deadline ($R_3(10) > D_3$). Nevertheless, there is some slack time available at priority levels p_1 and p_2 that is not being used for carrying out the execution of $\overline{\tau}_3$.

Our goal in this work is to develop a more generic schedulability analysis, according to which we can deal with alternative tasks running at higher priority levels. By doing so we can make use of the available slack time to carry out task recoveries. This involves two steps. First, new equations that can cope with this characteristic have to be derived (Section 4). Second, an efficient algorithm to calculate the optimal priority assignment for alternative tasks must be available (Sections 5 and 6).

4 Schedulability Analysis

Our goal in this section is to derive schedulability analysis that takes into account alternative tasks running at higher priority levels. We assume that the priorities of alternative tasks are known beforehand. This assumption will be withdrawn in Section 5. Thus, in this section, we are only concerned with finding out whether or not a given task set is considered schedulable for a given value of T_E.

A particular choice for alternative task priorities is named a *priority configuration*, which is defined as follows:

Definition 4.1. *A priority configuration, P_x, is a tuple $\langle h_{x,1}, \dots, h_{x,n} \rangle$, where $0 \le h_{x,i} < i$ and $h_{x,i} = \overline{p}_i - p_i$.*

As can be noted from the definition, $h_{x,i}$ represents the priority increment for task $\overline{\tau}_i$ in relation to the priority of 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. For example, consider $P_x = \langle 0, 0, \dots, 0 \rangle$, 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 $P_x = \langle 0, 0, \dots, 0, 1 \rangle$, all tasks execute at their original priority level apart from $\overline{\tau}_n$, which executes one priority level above its primary task. Thus, the schedulability analysis we will present is a function of P_x and T_E.

This section is structured as follows: Section 4.1 illustrates and characterizes the effects of raising priorities of alternative tasks. Section 4.2 describes the equations to calculate worst-case task response times. In Section 4.3, the effectiveness of the analysis using the example given in Table 1 is illustrated. As we will see from this example, one can achieve significant gains by using the proposed analysis.

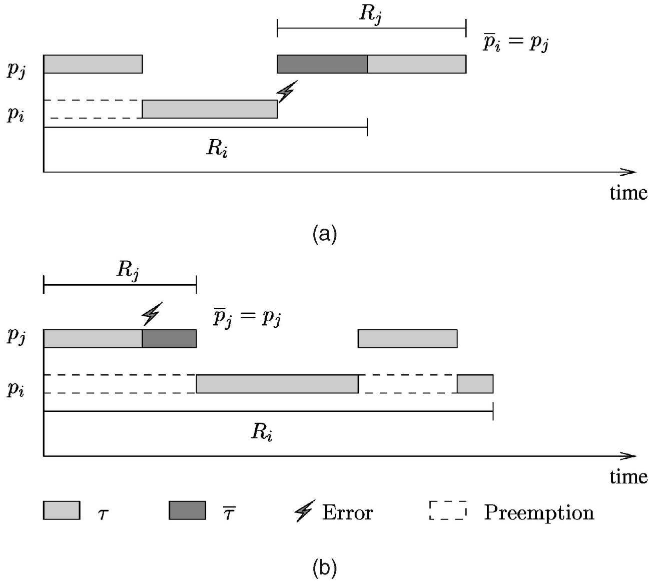

Fig. 2. Worst-case execution scenarios: (a) for task τ_j and (b) for task τ_i.

4.1 Raising Priorities of Alternative Tasks

In order to understand the effects of raising priorities of alternative tasks, consider the set of two tasks in Fig. 2. Task τ_j has higher priority than τ_i. Suppose that an error interrupts τ_i just before the completion of its execution (Fig. 2a). As can be seen, $\overline{\tau}_i$ is then selected to execute with precedence over 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 (since the second execution of τ_j is delayed). 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.

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 alternative task as in (2). For example, consider Fig. 2b, which represents a different execution scenario for tasks τ_i and τ_j. Consider that $\overline{C}_j < \overline{C}_i$ and that an error interrupts the execution of τ_j instead of τ_i. This situation, as can be noted from the figure, leads to a longer response time for task τ_i when compared to Fig. 2a. This is because task τ_i suffers not only the interference of $\overline{\tau}_j$, but also the interference of another activation of task τ_j.

Summing up, the characterization of the worst-case scenario is more complex than the traditional approach (by (2)). Indeed, we have to observe the worst-case interferences due to both preemption and possible errors, which may involve the recovery of lower priority tasks. For example, from the above figure, 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 error interrupts the execution of τ_j just before its execution. By contrast, the worst case for task τ_j is when it is released just after task τ_i is interrupted by an error. Let us identify tasks

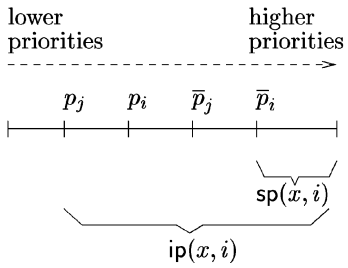

Fig. 3. Γ subsets with respect to task τ_i.

according to the extra interference they cause or suffer due to errors by defining three subsets of Γ:

- $\mathrm{ip}(x, i)$. These are the tasks that may interfere with the response time of τ_i as regards priority configuration P_x if an error occurs. More formally, $\mathrm{ip}(x, i) = \{\tau_j \in \Gamma |\, h_{x,j} + p_j \geq p_i\}$.
- $\mathrm{sp}(x, i)$. 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. Their priorities are higher than $\overline{p}_i$. More formally, $\mathrm{sp}(x, i) = \{\tau_j \in \Gamma |\, p_j > h_{x,i} + p_i\}$.
- $\mathrm{ipe}(x, i)$. This subset is defined as

$$\mathrm{ipe}(x, i) = \begin{cases} \mathrm{ip}(x, i) & \text{if } h_{x,i} = 0 \\ \mathrm{ip}(x, i) - \{\tau_i\} & \text{if } h_{x,i} > 0. \end{cases}$$

  This subset is particularly useful, as we will see in Section 4.2.2, for modeling cases where errors may interrupt task τ_i since the maximum interference its recovery suffers depends on whether or not $p_i = \overline{p}_i$.

Fig. 3 illustrates the meaning of subsets $\mathrm{ip}(x, i)$ and $\mathrm{sp}(x, i)$. Note that τ_i does not suffer any interference from tasks in $\Gamma - \mathrm{ip}(x, i)$, but suffers the interference from $\overline{\tau}_j$ since $\tau_j \in \mathrm{ip}(x, i)$. Thus, when calculating the response time of τ_i for a given priority configuration P_x, we need to consider only errors in tasks belonging to $\mathrm{ip}(x, i)$.

4.2 Task Response Time Calculation

In this section, we show how the worst-case response times of tasks are computed. Let Γ be a task set which is subject to faults so that the minimum time between error occurrences is bounded by $T_E > 0$ and assume that the priority configuration for the alternative tasks is given by P_x. The derivation of the worst-case response time of any task $\tau_i \in \Gamma$, denoted $R_i(x, T_E)$, is split into two branches: considering that errors interrupt the execution of any task but τ_i and considering that τ_i may be interrupted by some error. The justification of this approach is that the worst-case response time of any task τ_i may depend on whether or not the execution of τ_i is itself interrupted by some error (see Fig. 2).

We call an error *internal* if it interrupts τ_i (or $\overline{\tau}_i$) or *external* if it interrupts other tasks. We define $R_i^{ext}(x, T_E)$ to be the worst-case response time of τ_i in cases where only external errors are considered. In cases where some internal error takes place, the computation of the worst-case response time of τ_i is given by $R_i^{int}(x, T_E)$. Sections 4.2.1 and 4.2.2 describe the equations that give the values of $R_i^{ext}(x, T_E)$ and $R_i^{int}(x, T_E)$, respectively. Once the values of $R_i^{ext}(x, T_E)$ and $R_i^{int}(x, T_E)$ are known, $R_i(x, T_E)$ can be easily derived (Section 4.2.3).

4.2.1 Considering Only External Errors

The computation of the worst-case response time of task τ_i due to external errors, $R_i^{ext}(x, T_E)$, is straightforward. This is because we do not need to consider the recovery of τ_i. In this situation, the worst-case scenario as for task τ_i can be described as follows: 1) Errors take place at a rate of $1/T_E$; 2) just before the release of τ_i, some alternative task with maximum recovery time among all tasks in $\mathrm{ip}(x, i) - \{\tau_i\}$ is released; and 3) all tasks in $\mathrm{hp}(i)$ are released so that they cause the maximum interference in the execution of τ_i. Therefore, we have to take into account the time to execute τ_i plus all tasks in $\mathrm{hp}(i)$ and the time to recover the faulty task times the maximum number of errors that may occur over $R_i^{ext}(x, T_E)$. This scenario yields (3), which is similar to (2). Here, we consider that $\max_{\tau_k \in \emptyset}(C_k) = 0$.

$$\begin{aligned} R_i^{ext}(x, T_E) = C_i + \sum_{\tau_j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{ext}(x, T_E)}{T_j} \right\rceil C_j \\ + \left\lceil \frac{R_i^{ext}(x, T_E)}{T_E} \right\rceil \max_{\tau_k \in \mathrm{ip}(x,i) - \{\tau_i\}} (\overline{C}_k). \end{aligned} \tag{3}$$

It is clear that, in general, if errors arrive at each T_E time units, some of them may not hit the task with the largest recovery cost. However, like (2), here we make this conservative assumption for the sake of simplicity. Note that analyzing all the possibilities of error occurrences to have a less pessimistic approach may lead to a computationally impractical and/or complex solution.

Not considering τ_i in the computation of $R_i^{ext}(x, T_E)$ may appear counterintuitive at first. Indeed, after the recovery of some faulty task $\tau_k \in \mathrm{ip}(x, i) - \{\tau_i\}$, internal errors may take place. However, these internal errors are only relevant for the derivation of the worst-case response time of τ_i when the recovery cost of τ_i is maximum. This is the result of Lemma 4.1. If $\overline{C}_i$ is maximum, we need to consider these internal errors, a problem that we address in the next section.

Lemma 4.1. *Consider a fixed-priority set of primary tasks Γ and their respective alternative tasks. Suppose that Γ is subject to faults so that the minimum time between error occurrences is bounded by $T_E > 0$ and let P_x be a priority configuration for the alternative tasks. If $\overline{C}_i < \max_{\tau_k \in \mathrm{ip}(x,i)}(\overline{C}_k)$, $R_i^{ext}(x, T_E)$ represents the worst-case response time of τ_i regardless of whether or not the execution of τ_i is interrupted by some error.*

Proof. If just external errors take place regarding τ_i, the lemma holds by the explanation given earlier in this section. Hence, we have to prove that the lemma holds assuming that some internal error takes place. Thus, let us assume a hypothetical (but generic) scenario in which there is at least one internal error as for τ_i. Without loss of generality, define t as the time at which τ_i is released, $t' > t$ the time at which an internal error interrupts its execution, and $t'' > t'$ the worst-case finishing time of $\overline{\tau}_i$ despite other possible errors. Note that the existence of t and t' is guaranteed by assumption. In this circumstance, there have been at most $m + m' + 1 = \lceil \frac{(t''-t)}{T_E} \rceil$ error occurrences, $m \geq 0$ of which take place during $[t, t')$,

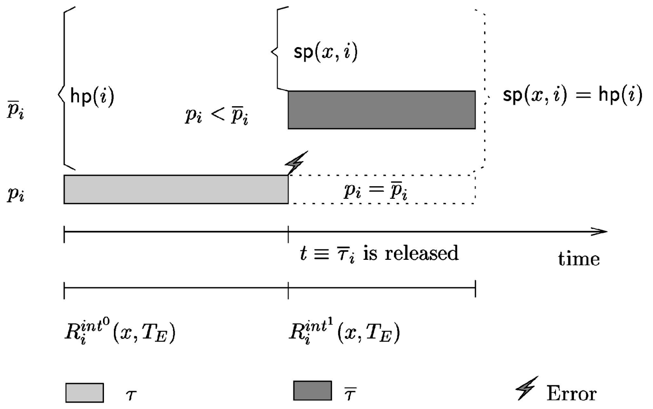

Fig. 4. Illustration of the derivation of $R_i^{int}(x, T_E)$.

one error that interrupts the execution of τ_i at time t', and $m' = \lceil \frac{t''-t}{T_E} \rceil - m - 1 \geq 0$ error occurrences that take place during the interval (t', t'') (by definition of t' and t''). From time t until time t', τ_i suffers the interference due to the execution of tasks in $\mathrm{hp}(i)$, whereas, from t' to t'', it suffers the interference due to the execution of tasks in $\mathrm{sp}(x,i)$. Taking errors into account, in the worst-case: m error occurrences interrupt the execution of a task τ_k that has the longest recovery cost among all tasks in $\mathrm{ip}(x,i) - \{\tau_i\}$ and m' error occurrences interrupt the execution a task τ_l that has the longest recovery cost among all tasks in $\mathrm{sp}(x,i) \cup \{\tau_i\}$.

Now, let us compare $R_i^{ext}(x, T_E)$ with $t'' - t$, where we have to prove that $R_i^{ext}(x, T_E) \geq t'' - t$. It is clear that $R_i^{ext}(x, T_E)$ cannot be less than $t' - t$ since, by assumption, τ_i was executing at t' and, during this interval of time, (3) takes into account the worst-case interference. In other words, (3) takes into account at least: $m + 1$ error occurrences during $[t, t']$ and the same amount of interference caused by alternative and primary tasks that may preempt τ_i during $[t, t']$. From t' onward, however, (3) takes into account external errors in task τ_l. Since $\mathrm{sp}(x,i) \subseteq \mathrm{ip}(x,i)$, $\overline{C}_i < \overline{C}_k$, and $\overline{C}_l \leq \overline{C}_k$, (3) cannot converge to a number smaller than $t'' - t$, as required. □

Although we do not need to carry out the computation of $R_i^{int}(x, T_E)$ when $\overline{C}_i < \max_{\tau_k \in \mathrm{ip}(x,i)}(\overline{C}_k)$, we will do so for the sake of illustration. This means that the equations we derive in the next section will take into account scenarios ruled out by Lemma 4.1.

4.2.2 Considering Internal Errors

In this section, we assume that there is at least one internal error during the execution of τ_i. The general strategy for deriving $R_i^{int}(x, T_E)$ is illustrated in Fig. 4. As can be seen from the figure, an internal error takes place at time t which releases $\overline{\tau}_i$. The interference that τ_i and $\overline{\tau}_i$ suffer due to the execution of other tasks may be different if $p_i < \overline{p}_i$, as is illustrated in the figure. The objective of the analysis is to derive the worst-case response times of τ_i before and after the error. We call these times $R_i^{int^0}(x, T_E)$ and $R_i^{int^1}(x, T_E)$, respectively.

Deriving the values of $R_i^{int^0}(x, T_E)$ and $R_i^{int^1}(x, T_E)$ is not simple since: 1) This may involve two levels of priorities, before and after the first internal error; 2) the procedure to carry out response time analysis is iterative; and 3) the information about *when* the first internal error takes place is not available beforehand. In other words, in general, $R_i^{int^0}(x, T_E)$ and $R_i^{int^1}(x, T_E)$ cannot both be derived *at once* using response time analysis.

In order to circumvent difficulties 1) and 2), we compute the values of $R_i^{int^0}(x, T_E)$ and $R_i^{int^1}(x, T_E)$ separately. This strategy makes it easier to use response time analysis for taking into account the different interference in both priority levels, before and after the error. The final result of $R_i^{int}(x, T_E)$ can then be given by the sum of $R_i^{int^0}(x, T_E)$ and $R_i^{int^1}(x, T_E)$, as we will see. Due to difficulty 3), we carry out the following approach: First, we suppose that the execution of τ_i is interrupted by an error at some time t (as illustrated in Fig. 4). Then, we can easily derive $R_i^{int^1}(x, T_E)$. Note that this derivation does not need any information about what happened before t. Then, using the computed value of $R_i^{int^1}(x, T_E)$, we derive $R_i^{int^0}(x, T_E)$. We detail this approach below.

Computing $R_i^{int^1}(x, T_E)$. Here, we assume that an internal error took place. What we have to do is to show how long the recovery of τ_i will last subject to both other possible errors and the interference due to tasks in $\mathrm{sp}(x,i)$. In the worst-case, there may be $\lceil \frac{R_i^{int^1}(x,T_E)}{T_E} \rceil$ errors over the period $R_i^{int^1}(x, T_E)$. The first error accounts for $\overline{C}_i$, while the others may cause the release of the recovery of any task in $\mathrm{sp}(x,i) \cup \{\tau_i\}$. The worst case is when all other errors interrupt a task in $\mathrm{sp}(x,i) \cup \{\tau_i\}$ that has the longest recovery time.[3] Therefore, $R_i^{int^1}(x, T_E)$ is given by (4).

$$\begin{aligned} R_i^{int^1}(x, T_E) = \overline{C}_i &+ \sum_{\tau_j \in \mathrm{sp}(i)} \left\lceil \frac{R_i^{int^1}(x, T_E)}{T_j} \right\rceil C_j \\ &+ \left(\left\lceil \frac{R_i^{int^1}(x, T_E)}{T_E} \right\rceil - 1 \right) \max_{\tau_k \in \mathrm{sp}(x,i) \cup \{\tau_i\}} (\overline{C}_k). \end{aligned} \tag{4}$$

Computing $R_i^{int^0}(x, T_E)$. The computation of $R_i^{int^0}(x, T_E)$ is slightly more complex. Let us analyze it considering two cases depending on the values of p_i and $\overline{p}_i$.

When $p_i < \overline{p}_i$. This means that $\overline{\tau}_i$ executes at a higher priority level. Note that, in this case, knowing $R_i^{int^0}(x, T_E)$ 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. 4. During $R_i^{int^0}(x, T_E)$, τ_i may suffer the preemption of tasks in $\mathrm{hp}(i)$ and possibly the recoveries of tasks in $\mathrm{ip}(x,i) - \{\tau_i\}$ due to other errors. It is important to note that we have to remove τ_i from the set of tasks that may suffer errors in this phase because, by assumption, the first 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.

[3] As said before, here we are assuming a generic situation. However, in practice, one can consider that all errors from t onward are internal due to Lemma 4.1.

When $p_i = \overline{p}_i$. Unlike the former case, the maximum interference during $R_i^{int^0}(x, T_E)$ 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 $\overline{C}_i = \max_{\tau_k \in \mathrm{ip}(x,i)}(\overline{C}_k)$ (recall (2)). As a result, instead of considering errors in $\mathrm{ip}(x,i) - \{\tau_i\}$, one should consider errors in the whole $\mathrm{ip}(x,i)$.

In summary, as for possible errors during $R_i^{int^0}(x, T_E)$, when $p_i < \overline{p}_i$, one has to consider errors in $\mathrm{ip}(x,i) - \{\tau_i\}$. Otherwise, errors in $\mathrm{ip}(x,i)$ should be taken into account. This is the main difference between the cases analyzed above. In order to join both cases together in a single equation, we say that errors during the interval $R_i^{int^0}(x, T_E)$ may take place in any task in $\mathrm{ipe}(x,i)$ (see Section 4.1).

Now, we are able to derive the equation that gives $R_i^{int^0}(x, T_E)$. It has to take into account: the worst-case execution time of τ_i (C_i), the interference due to tasks in $\mathrm{hp}(i)$, and possible recoveries of tasks in $\mathrm{ipe}(x,i)$. Note that some releases of tasks in $\mathrm{sp}(x,i)$ and some error occurrences may have already been taken into account when computing $R_i^{int^1}(x, T_E)$. This means that we have to take care not to compute the same task in $\mathrm{sp}(x,i)$ and the same error occurrence twice. In other words, we have to subtract, for each task in $\mathrm{sp}(x,i)$ and each error occurrence, the interference already computed in $R_(^{int^1}i)(x, T_E)$.

From the description above, (5) gives the value of $R_i^{int^0}(x, T_E)$. Note that, instead of computing the worst-case interference due to tasks in $\mathrm{hp}(i)$, we split this computation as for two complementary subsets, $\mathrm{hp}(i) - \mathrm{sp}(x,i)$ and $\mathrm{sp}(x,i)$. This is to avoid the computation of tasks in $\mathrm{sp}(x,i)$ more than once, as commented before. We do so by subtracting $\lceil \frac{R_i^{int^1}(x,T_E)}{T_l} \rceil \overline{C}_l$ for each task $\tau_l \in \mathrm{sp}(x,i)$. Similarly, possible double counting of errors is removed by subtracting $\lceil \frac{R_i^{int^1}(x,T_E)}{T_E} \rceil$ from the total number of errors.

$$R_i^{int^0}(x, T_E) = C_i + \sum_{\tau_j \in \mathrm{hp}(i) - \mathrm{sp}(x,i)} \left\lceil \frac{R_i^{int^0}(x, T_E)}{T_j} \right\rceil C_j + \sum_{\tau_l \in \mathrm{sp}(x,i)} \left(\left\lceil \frac{R_i^{int}(x, T_E)}{T_l} \right\rceil - \left\lceil \frac{R_i^{int^1}(x, T_E)}{T_l} \right\rceil \right) C_l + \left(\left\lceil \frac{R_i^{int}(x, T_E)}{T_E} \right\rceil - \left\lceil \frac{R_i^{int^1}(x, T_E)}{T_E} \right\rceil \right) \max_{\tau_k \in \mathrm{ipe}(x,i)} (\overline{C}_k). \quad (5)$$

The value of $R_i^{int}(x, T_E)$ can then be simply derived by taking the sum of $R_i^{int^0}(x, T_E)$ and $R_i^{int^1}(x, T_E)$:

$$R_i^{int}(x, T_E) = R_i^{int^0}(x, T_E) + R_i^{int^1}(x, T_E). \quad (6)$$

The computation of $R_i^{int}(x, T_E)$ is carried out by iteration as usual. Initially, the calculation of $R_i^{int^1}(x, T_E)$ is done and then its value is used in (5). Notice that the procedure for calculating $R_i^{int^0}(x, T_E)$ does not affect the value of $R_i^{int^1}(x, T_E)$.

4.2.3 Worst-Case Response Time

In this section, we show how to derive $R_i(x, T_E)$, the worst-case response time of τ_i, from $R_i^{ext}(x, T_E)$ and $R_i^{int}(x, T_E)$. In order to give an intuition behind this derivation, let us analyze two cases below. Consider τ_k a task in $\mathrm{ip}(x,i)$ such that $\overline{C}_k = \max_{\tau_l \in \mathrm{ip}(x,i)}(\overline{C}_l)$.

If $\tau_k \in \mathrm{ip}(x,i) - \{\tau_i\}$. In this case, $R_i^{int}(x, T_E)$ is maximum when all errors hit task τ_k or some other task τ_l (or their alternative tasks) that has recovery cost equal to $\overline{C}_k$. This, in turn, can only be true if τ_k or τ_l belong to $\mathrm{sp}(x,i)$. Without loss of generality, assume that $\tau_k \in \mathrm{sp}(x,i)$. Hence, the computation of $R_i^{int}(x, T_E)$ takes into account $m - 1 \geq 0$ error occurrences regarding τ_k and one for τ_i, where m is the maximum number of errors that may occur during $R_i^{int}(x, T_E)$. Clearly, this represents the worst-case scenario when some internal error takes place. However, as the computation of $R_i^{ext}(x, T_E)$ takes into account all error occurrences for τ_k, $R_i^{ext}(x, T_E)$ assumes a value at least as big as $R_i^{int}(x, T_E)$. Indeed, if $\overline{C}_i < \overline{C}_k$, by Lemma 4.1, we know that $R_i^{ext}(x, T_E) \geq R_i^{int}(x, T_E)$. Moreover, if $\overline{C}_i = \overline{C}_k$, it is not difficult to see that $R_i^{ext}(x, T_E) \geq R_i^{int}(x, T_E)$ since $\mathrm{sp}(x,i) \subseteq \mathrm{hp}(i)$. Therefore, $R_i(x, T_E) = R_i^{ext}(x, T_E)$.

If $\tau_k = \tau_i$. In this case, the computation of $R_i^{int}(x, T_E)$ takes into account some errors in another task $\tau_l \in \mathrm{ip}(x,i) - \{\tau_i\}$ and some in τ_i. Since $R_i^{int}(x, T_E)$ depends on p_i, $\overline{p}_i$, $\overline{C}_i$, and $\overline{C}_l$, the relation between $R_i^{ext}(x, T_E)$ and $R_i^{int}(x, T_E)$ is unknown before the computation of their values. In other words, in this case, $R_i(x, T_E)$ is given by the maximum of $R_i^{ext}(x, T_E)$ and $R_i^{int}(x, T_E)$.

Therefore, the generic expression that gives the value of $R_i(x, T_E)$ is straightforwardly given by

$$R_i(x, T_E) = \max(R_i^{ext}(x, T_E), R_i^{int}(x, T_E)). \quad (7)$$

To conclude this section, we call the attention of the reader to the fact that the described analysis represents a generalization of the analysis by (2). This is proven by the lemma below.

Lemma 4.2. *Consider a set of fixed-priority scheduled set of primary tasks Γ and their alternative tasks. For any value of $T_E > 0$, the worst-case response time given by (2) equals the maximum of $R_i^{ext}(x, T_E)$ and $R_i^{int}(x, T_E)$ whenever $P_x = \langle 0, 0, \ldots, 0 \rangle$.*

Proof. The proof of this lemma is straightforward and follows the observation that, when $P_x = \langle 0, 0, \ldots, 0 \rangle$, $\mathrm{hp}(i) = \mathrm{sp}(x,i)$ and $\mathrm{ipe}(x,i) = \mathrm{hpe}(i) = \mathrm{ip}(x,i)$. After some simple algebra, (3) and (6), respectively, can be rewritten as follows:

$$R_i^{ext}(x, T_E) = C_i + \sum_{\tau_j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{ext}(x, T_E)}{T_j} \right\rceil C_j + \left\lceil \frac{R_i^{ext}(x, T_E)}{T_E} \right\rceil \max_{\tau_k \in \mathrm{hp}(i)} (\overline{C}_k)$$

and

$$R_i^{int}(x, T_E) = C_i + \sum_{\tau_j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{int}(x, T_E)}{T_j} \right\rceil C_j + \left(\left\lceil \frac{R_i^{int}(x, T_E)}{T_E} \right\rceil - 1 \right) \max_{\tau_k \in \mathrm{hpe}(i)} (\overline{C}_k) + \overline{C}_i.$$

It is clear that if

TABLE 2
The Effects of Raising Priorities of Alternative Tasks for Different Priority Configurations

		$T_E = 10$						$T_E = 8$	
		$\langle 0,0,0\rangle$		$\langle 0,0,1\rangle$		$\langle 0,0,2\rangle$		$\langle 0,0,2\rangle$	
Task	Deadline	R_i^{int}	R_i^{ext}	R_i^{int}	R_i^{ext}	R_i^{int}	R_i^{ext}	R_i^{int}	R_i^{ext}
τ_1	13	**4**	2	**4**	2	4	**7**	4	**7**
τ_2	25	**8**	7	8	**10**	8	**10**	8	**22**
τ_3	30	**37**	18	**20**	18	**18**	18	**23**	21

$$\overline{C}_i = \max_{\tau_k \in \text{hpe}(i)} (\overline{C}_k),$$

$R_i^{int}(x, T_E) \geq R_i^{ext}(x, T_E)$. Otherwise,

$$R_i^{int}(x, T_E) \leq R_i^{ext}(x, T_E).$$

The maximum of these two equations can then be rewritten as a single equation, which yields (2). □

4.3 An Illustrative Example

As we have seen, when T_E is set to 10 time units, the task set presented in Table 1 is unschedulable for priority configuration $\langle 0,0,0\rangle$. Table 2 shows that, for priority configurations $\langle 0,0,1\rangle$ and $\langle 0,0,2\rangle$, the task set is schedulable according to the analysis described earlier. This is because the slack time available at higher priority levels is being used to execute $\overline{\tau}_3$. The two values given in each cell of Table 2 are the solutions of (6) and (3), respectively. The maximum value (i.e., the worst-case response time) is in bold.

The advantages of considering alternative tasks executing at higher priority levels are not only noted from the significant reductions of task response times, but also from the increase in the fault resilience of the task set. In this example, the value of T_E drops from 11 (priority configuration $\langle 0,0,0\rangle$) to 8 (priority configuration $\langle 0,0,2\rangle$), as illustrated in the table. This represents a gain of 27.3 percent, which may be very significant when dealing with critical applications.

Fig. 5 illustrates some examples of scheduling that lead to the worst-case response times of τ_3 when an internal error takes place. Scenarios (a), (b), and (c) correspond to the three last columns of Table 2, respectively. Let us focus on scenario (c) in the figure and compare with the values given by the analysis. By (4) and (5), $R_3^{int^1}(x, 8) = 5$ and $R_3^{int^0}(x, 8) = 18$, respectively. This is because the analysis takes into account two errors as for τ_2 and one internal error in τ_3. It is clear that τ_2 (or its recovery) cannot be interrupted by two errors since its period is 25 and $T_E = 8$. This *approximation* is the result of our conservative assumption, which says that any error always interrupts the task with the longest recovery time among all tasks that may interfere in the execution of τ_3 (in this case). The approximation is represented in the figure as if there were two consecutive executions of $\overline{\tau}_2$. Similar consequences of this assumption can also be seen in both (2) and (3), as noted earlier.

Up to now we have not been concerned with determining the best priority configuration so that the fault resilience of the task set is minimized. The difficulty in finding such an *optimal* priority configuration is twofold. First, there are a huge number of possible different arrangements of alternative task priorities. For a set of n tasks (one alternative task per primary task), this number is $n!$ since there are n possible priority values for the lowest priority task, $n-1$ for the second lowest priority task, and so on. Second, the search for the optimal priority configuration depends on the slack time available in higher priority levels, which, in turn, depends on the worst-case response times of tasks. The next section addresses this problem and presents an efficient solution to it.

5 The Priority Configuration Search Method

A description of the method to find out the optimal priority configuration regarding the analysis described earlier is given in this section. The main idea behind the method can be summarized as follows: Based on some properties of the analysis, an iterative procedure transforms a given priority configuration, say P_x, into another, say P_y, where P_y is a potential optimization of P_x. We say that P_y is an optimization of P_x if smaller values of T_E may be used on P_y without causing any task to miss its deadline. The procedure for improving a priority configuration is based on raising the priority of the alternative tasks that are causing the unschedulability of the task set. These tasks are called *dominant tasks* (Section 5.1). This iterative procedure stops when it is no longer possible to carry out any improvement. In order to search for optimized priority configurations from the initial configuration, a *partial order* of priority configurations is established (Section 5.2). The search is carried out in ascending order of priority configurations and it chooses those that could potentially reduce the value of T_E. The great advantage of this approach is that we do not need to consider all possible priority configurations, which would be too expensive. Only a small number of possibilities are checked.

5.1 Dominant Tasks

A given priority configuration, say P_x, has a minimum *allowed* value of T_E, denoted by the function $T_e(x)$. If any value less than $T_e(x)$ is attributed to T_E, some task may be unschedulable. In particular, if $T_E = T_e(x) - 1$, then there is at least a task τ_i in Γ such that $R_i(x, T_e(x) - 1) > D_i$. The tasks that *cause* the unschedulability of Γ under this circumstance are called *dominant* tasks. We distinguish two kinds of dominant task: 1-dominant and 2-dominant. A task τ_i is 1-dominant regarding the priority configuration P_x if $R_i^{int}(x, T_e(x) - 1) > D_i$. Two-dominant tasks are those tasks that may cause other tasks to miss their deadlines when $T_E = T_e(x) - 1$ because of the execution of their alternative tasks. Below, we define more formally the

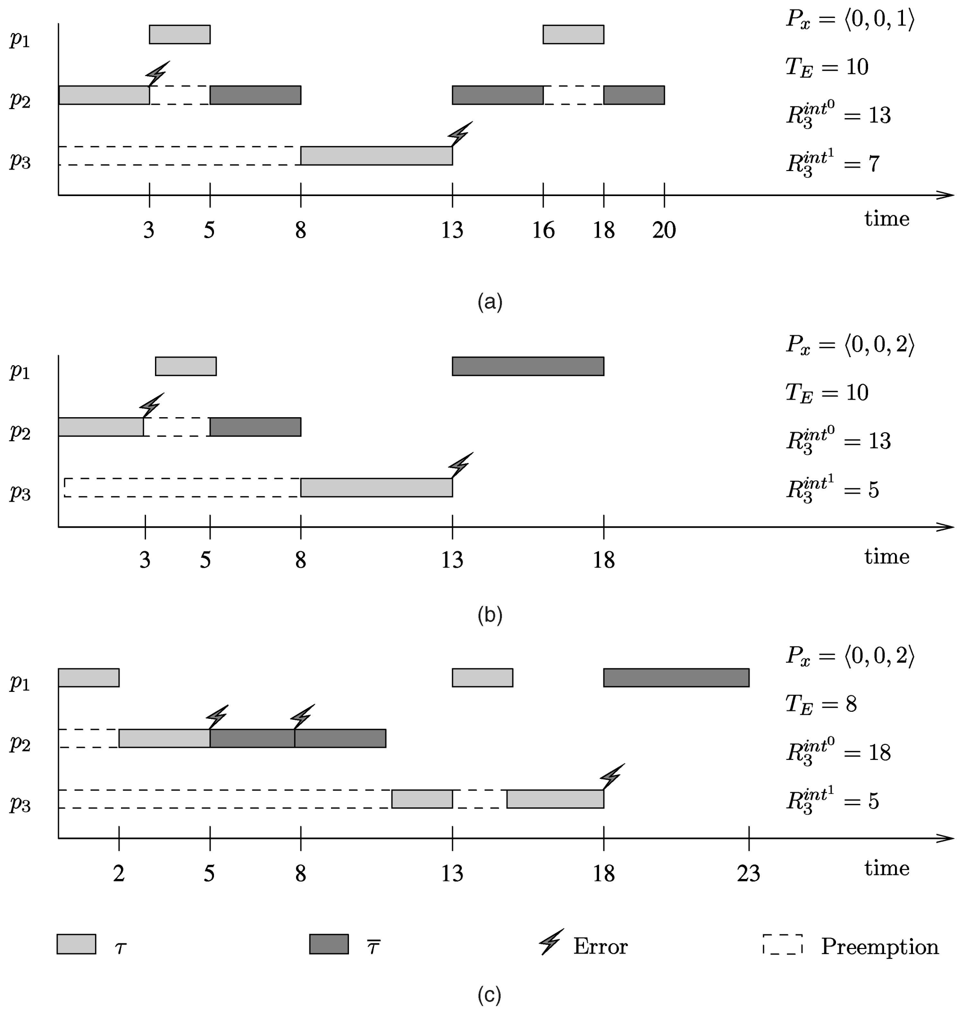

Fig. 5. Illustration of $R_3^{int}(x, T_E)$ regarding Table 2.

concept of dominant tasks and give the definition of their respective task sets.

Definition 5.1. *A task τ_i is a dominant task in relation to a priority configuration P_x if τ_i is 1-dominant, i.e., it belongs to $\mathcal{D}^1(x)$, or 2-dominant, i.e., it belongs to $\mathcal{D}^2(x)$, where*

$$\mathcal{D}^1(x) = \left\{\tau_i \in \Gamma \mid R_i^{int}(x, T_e(x) - 1) > D_i\right\}$$

and

$$\mathcal{D}^2(x) = \Big\{\tau_i \in \Gamma \mid \exists \tau_j \in \Gamma : \tau_i \in \mathrm{ip}(x, j) \wedge R_j^{ext}(x, T_e(x) - 1) > D_j \wedge \overline{C}_i = \max_{\tau_k \in \mathrm{ip}(x,j)} (\overline{C}_k)\Big\}.$$

Therefore, optimizing a priority configuration means reducing the worst-case response times due to internal errors of all 1-dominant tasks. Worst-case response times due to internal errors can only be reduced by increasing alternative task priorities. Table 2 illustrates this, where the $R_3^{int}(x, 10)$ is decreased when $\overline{p}_3$ is increased. This is because the size of $\mathrm{sp}(x, 3)$ is reduced and, so, the interference due to preemption over the execution of $\overline{\tau}_3$ is reduced as well. As for 2-dominant tasks, there is no space for optimization by raising the priorities of their alternative tasks. This is because doing so does not decrease the interference 2-dominant tasks cause in other tasks.

Table 3 shows the worst-case response times due to internal and external errors for three different configurations with regard to the task set of Table 1. The symbol "×" means that the task is unschedulable. The minimum allowed value of T_E in $\langle 0, 0, 0 \rangle$ is 11 time units, where τ_3 is 1-dominant. Increasing the priority of $\overline{\tau}_3$ by 1 leads to $\langle 0, 0, 1 \rangle$, which makes $T_e(\langle 0, 0, 1 \rangle) = 8$. Note that, in priority configuration $\langle 0, 0, 1 \rangle$, τ_3 is 2-dominant since it makes τ_2 unschedulable for $T_E = 7$. Since $R_2^{ext}(x, 7)$ cannot decrease by raising $\overline{p}_3$ further, from $\langle 0, 0, 1 \rangle$ no optimization is possible.

As can be noted from Table 3, the reduction of $R_i^{int}(x, T_E)$, where τ_i is some 1-dominant task, plays an important role in optimizing priority configurations. However, sometimes it is not possible to decrease $R_i^{int}(x, T_E)$ by raising the priorities of alternative tasks. For example, if $\overline{\tau}_2$ ran at the highest priority level in the priority configuration $P_x = \langle 0, 1, 0 \rangle$, $R_2^{int}(x, T_E)$ would still be eight time units. A similar situation occurs with $R_3^{int}(x, T_E)$ regarding priority configurations $\langle 0, 0, 1 \rangle$ and $\langle 0, 0, 2 \rangle$. Let us formalize this property by means a condition, which is a direct consequence of (4) and (5).

TABLE 3
Worst-Case Response Times Due to Internal and External Errors when $T_E = T_e(x) - 1$

		$P_0 = \langle 0,0,0 \rangle$		$P_1 = \langle 0,0,1 \rangle$		$P_2 = \langle 0,0,2 \rangle$	
		$T_E = 10$		$T_E = 7$		$T_E = 7$	
Task	Deadline	R_i^{int}	R_i^{ext}	R_i^{int}	R_i^{ext}	R_i^{int}	R_i^{ext}
τ_1	13	4	2	4	2	4	7
τ_2	25	8	7	11	×	11	×
τ_3	30	×	18	26	21	26	21

Consider a priority configuration, say P_x. $R_i^{int}(x, T_E)$ can be reduced by increasing $\overline{p}_i$ if the following *improvement* condition holds:

$$\mathrm{Cond}(x,i,j) \equiv \exists\, \tau_j \in \mathrm{sp}(x,i) : \left\lceil \frac{R_i^{int}(x,T_E)}{T_j} \right\rceil > \left\lceil \frac{R_i^{int^0}(x,T_E)}{T_j} \right\rceil . \quad (8)$$

This condition means that all preemption on the execution of $\overline{\tau}_i$ caused by the releases of τ_j can be eliminated if we set $\overline{p}_i \geq p_j$. We use the predicate above to avoid checking all configuration priorities in the optimization procedure. Only those that may reduce the values of 1-dominant task worst-case response times due to internal errors (where the predicate is true) need to be checked. It is important to note that this condition is necessary (but not sufficient) to optimize priority configurations. The next section presents the method used for such an optimization.

5.2 Search Graph and Search Path

Consider tasks τ_i and τ_j in Γ, a given priority configuration P_x and $T_E > 0$. Raising the priority of $\overline{\tau}_i$, as we have seen, may decrease the value of $R_i^{int}(x, T_E)$, but cannot decrease the value of $R_i^{ext}(x, T_E)$. Also, if $\tau_i \in \mathrm{ip}(x, j) - \mathrm{hp}(j)$, raising the priority of $\overline{\tau}_i$ may increase the value of $R_j^{ext}(x, T_E)$. Hence, in general, we can say that the maximum worst-case response times considering internal errors and the minimum worst-case response times due to external errors is when all alternative tasks run at the same priority level as their respective primary tasks, i.e., when the priority configuration equals $\langle 0, 0, \ldots, 0 \rangle$. Conversely, when all alternative tasks run with the highest possible priority, i.e., priority configuration $\langle 0, 1, \ldots, n-1 \rangle$, we have the minimum worst-case response times due to internal errors, but the maximum values of worst-case response times due to external errors. As the schedulability of any task τ_i is given by the maximum of $R_i^{int}(x, T_E)$ and $R_i^{ext}(x, T_E)$, we have to search the optimal priority configuration in the interval $\langle 0, 0, \ldots, 0 \rangle$ and $\langle 0, 1, \ldots, n-1 \rangle$. Based on this observation, let us order the set of all possible priority configurations by means of a direct acyclic graph, the *search graph*, where the priorities configurations $\langle 0, 0, \ldots, 0 \rangle$ and $\langle 0, 1, \ldots, n-1 \rangle$ are in the first and last position of such an order, respectively.

Definition 5.2. *A search graph $SG = \{V, E\}$ is a direct acyclic graph. Its vertex set is a set of $n!$ vertices, $V = \{v_0, v_1, \ldots, v_{n!-1}\}$, where each v_x is labeled with the priority configuration P_x. Its edge set is defined as*

$$E = \{(v_x, v_y) \in V \times V \mid \exists_1 j,\ \forall\, i \neq j : h_{x,i} = h_{y,i} \wedge h_{x,j} < h_{y,j}\}.$$

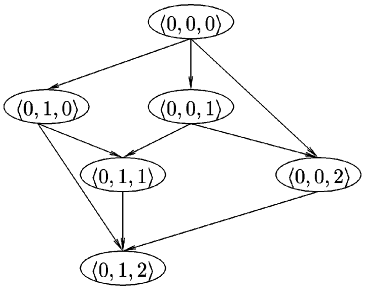

Fig. 6. The search graph for a set of three tasks.

Fig. 6 illustrates the search graph for a set of three tasks. It can be seen from the graph that the vertex labeled $\langle 0, 0, \ldots, 0 \rangle$ does not have any incoming edges and the vertex labeled $\langle 0, 1, \ldots, n-1 \rangle$ does not have any outcome edges. We name these vertices the *source* (v_0) and the *sink* vertices ($v_{n!-1}$), respectively. The order shown in the graph is expressed by the relation $\rightsquigarrow$, which is defined below.

Definition 5.3. *Let v_x and v_y be two vertices of a search graph SG. We say that v_y is reached from v_x, denoted $v_x \rightsquigarrow v_y$, if and only if $x = y$ or there is a path in SG from v_x to v_y. More formally, $x_x \rightsquigarrow v_y \Leftrightarrow (x = y) \vee (v_x, \ldots, v_y) \in SG$, where $(v_x, \ldots, v_y)$ is a path in SG.*

Consider the search graph presented in Fig. 6. Let v_x be a given vertex of the search graph and P_x its associated priority configuration. The problem we will now address can be stated as follows: *Is there any vertex v_y, where $v_x \rightsquigarrow v_y$, such that its associated priority configuration, P_y, makes the task set schedulable with $T_E < T_e(x)$?* Suppose, for instance, that $P_x = \langle 0, 0, 1 \rangle$ has two 1-dominant tasks, τ_2 and τ_3. In this scenario, only the sink vertex may optimize P_x provided that 1) the task set is schedulable in such a priority configuration with $T_E < T_e(x)$ and 2) it is possible to reduce $R_2^{int}(x, T_E)$ and $R_3^{int}(x, T_E)$. This is because it is necessary (but not sufficient!) that both $\overline{\tau}_2$ and $\overline{\tau}_3$ run with higher priorities than their priorities in P_x. Consider now that only τ_3 is dominant. Thus, we can guarantee that neither $\langle 0, 1, 0 \rangle$ nor $\langle 0, 1, 1 \rangle$ can optimize P_x since the priority of $\overline{\tau}_3$ is not increased in those priority configurations. If τ_3 is 1-dominant, $\langle 0, 0, 2 \rangle$ may be an optimization of P_x. However, if τ_3 is 2-dominant, no improvement is possible (recall that $\overline{\tau}_3$ is causing $R_1^{ext}(x, T_E) > D_1$ or $R_2^{ext}(x, T_E) > D_2$). Let us now take $P_x = \langle 0, 0, 0 \rangle$ and let us look at the possibilities to optimize P_x. If the only 1-dominant task is τ_3, either $\langle 0, 0, 1 \rangle$ or $\langle 0, 0, 2 \rangle$ may optimize P_x. Which one is the best choice? To answer this question, we have to look at the improvement condition, (8). If $\langle 0, 0, 1 \rangle$ does not satisfy this condition, we try $\langle 0, 0, 2 \rangle$. Otherwise, $\langle 0, 0, 1 \rangle$ is a better choice since we avoid increasing the priority of $\overline{\tau}_3$ too much. If other improvements are possible from $\langle 0, 0, 1 \rangle$, similar analysis will lead to $\langle 0, 0, 2 \rangle$ or even further to $\langle 0, 1, 2 \rangle$.

As can be seen, if we start searching for the optimal configuration from the source vertex, we only need to carry optimization with respect to increasing in priorities. The idea is to keep decreasing the worst-case response times

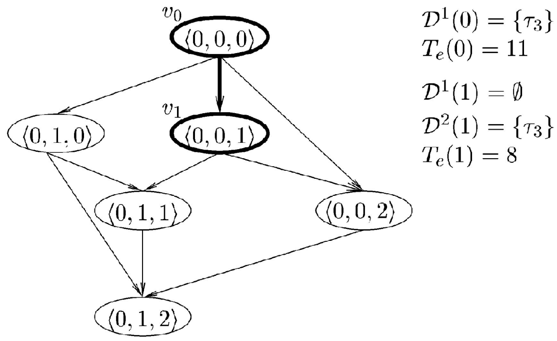

Fig. 7. The search path for the task set given by Table 1.

due to internal errors of 1-dominant tasks through a path from the source vertex. The last vertex of this path is one that can no longer be improved (either because it is the sink vertex or because the improvement condition is not true). We call this path the *search path*.

Definition 5.4. *A search path $SP = (v_0, v_1, \ldots, v_w)$ is any path in SG beginning from the source vertex such that, for all edges $(v_x, v_y) \in SP$, there is a 1-dominant task τ_i with regard to P_x such that*

$$R_i^{int}(x, T_e(x) - 1) > R_i^{int}(y, T_e(x) - 1) \tag{9}$$

and

$$h_{y,i} = \min_{(v_x, v_z) \in SG} (h_{z,i}). \tag{10}$$

If an edge belongs to a search path, it leads to a priority configuration which reduces the value of $R_i^{int}(x, T_e(x) - 1)$ for a given 1-dominant task τ_i (by (9)) and such a priority configuration has the minimum possible value of $\overline{p}_i$ (by (10)).

Consider the task set given in Table 1. Its search path, (v_0, v_1), is shown in Fig. 7. In $\langle 0,0,0\rangle$, we know that τ_3 is 1-dominant. Observing the definition of the search path, we move to $\langle 0,0,1\rangle$. Note that $\langle 0,0,1\rangle$ is the last priority configuration in the path since there is no other 1-dominant task. Also, observe that, even if τ_3 were 1-dominant in this priority configuration, its worst-case response time could not be decreased (recall Table 3 and the improvement condition).

Summing up, in order to find out an optimal priority configuration one has to follow a search path. This is formalized by the theorem below.

Theorem 5.1. *Consider Γ a fixed-priority scheduled set of primary tasks and their respective alternative tasks. Suppose that Γ is subject to faults so that the minimum time between error occurrences is bounded by $T_E > 0$. Let $SP = (v_0, v_1, \ldots, v_w)$ be a search path in a search graph SG as for tasks in Γ. The priority configuration P_x such that $T_e(x) = \min_{\forall v_z \in SP}(T_e(z))$ is the minimal value of T_E such that $R_i(x, T_e(x)) < D_i$ for any task $\tau_i \in \Gamma$.*

Proof. Assume by contradiction that there is a priority configuration $P_y \neq P_x$ such that $T_e(y) < T_e(x)$ and $R_i(y, T_e(y)) < D_i$ for any task $\tau_i \in \Gamma$. If $v_y \in SP$, then the proof is trivial. Consider that $v_y \notin SP$. This means that: 1) $\forall \tau_i \in \mathcal{D}^1(x) : h_{y,i} > h_{x,i}$ and 2) $\forall \tau_j \in \mathcal{D}^2(x) : h_{y,j} < h_{x,j}$ since these conditions are necessary for decreasing the value of $T_E = T_e(x)$. Consider the path $\mathcal{P} = (v_x, \ldots, v_w) \subset SP$. See Fig. 8, where the dotted line represents the search graph and the dashed line represents the search path. By the definition of the search path, all vertices in $\mathcal{P}$ have increased the priority of the alternative task of some dominant task in $\mathcal{D}^1(x)$ and, from P_w, it is no longer possible to reduce any dominant task worst-case response time due to internal errors by increasing the priorities of alternative tasks. Since P_y exists (by assumption), $T_e(x)$ (by definition) is minimum in SP and 1) holds, we conclude that $\mathcal{D}^1(x) = \emptyset$. Now, consider $\mathcal{D}^2(x) \neq \emptyset$. Without loss of generality, consider some $\tau_j \in \mathcal{D}^2(x)$. Thus, there is an edge (v_u, v_s) in SP such that $h_{y,j} = h_{u,j}$, making the value $h_{s,j}$ too high. By (10), $h_{s,j}$ is the minimum necessary to decrease $R_j^{int}(u, T_e(u) - 1)$ (note that $\tau_j \in \mathcal{D}^1(u)$). Any priority configuration with the same value of $h_{u,j}$ cannot be schedulable using $T_E < T_e(u)$ since τ_j is a dominant task in P_u. Therefore, since $T_e(x)$ is minimum in SP and $(v_u, v_s) \in SP$, $T_e(x) \leq T_e(u) \leq T_e(y)$, which provides the contradiction. □

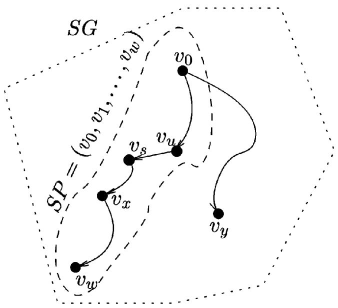

Fig. 8. Illustration that the search path contains the vertex labeled with the optimal priority configuration.

6 The Algorithm

Theorem 5.1 proves the correctness of the method for finding out the optimal configuration based on the concepts of search graph and search path. This section presents an algorithm to implement such a method. As we will see, its execution is equivalent to traversing a search graph through a search path up to a point at which some task becomes 2-dominant. However, it is not necessary to use the implementation of the search graph itself. This approach would be too expensive since the search graph has $n!$ vertices. The intuition behind the algorithm is to make $T_E = T_e(x) - 1$ for a given priority configuration P_x and then to look for a priority configuration P_y (where $v_x \rightsquigarrow v_y$) which makes the task set schedulable with such a value of T_E. If such a P_y exists, the algorithm finds it. Otherwise, the algorithm stops. This procedure is iterative and starts with $P_x = \langle 0, 0, \ldots, 0\rangle$.

The algorithm is straightforward (see Fig. 9). First of all, some initialization is done in lines 1-2. Then, the lower bound L on T_E and the minimum value for T_E regarding the initial configuration are calculated (lines 3, 4, and 5, respectively). The value of L is set to $1 + \max_{\forall \tau_j \in \Gamma}(\overline{C}_j)$. This

```
Priority Configuration Search (PCS)
(1)  $p_i \leftarrow \mathsf{FP}(\Gamma)$, $i = 1, 2, \ldots, n$
(2)  $P_x \leftarrow \langle 0, \ldots, 0 \rangle$; $P_x^* \leftarrow P_x$
(3)  $L \leftarrow 1 + \max_{\forall \tau_j \in \Gamma}(\overline{C}_j)$
(4)  $T_E \leftarrow T_e(x)$; $T_E^* \leftarrow T_E$
(5)  while TRUE
(6)      calculate $R_i(x, T_E)$, $i = 1, 2, \ldots, n$
(7)      if $(\forall \tau_i \in \Gamma : R_i(x, T_E) \leq D_i)$
(8)          $P_x^* \leftarrow P_x$
(9)          $T_E^* \leftarrow T_E$
(10)         $T_E \leftarrow T_E - 1$
(11)         if $(T_E < L)$ exit while
(12)     else
(13)         if $(\mathcal{D}^2(x) \neq \emptyset)$ exit while
(14)         let $\tau_i$ be a task in $\mathcal{D}^1(x)$
(15)         let PromotionSet $= \{\tau_j \in \Gamma \,|\, \mathsf{Cond}(x, i, j)\}$
(16)         if (PromotionSet $\neq \emptyset$)
(17)             $h_{x,i} \leftarrow \min_{\forall \tau_j \in \mathtt{PromotionSet}}(p_j) - p_i$
(18)             if (|PromotionSet| $= 1$) $T_E \leftarrow \mathsf{MIN}(T_e(x), T_E)$
(19)         else
(20)             exit while
(21)         endif
(22)     endif
(23) endwhile
(24) $P_x \leftarrow P_x^*$
(25) $T_E \leftarrow T_E^*$
```

Fig. 9. The optimal priority configuration search algorithm.

is because, if T_E assumes lower values, in the worst-case, the same alternative task is always interrupted by an error. This means that the task with the longest recovery cost never completes, which implies that the task set is unschedulable. The initial priority configuration P_x and the found value of $T_e(x)$ is saved in variables P_x^* and T_E^*, respectively. This is necessary in cases where the task set is unschedulable in $\langle 0, 0, \ldots, 0 \rangle$. These values will change throughout the execution of the search algorithm if some optimal priority configuration is found. Otherwise, $P_0 = \langle 0, 0, \ldots, 0 \rangle$ and $T_e(0)$ are returned as default values. After the initialization, the optimization procedure is carried out (lines 5-23) until no optimization is possible (lines 13 or 20) or $T_E < L$ (line 11).

The iterative search has two blocks, the save-block (lines 8-11) and the promotion-block (lines 13-21). Whenever the task set is schedulable, the save-block is executed in order to save both the last improved priority configuration and the minimum value found for T_E regarding such a priority configuration. Each execution of the save-block is followed by the execution of the promotion-block. This is because line 10 guarantees that the task set will not be schedulable in the next iteration.

Whenever the task set is considered unscheduled and there are no 2-dominant tasks, the promotion-block is executed. If there is some 2-dominant task, the algorithm stops. In line 14, some 1-dominant task is selected for promotion. Note that any 1-dominant task can be selected. Then, the improvement condition is checked since this is necessary for decreasing the worst-case response time due to internal errors of the selected dominant task. If there is no task that satisfies the improvement condition (i.e., PromotionSet is empty), the search stops and the last saved configuration is optimal. Otherwise, the promotion of the alternative task of the selected dominant task is carried out (line 17). Note that its alternative task priority is set to the lowest priority level, which allows a smaller value of the worst-case response times due to internal errors. Then, in line 18, a new value of T_E is calculated. This is necessary because, if PromotionSet is a unitary set, the promotion carried out in the earlier line may reduce the value of $T_e(x)$. The value of $T_e(x)$ may increase throughout the optimization process if the selected 1-dominant task becomes 2-dominant. In this case, the algorithm stops in the next iteration in line 13.

We have assumed up to now that the value $T_e(x)$ for any priority configuration P_x is available. Indeed, this function can be implemented straightforwardly as a binary search. The initial search interval can be set to $[L, \max_{\forall \tau_i}(D_i)]$. As we mentioned earlier, T_E cannot assume values less than L without compromising the schedulability of the task set. If $T_E \geq \max_{\forall \tau_i}(D_i)$, only one error occurrence within the longest response time of the task set may take place. If the task set is unschedulable with this maximum value, it will be unschedulable with errors occurring at any rate.

It is interesting to note that it is possible to improve the implementation of the algorithm by making two slight changes. The first is with respect to the implementation of the function $T_e(x)$. As can be seen, we only set a new value to T_E in line 18 if the priority configuration is optimized. Thus, we can reduce the search interval for the binary search to $[L, T_E]$, where T_E is its current value. The second modification is related to the choice of the dominant task in line 14. Although any 1-dominant task can be selected, it is preferable to select one, say τ_i, with the highest alternative task priority. This is because the possibility of reducing $R_i^{int}(x, T_E)$ is lower, which may lead to a smaller number of iterations when it is not possible to improve priority configurations.

6.1 Proof of Correctness and Complexity

In order to prove the correctness of the PCS algorithm, we have to show that 1) an optimal priority configuration is found (Theorem 6.1) and 2) the algorithm stops (Theorem 6.2). Before showing this, the equivalence between search path and the execution of the algorithm is shown (Lemma 6.1).

Lemma 6.1. *Let $S = (P_0, P_1, \ldots, P_w)$ be the sequence of priority configurations generated by the algorithm PCS. S is a prefix of or is equal to the label sequence of a search path $SP = (v_0, v_1, \ldots, v_w)$.*

Proof. First, suppose that, during the execution of the algorithm, no task becomes 2-dominant. In this case, we prove by induction that S is the exact sequence of the vertices in SP. The induction is on the number of times that the algorithm executes the promotion-block. The base case is the first execution of the promotion-block. Note that the execution of the save-block does not change the priority configuration. It is clear that v_0, labeled $P_0 = \langle 0, 0, \ldots, 0 \rangle$, belongs to the search path by definition. Since $\mathcal{D}^2(0) = \emptyset$, during the first execution of the promotion-block either the algorithm stops (PromotionSet $= \emptyset$) and, so, $|S| = |SP| = 1$ or a promotion is carried out. Let P_1 be the second priority configuration in S. By definition of the search path, P_1 is the label of v_1 since (10) corresponds to the execution of line 17 and (9) holds because $\texttt{PromotionSet} \neq \emptyset$. Hence, the base case holds. Now, suppose that a given $P_x \in S$ is the label of v_x so that $(P_x, P_w) \in S$ and $(v_x, v'_w) \in SP$. By the algorithm, this means that line 17 was executed and the promotion of a dominant task was carried out. By a similar argument made for the base case, this promotion is equivalent to traversing an edge in SP and, so, P_w is the label of v'_w, i.e., $v'_w = v_w$. Therefore, if no 2-dominant task is found during the execution of the algorithm, the sequence S is the exact sequence of a given SP. Now, consider that some 2-dominant task is found in some priority configuration $P_x \in S$. As a result, by line 13, the algorithm stops in P_x. Observe that, in this case, P_x is the last priority configuration in S and the first one such that $\mathcal{D}^2(x) \neq \emptyset$. Hence, by the induction above, it is clear that there exists $v_x \in SP$ and P_x is the label of v_x. As a result, the sequence $(P_0, P_1, \ldots, P_x)$ is the label of the vertices of the subsequence $(v_0, v_1, \ldots, v_x) \in SP$. Therefore, S is a prefix of the label sequence of SP, as required. □

Theorem 6.1. *The algorithm PCS finds an optimal configuration priority regarding the proposed analysis.*

Proof. Based on the results of Lemma 6.1 and Theorem 5.1, we only need to show that the last saved configuration corresponds to the optimal one. By the algorithm, $P_0 = \langle 0, 0, \ldots, 0 \rangle$ is the first saved priority configuration. Assume first that there is no other execution of the save-block. This is because the algorithm stops in line 11, 13, or 20, which means that no optimization was possible from P_0. In other words, for any other priority configuration reached by the algorithm, say P_z, the task set is unschedulable with $T_E = T_e(0) - 1$. As a result, P_0 is optimal. Now, assume that there are at least two (consecutively) saved priority configurations, P_y and P_x say. By the construction of the algorithm, the values attributed to T_E do not increase throughout the iterations. Thus, $T_e(x) \leq T_e(y)$. This implies that the last saved priority configuration has the minimum (optimal) value of T_E, as required. □

Theorem 6.2. *The algorithm PCS stops with at most $n(n-1)$ iterations.*

Proof. By construction of the algorithm, it stops either when $\texttt{PromotionSet} = \emptyset$ or when $L > T_E$ or when the algorithm reaches some configuration with a 2-dominant task. Let us assume that there is some task set that does not have any 2-dominant task for all possible priority configurations. In this case, the algorithm never stops due to 2-dominant tasks. As $L \leq T_E$ is a precondition of the algorithm which is guaranteed to be true throughout the iterations (line 11), we have to prove that the condition $\texttt{PromotionSet} = \emptyset$ is eventually true at most at the iteration number $n(n-1)$. Our proof will be by looking at the longest possible search path in the search graph (using the result of Lemma 6.1). By the definition of the search graph, the longest path is $(v_0, v_1, \ldots, v_{n!-1})$. If this path is the longest one, it is characterized by increasing one task priority level per edge. Thus, for the lowest priority task, we have to traverse n edges, for the second lowest priority task, $n-1$ edges, and so on. The maximum number of traversed edges is

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

The worst case is when there is only one 1-dominant task in each vertex of the search path and each promotion of its priority makes the task set schedulable (i.e., each execution of the promotion-block is followed by one execution of the save-block). As a result, two iterations per priority promotion are necessary, one to promote the priority of a dominant task and the other to save the priority configuration. As each promotion is equivalent to traversing an edge of the search graph, the maximum number of iterations is twice the maximum number of traversed edges. Also, for the priority configuration $\langle 0, 1, \ldots, n-1 \rangle$, $\texttt{PromotionSet} = \emptyset$ since all alternative tasks are executing in the highest priority level. Therefore, there are at most $n(n-1)$ iterations. □

The time complexity of the search is determined by the worst-case number of iterations, i.e., $O(n^2)$. This can be considered a significant result since we reduced the search space from $n!$ to n^2. The whole algorithm has time complexity nearly $O(n^4)$ since, in the worst case, we have to calculate the response time (line 23) n^2 times and carry out the sensitivity analysis (function $T_e(x)$—line 18) whenever the promotion block is executed.

7 Assessment of Effectiveness

This section characterizes the applicability of the described approach by simulation, where 18,000 task sets (10 tasks per task set) were generated. The values of worst-case computation time and recovery costs of each task set were generated according to an exponential distribution with mean $U/10$, where U is the processor utilization. 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 used the deadline monotonic algorithm to assign the priorities of primary tasks. We did not consider processor utilization 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).

The points in Fig. 10 represent the obtained gain in terms of fault resilience of the task sets. This gain was measured by comparing the values of $T_e(0)$ and $T_e(x)$, where P_x is the optimal priority configuration found by the algorithm of Fig. 9. In other words, the gain was measured as $\frac{T_e(0)-T_e(x)}{T_e(0)}$. The line plotted in the graph represents the mean gain obtained by the proposed approach. As can be seen from the figure, the obtained reductions on T_E are, on average, low (up to 10 percent). However, high gains may be obtained in some cases, mainly when processor utilization is greater than 0.4. In our experiments, we found gains of up

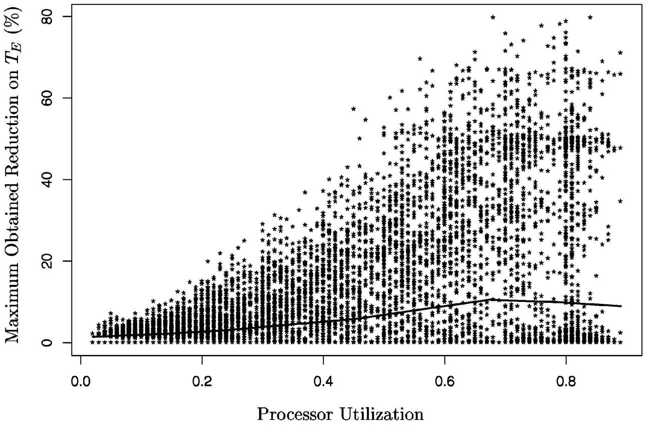

Fig. 10. Obtained reduction of the time between error occurrences.

to 78 percent. The lower gains for lower processor utilizations 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 lower values of T_E. Promoting the priority of alternative tasks for these cases, therefore, has lower impact in fault resilience since it is already high.

8 Related Work

Several scheduling mechanisms for fault-tolerant purposes can be found in the literature. Most of them either assume a restrictive task/fault model and, so, lack flexibility or impose restrictions on the way in which fault tolerance is carried out. Moreover, most approaches that are summarized below assume that alternative tasks run with the same priorities as their respective primaries. Here, we refer mainly to approaches designed for scheduling fault-tolerant uniprocessor systems.

One of the first scheduling mechanisms for fault tolerance purposes was described by Liestman and Campbell [13]. This mechanism only deals with periodic tasks, whose periods have to be multiples of each other. Another restriction of this mechanism is that the execution times of alternative tasks have to be shorter than the execution times of their respective primaries.

The approach presented by Ghosh et al. [8] considers only reexecution of faulty tasks to tolerate transient faults. As it is based on the Rate Monotonic priority assignment policy [15], its disadvantages are inherited by the proposed mechanism. Another approach that uses similar assumptions has been proposed [2]. This approach carries out fault tolerance by periodically checkpointing tasks in different nodes and, so, both transient and permanent faults can be tolerated.

Kandasamy et al. [10] describe a recovery technique that tolerates transient faults in an offline 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 transient faults which is independent of the schedulability analysis being used has been described by Ghosh et al. [7]. However, it is considered that recovery is carried out only by reexecution of tasks.

Recently, an EDF-based scheduling approach, 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 times for executing task recoveries given that a *fault pattern* or the maximum number of faults per task is known a priori. This means that the assumed error occurrences of the task set is represented by several parameters, the fault pattern.

Another EDF-based scheduling approach for supporting fault-tolerant systems has been proposed by Caccamo and Buttazzo [6]. 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 a backup job. The primary job is scheduled online and provides high-quality service, while the backup job is scheduled offline and provides acceptable services.

In this work, we use fixed-priority-based scheduling for both primaries and alternative tasks. We represent, using T_E, the extent that the task set is subject to errors. This unique representation parameter has allowed us to derive a simple but effective optimization algorithm. Also, we do not restrict the number of errors that may interrupt primary and alternative tasks since this number is a function of T_E. Yet, T_E can be used to establish probabilistic scheduling guarantees under the assumption that error occurrences follow a Poison distribution. This result has been shown by Burns et al. [5] and is the main motivation for our approach. Our work is a generalization of the approach proposed by Burns et al. [3]. As we have seen, 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 (2).

The priority assignment problem has not been satisfactorily addressed for fault tolerance purposes. To the best of our knowledge, the approach presented by Ramos-Thuel and Strosnider [19] is the closest work related to ours. It is based on the concept of transient server and its basic idea is to explore the spare capacity of the task set to determine the maximum server capacity at each priority level. This information is used for *online* scheduling decisions in the case of error occurrences. The authors have not presented a reasonable way of determining the server periods, though. Unlike them, we are concerned with *static* scheduling, where the schedulability of the whole task set must be guaranteed.

9 Conclusion

In this paper, we have addressed the problem of providing suitable schedulability analysis for fault-tolerant hard real-time systems. Two major contributions have been presented. First, response time analysis which allows task recovery to be carried out at higher priority levels has been described. Second, an efficient algorithm that assigns the optimal priority configuration for task recovery has been presented. By optimal, we mean the priority configuration which maximizes the fault resilience of task sets as for the described analysis. The proposed algorithm uses the properties of the analysis to reduce the space of search from $O(n!)$ to $O(n^2)$. To the best of our knowledge, the problem of maximizing fault resilience by priority manipulation has not been addressed before in the context of fixed-priority scheduling. We have shown by simulation that significant gains in terms of fault resilience may be obtained by applying our approach.

ACKNOWLEDGMENTS

George M. de A. Lima was supported by CAPES/Brazil under grant BEX1438/98-0. The authors would like to thank to Guillem Bernat and the anonymous referees for their comments about this work.

REFERENCES

[1] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings, "Applying New Scheduling Theory to Static PriorityPre-Emptive Scheduling," *Software Eng. J.,* vol. 8, no. 5, pp. 284-292, 1993.

[2] A.A. Bertossi and L.V. Mancini, "Scheduling Algorithms for Fault-Tolerance in Hard-Real-Time Systems," *Real-Time Systems,* vol. 7, no. 3, pp. 229-245, 1994.

[3] A. Burns, R.I. Davis, and S. Punnekkat, "Feasibility Analysis of Fault-Tolerant Real-Time Task Sets," *Proc. Euromicro Real-Time Systems Workshop,* pp. 29-33, 1996.

[4] A. Burns, M. Nicholson, K. Tindell, and N. Zhang, "Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point Distributed System," *Proc. Workshop Parallel and Distributed Real-Time Systems,* pp. 11-20, 1993.

[5] A. Burns, S. Punnekkat, L. Stringini, and D. Wright, "Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems," *Proc. Seventh Int'l Working Conf. Dependable Computing for Critical Application,* pp. 339-356, 1999.

[6] M. Caccamo and G. Buttazzo, "Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems," *Proc. Fifth Conf. Real-Time Computing and Applications (RTCSA),* pp. 223-231, 1998.

[7] S. Ghosh, R. Melhem, and D. Mossé, "Enhancing Real-Time Schedules to Tolerate Transient Faults," *Proc. 16th Real-Time Systems Symp. (RTSS),* pp. 120-129, 1995.

[8] S. Ghosh, R. Melhem, and D. Mossé, "Fault-Tolerant Rate-Monotonic Scheduling," *Proc. IFIP Int'l Conf. Dependable Computing for Critical Applications,* 1997.

[9] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer J. (British Computer Soc.),* vol. 29, no. 5, pp. 390-395, 1996.

[10] N. Kandasamy, J.P. Hayes, and B.T. Murray, "Tolerating Transient Faults in Statically Scheduled Safety-Critical Embedded Systems," *Proc. 18th IEEE Symp. Reliable Distributed Systems (SRDS),* pp. 212-221, 1999.

[11] J.C. Laprie, *Dependability: Basic Concepts and Terminology, vol. 5, Dependable Computing and Fault-Tolerant Systems.* Springer-Verlag, 1992.

[12] F. Liberato, R. Melhem, and D. Mossé, "Tolerance to Mutiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems," *IEEE Trans. Computers,* vol. 49, no. 9, pp. 906-914, Sept. 2000.

[13] L. Liestman and R.H. Campbell, "A Fault-Tolerant Scheduling Problem," *IEEE Trans. Software Eng.,* vol. 12, no. 11, pp. 1089-1095, 1986.

[14] G.M.A. Lima and A. Burns, "An Effective Schedulability Analysis for Fault-Tolerant Hard Real-Time Systems," *Proc. 13th Euromicro Conf. Real-Time Systems,* pp. 209-216, 2001.

[15] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multi-program in a Hard Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 40-61, 1973.

[16] J.C. Palencia and M.G. Harbour, "Schedulability Analysis for Tasks with Static and Dynamic Offsets," *Proc. 19th Real-Time Systems Symp. (RTSS),* pp. 26-37, 1998.

[17] J.C. Palencia and M.G. Harbour, "Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems," *Proc. 20th Real-Time Systems Symp. (RTSS),* pp. 328-339, 1999.

[18] S. Punnekkat, "Schedulability Analysis for Fault Tolerant Real-Time Systems," PhD thesis, Dept. of Computer Science, Univ. of York, 1997.

[19] S. Ramos-Thuel and J.K. Strosnider, "The Transient Server Approach to Scheduling Time-Critical Recovery Operations," *Proc. 12th Real-Time Systems Symp. (RTSS),* pp. 286-295, 1991.

[20] K. Tindell, A. Burns, and A. Wellings, "Analysis of Hard Real-Time Communications," *Real-Time Systems,* vol. 9, no. 2, pp. 147-171, 1995.

[21] K. Tindell and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems," *Microprocessing and Microprogramming, Euromicro J.,* vol. 40, pp. 117-134, 1994.

George M. de A. Lima graduated in computer science from the Federal University of Bahia (UFBA) in 1993 and received the MSc degree, also in computer science, from the State University of Campinas in 1996, both courses in Brazil. Over the period from 1996 to 1999, he took a temporary lectureship position at UFBA. During this period of time, he also worked at Distributed Systems Laboratory (LaSiD/UFBA) as a research assistant. He is currently pursuing the PhD degree at the University of York in the area of fault tolerance and real-time systems.

Alan Burns graduated in 1974 with a first class honours degree in mathematics from Sheffield University; he then received a DPhil degree from the Computer Science Department at the University of York. He has worked for many years on a number of different aspects of real-time systems engineering. After a short period of employment at UKAEA Research Centre, Harwell, he was appointed to a lectureship at Bradford University in 1979. He was subsequently promoted to senior lecturer in 1986. In January 1990, he took up a readership at the University of York in the Computer Science Department. During 1994, he was promoted to a personal chair. Since 1 July 1999, he has been head of the Computer Science Department at York. His research activities have covered a number of aspects of real-time and safety critical systems including requirements for such systems, the specification of safety and timing needs, system architectures appropriate for the design process, the assessment of languages for use in the real-time safety critical domain, distributed operating systems, the formal specification of scheduling algorithms and implementation strategies, and the design of dependable user interfaces to safety critical applications. Professor Burns, together with Professor Wellings, heads the Real-Time Systems research group at the University of York—one of the largest research groups in this area in the world and with a strong international reputation. He has authored/coauthored more than 350 papers/reports and eight books. Most of these are in the Ada or real-time area. His teaching activities include courses in operating systems, scheduling, and real-time systems. He is a senior member of the IEEE.