Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems

Frank Liberato, Rami Melhem, *Fellow*, *IEEE*, and
Daniel Mossé, *Member*, *IEEE Computer Society*

Abstract—Real-time systems are being increasingly used in several applications which are time-critical in nature. Fault tolerance is an essential requirement of such systems, due to the catastrophic consequences of not tolerating faults. In this paper, we study a scheme that guarantees the timely recovery from multiple faults within hard real-time constraints in uniprocessor systems. Assuming earliest-deadline-first scheduling (EDF) for aperiodic preemptive tasks, we develop a necessary and sufficient feasibility-check algorithm for fault-tolerant scheduling with complexity $O(n^2 \cdot k)$, where n is the number of tasks to be scheduled and k is the maximum number of faults to be tolerated.

Index Terms—Real-time scheduling, earliest-deadline first, fault-tolerant schedules, fault recovery.

1 Introduction

The interest in embedded systems has been growing steadily in the recent past, especially those systems in which timing constraints are essential for the correct execution of the systems. Examples include autopilot systems, satellite and launch vehicle control, as well as robots, whether in collaborating teams or not. For some of these systems, termed *hard real-time systems* (HRTSs), the consequences of missing a deadline may be catastrophic. The ability to tolerate faults in HRTSs is crucial since a task can potentially miss a deadline when faults occur. In case of a fault, a deadline can be missed if the time taken for recovery from faults is not taken into account during the phase in which tasks are submitted/accepted to the system. Clearly, accounting for recovery from faults is an essential requirement of HRTSs.

When dealing with such HRTSs, permanent faults can be tolerated by using hot-standby spares [14] or they can be masked by modular redundancy techniques [25]. In addition to permanent faults, tolerance to transient faults is very important since it has been shown to occur much more frequently than permanent faults [11], [12], [5]. In a study, an orbiting satellite containing a microelectronics test system was used to measure error rates in various semiconductor devices including microprocessor systems [4]. The number of errors, caused by protons and cosmic ray ions, mostly ranged between 1 and 15 in 15-minute intervals and was measured to be as high as 35 in such intervals. More examples of such safety critical applications can be found in [17]. Transient faults can be dealt with through temporal redundancy, that is, allowing extra time (slack) in the schedule to reexecute the task or to execute a recovery block [10].

The problem solved in this paper is as follows: Given a set of n aperiodic tasks, $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$, we seek to determine if each task in the set $\mathcal{T}$ is able to complete execution before its deadline under EDF scheduling, even if the system has to recover from (at most) k faults. We consider a uniprocessor system and assume that each task may be subjected to multiple transient faults.

A simple solution would be to check the feasibility of each of the schedules generated by the $O(n^k)$ possible *combination* of faults using the approach described in [18] for each schedule. The high complexity of this scheme provides the impetus for searching for a more efficient solution. The solution presented in this paper develops an optimal (necessary and sufficient) feasibility check that runs in $O(n^2 \cdot k)$ time in the worst case.

Although we consider aperiodic tasks, we note that the technique presented in this paper can be used to verify the fault tolerance capabilities of a set of periodic tasks by considering each instance of a periodic task as an aperiodic task within the Least Common Multiple of the periods of all the periodic tasks. Moreover, scheduling aperiodic tasks is the basis for scheduling periodic tasks in *frame-based* systems, where a set of tasks (usually having precedence constraints) is invoked at regular time intervals. This type of systems is commonly used in practice because of its simplicity. For example, in tracking/collision avoidance applications, motion detection, recognition/verification, trajectory estimation, and computation of time to contact are usually component subtasks within a given frame (period) [3]. Similarly, a real-time image magnification task might go through the steps of nonlinear image interpolation, contrast enhancement, noise suppression, and image extrapolation during each period [20]. Even though these are periodic tasks, the system period is unique and, therefore, the scheduling of the each instance (corresponding in our nomenclature to an "aperiodic" task) can be done within a specific time interval.

• *The authors are with the Department of Computer Science, The University of Pittsburgh, Pittsburgh, PA 15260.*
E-mail: {melhem, mosse}@cs.pitt.edu.

Manuscript received 22 June 1999; revised 19 Nov. 1999; accepted 14 June 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 110106.

The rest of this paper is organized as follows: In Section 2, we present the model and notation for the aperiodic, fault-tolerant scheduling problem. In Section 3, we introduce an auxiliary function that will aid in the presentation of our solution. In Section 4, we describe the feasibility tests for a set of tasks under a specific fault pattern and generalize it in Section 5 for any fault pattern, examining the worst case behavior with respect to k faults. In Section 6, we survey some strongly related work and, in Section 7, we finalize the paper with concluding remarks.

2 Model and Notation

We consider a uniprocessor system to which we submit a set $\mathcal{T}$ of n tasks: $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$. A task τ_i is modeled by a tuple $\tau_i = \langle R_i, C_i, D_i \rangle$, where R_i is the ready time (earliest start time of the task), D_i is the deadline, and C_i is the maximum computation time (also called worst case execution time). The set of tasks that become ready at a given time t is denoted by $RS(\mathcal{T}, t)$. That is, $RS(\mathcal{T}, t) = \{\tau_i \in \mathcal{T} : R_i = t\}$.

We assume EDF scheduling with ties in deadlines broken arbitrarily. The schedule of $\mathcal{T}$ is described by the function

$$EDF(\mathcal{T}, t) = \begin{cases} \tau_i & \text{if EDF schedules } \tau_i \text{ between } t \text{ and } t+1 \\ \uparrow & \text{if EDF does not schedule any task between } t \text{ and } t+1, \end{cases}$$

where $t = 0, 1, 2, \ldots$ represents time. We will use $EDF(\mathcal{T})$ to refer to the EDF schedule of $\mathcal{T}$.

We define e_i to be the time at which task τ_i completes execution in $EDF(\mathcal{T})$ and we define the function $slack(t_1, t_2)$ to be the number of free slots between $t = t_1$ and $t = t_2$ in $EDF(\mathcal{T})$. That is, the number of slots for which $EDF(\mathcal{T}, t) = \uparrow$ (excluding the slot that starts at t_2). $EDF(\mathcal{T})$ is said to be feasible if $e_i \leq D_i$ for all $i = 1, \ldots, n$.

It is assumed that faults can be detected at the end of the execution of each task. The time required by the fault detection mechanism can be added to the worst case computation time C_i of the task and does not hinder the timeliness of the system. Many mechanisms have been proposed for fault detection at the user level, the operating system level, and the hardware level. At the user level, a common technique is to use consistency or sanity checks, which are procedures supplied by the user, to verify the correctness of the results [9], [28]. For example, using checksums, checking the range of the results or substituting a result back into the original equations can be used to detect a transient error.

Many mechanisms exist in operating systems and computer hardware for error detection and for triggering recovery. Examples are the detection of illegal opcode (caused by bus error or memory corruption), memory range violation, arithmetic exceptions, and various time-out mechanisms. Hardware duplication of resources can also be used for detecting faults through comparison of results. It should be noted, however, that while each of the mechanisms described above is designed for detecting specific types of faults, it has been long recognized that it is not possible for a fault detection mechanism to accomplish a perfect coverage over arbitrary types of faults.

When a fault is detected, the system enters a *recovery mode* where some recovery action must be performed before the task's deadline. We assume that a task τ_i recovers from a fault by executing a recovery block [10], [16], $\tau_{i,1}$, at the same priority of τ_i. A fault that occurs during the execution of $\tau_{i,1}$ is detected at the end of $\tau_{i,1}$ and is recovered from by invoking a second recovery block, $\tau_{i,2}$, and so on. It is assumed that the maximum time for a recovery block of τ_i to execute is V_i. The recovery blocks for each task may have a different execution time from the task itself; in other words, the recovery is not restricted to reexecution of the task. Recovery blocks can be used for avoiding common design bugs in code, for providing a less accurate result in view of the limited time available for recovery, or for loading a "safe" state onto memory from some stable source (across the network or from shielded memory).

We shall denote a pattern of faults over $\mathcal{T}$ as a set $\mathcal{F} = \{f_1, \ldots, f_n\}$ such that f_i is the number of times the task $\tau_i \in \mathcal{T}$ or its recovery blocks will fail before successful completion. We use $EDF^{\mathcal{F}}(\mathcal{T})$ to denote the EDF schedule of $\mathcal{T}$ under the fault pattern $\mathcal{F}$, that is, when τ_i is forced to execute f_i recovery blocks. $EDF^{\mathcal{F}}(\mathcal{T})$ is said to be feasible if, for all $i = 1, \ldots, n$, task τ_i and its f_i recovery blocks complete by D_i. Note that $EDF^{\mathcal{F}}(\mathcal{T})$ cannot be feasible if $D_i - R_i < C_i + f_i \cdot V_i$ for any i.

Given a task set $\mathcal{T}$ and a specific fault pattern $\mathcal{F}$, we define two functions. The first function, $W(\mathcal{T}, t)$, defines the amount of work (execution time) that remains to be completed at time t in $EDF(\mathcal{T})$. This work is generated by the tasks that became ready at or before time t, that is, by the tasks in $\{\tau_i \in \mathcal{T} : R_i \leq t\}$. Specifically,

$$W(\mathcal{T}, t) = \begin{cases} \sum_{\tau_i \in RS(\mathcal{T}, t)} C_i & \text{if } t = 0 \\ W(\mathcal{T}, t-1) \dot{-} 1 + \sum_{\tau_i \in RS(\mathcal{T}, t)} C_i & \text{for all } t = 1, 2, \ldots, \end{cases}$$

where the $\dot{-}$ operator is defined as $a \dot{-} b = max(a - b, 0)$. At a time t, any positive amount of work in $W(\mathcal{T}, t-1)$ decreases by one during the period between $t-1$ and t, while, when a task becomes ready at t, the work increases by the computation time of this task.

The second function, $W^{\mathcal{F}}(\mathcal{T}, t)$ is defined in a similar way except that we include time for the recovery of failed tasks at the point they would have completed in the fault-free schedule. Specifically,

$$W^{\mathcal{F}}(\mathcal{T}, t) = \begin{cases} \sum_{\tau_i \in RS(\mathcal{T}, t)} C_i & \text{if } t = 0 \\ W^{\mathcal{F}}(\mathcal{T}, t-1) \dot{-} 1 + \sum_{\tau_i \in RS(\mathcal{T}, t)} C_i + f_j \cdot V_j & \text{if } t = e_j \text{ for some } \tau_j \in \mathcal{T} \\ W^{\mathcal{F}}(\mathcal{T}, t-1) \dot{-} 1 + \sum_{\tau_i \in RS(\mathcal{T}, t)} C_i & \text{otherwise.} \end{cases}$$

The two functions defined above will be used to reason about the extra work needed to recover from faults. Note that, although task τ_i may complete at a time different than e_i in $EDF^{\mathcal{F}}(\mathcal{T})$, the function $W^{\mathcal{F}}$ has the important property that it is equal to zero only at the beginning of an idle time slot in $EDF^{\mathcal{F}}(\mathcal{T})$. This, and other properties of the two functions defined above, are given next.

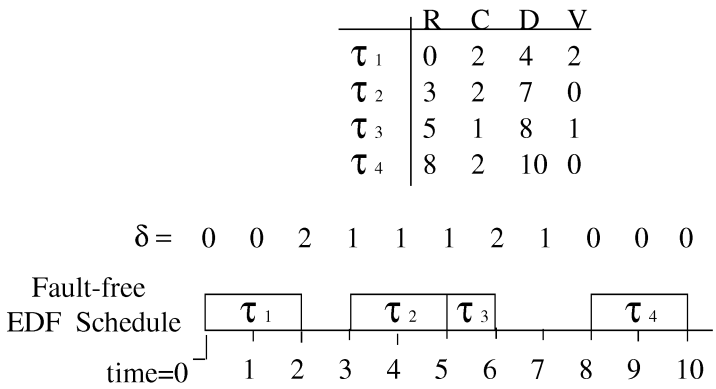

Fig. 1. Task Set, EDF schedule and δ values for $f_1 = 1,\ f_3 = 1,\ f_2 = f_4 = 0$.

Property 1. *$W(\mathcal{T},t) = 0$ if and only if $EDF(\mathcal{T},t) = \uparrow$. That is, $W(\mathcal{T},t) = 0$ if and only if there is no work to be done at time t in $EDF(\mathcal{T})$, which means that any task with $R_i \leq t$ finishes at or before time t in the fault-free case.*

Property 2. *$W^{\mathcal{F}}(\mathcal{T},t) = 0$ if and only if there is no work to be done at time t in $EDF^{\mathcal{F}}(\mathcal{T})$, which means that any task with $R_i \leq t$ finishes at or before time t when the tasks are subject to the fault pattern $\mathcal{F}$.*

Property 3. *$W^{\mathcal{F}}(\mathcal{T},t) \geq W(\mathcal{T},t)$. That is, the amount of work incurred when faults are present is never smaller than the amount of work in the fault-free case.*

Property 4. *$t = e_i \Rightarrow W(\mathcal{T},t-1) > 0$. That is, the slot before the end of a task is never idle.*

The above four properties follow directly from the definition of $W(\mathcal{T},t)$ and $W^{\mathcal{F}}(\mathcal{T},t)$.

3 The δ-Function

In order to avoid explicitly deriving the EDF schedule in the presence of faults, we define a function, δ, which loosely corresponds to the "extra" work induced by a certain fault pattern, $\mathcal{F}$.

$$\delta(\mathcal{T},t,\mathcal{F}) = W^{\mathcal{F}}(\mathcal{T},t) - W(\mathcal{T},t). \tag{1}$$

Intuitively, $\delta(\mathcal{T},t,\mathcal{F})$ is the amount of unfinished "extra" work that has been induced by the fault pattern $\mathcal{F}$ at time t. In other words, it is the work needed above and beyond what is required in the fault-free schedule for $\mathcal{T}$. The idle time in the fault-free EDF schedule is used to do this extra work.

The δ-function will play an important role in the process of checking if each task meets its deadline in $EDF^{\mathcal{F}}(\mathcal{T})$. Following is a method for computing δ directly from the fault-free EDF schedule of $\mathcal{T}$ and the fault pattern $\mathcal{F}$.

$$\delta(\mathcal{T},t,\mathcal{F}) = \begin{cases} 0 & t = 0 \\ \delta(\mathcal{T},t-1,\mathcal{F}) + V_j \cdot f_j & t = e_j \text{ for some } \tau_j \in \mathcal{T} \\ \delta(\mathcal{T},t-1,\mathcal{F}) \dot{-} 1 & EDF(\mathcal{T},t-1) = \uparrow \\ \delta(\mathcal{T},t-1,\mathcal{F}) & \text{otherwise.} \end{cases} \tag{2}$$

In order to show that the above form for $\delta(\mathcal{T},t,\mathcal{F})$ is equivalent to $W^{\mathcal{F}}(\mathcal{T},t) - W(\mathcal{T},t)$, we consider the four different cases above.

Case 1: At $t = 0$, no task can end and, thus, $t_0 \neq e_j$ for any j. From the definitions of W and $W^{\mathcal{F}}$, this implies that $W(\mathcal{T},0) = W^{\mathcal{F}}(\mathcal{T},0)$ and, thus, $\delta(\mathcal{T},0,\mathcal{F}) = 0$.

Case 2: When $t = e_j$ for some $\tau_j \in \mathcal{T}$, Property 4 implies that $W(\mathcal{T},t-1) > 0$, which by Property 3 implies that also $W^{\mathcal{F}}(\mathcal{T},t-1) > 0$. Hence,

$$\begin{aligned} \delta(\mathcal{T},t,\mathcal{F}) &= \left(W^{\mathcal{F}}(\mathcal{T},t-1) - 1 + \sum_{\tau_i \in RS(\mathcal{T},t)} C_i + f_j \cdot V_j \right) \\ &\quad - \left(W(\mathcal{T},t-1) - 1 + \sum_{\tau_i \in RS(\mathcal{T},t)} C_i \right) \\ &= \delta(\mathcal{T},t-1,\mathcal{F}) + f_j \cdot V_j. \end{aligned}$$

Case 3: When $EDF(\mathcal{T},t-1) = \uparrow$, Property 1 states that $W(\mathcal{T},t-1) = 0$ and Property 4 states that $t \neq e_j$ for all j. In this case,

$$\begin{aligned} \delta(\mathcal{T},t,\mathcal{F}) &= \left(W^{\mathcal{F}}(\mathcal{T},t-1) \dot{-} 1 + \sum_{\tau_i \in RS(\mathcal{T},t)} C_i \right) - \sum_{\tau_i \in RS(\mathcal{T},t)} C_i \\ &= W^{\mathcal{F}}(\mathcal{T},t-1) \dot{-} 1 \\ &= \delta(\mathcal{T},t-1,\mathcal{F}) \dot{-} 1. \end{aligned}$$

Case 4: When $t \neq e_i$ and $EDF(\mathcal{T},t-1) \neq \uparrow$, then Property 1 implies that $W(\mathcal{T},t-1) > 0$, which by Property 3 implies that also $W^{\mathcal{F}}(\mathcal{T},t-1) > 0$. Hence, the $\dot{-}$ operations in the definitions of W and $W^{\mathcal{F}}$ reduce to the usual subtraction and it is straightforward to show that $\delta(\mathcal{T},t,\mathcal{F}) = \delta(\mathcal{T},t-1,\mathcal{F})$.

For illustration, Fig. 1 shows an example of a task set and the corresponding values of the δ function for a specific $\mathcal{F}$. In this example, we consider the case in which only τ_1 and τ_3 may be subject to a fault. Note that the value of δ decreases when $EDF(\mathcal{T})$ is idle and increases at the end of each task that is indicated as faulty in $\mathcal{F}$.

As we have mentioned above, the δ function is an abstraction that represents the extra work to be performed for recovery. This extra work reduces to zero when all ready

tasks complete execution and recovery, as demonstrated by the following theorem.

Theorem 1. *If $\delta(\mathcal{T}, t-1, \mathcal{F}) > 0$ and $\delta(\mathcal{T}, t, \mathcal{F}) = 0$, then, in both $EDF(\mathcal{T})$ and $EDF^{\mathcal{F}}(\mathcal{T})$, any task with $R_i \leq t$ finishes at or before time t.*

Proof. If $\delta(\mathcal{T}, t-1, \mathcal{F}) > 0$ and $\delta(\mathcal{T}, t, \mathcal{F}) = 0$ then, from (2), this decrease in the value of the δ-function is only possible if $EDF(\mathcal{T}, t) = \uparrow$, which, from Property 1, leads to $W(\mathcal{T}, t) = 0$. Thus, (1) gives $W^{\mathcal{F}}(\mathcal{T}, t) = 0$ and the proof follows from Properties 1 and 2. □

4 Feasibility Test for a Task Set Under a Specific Fault Pattern

Given a task set, $\mathcal{T}$, and a fault pattern, $\mathcal{F}$, we now present a method for checking whether the lowest priority task, denoted by $\tau_\ell \in \mathcal{T}$, completes by its deadline in $EDF^{\mathcal{F}}(\mathcal{T})$.

Theorem 2. *Given a task set, $\mathcal{T}$, and a fault pattern, $\mathcal{F}$, the lowest priority task, τ_ℓ, in $\mathcal{T}$ completes by D_ℓ in $EDF^{\mathcal{F}}(\mathcal{T})$, if and only if $\delta(\mathcal{T}, t, \mathcal{F}) = 0$ for some t, $e_\ell \leq t \leq D_\ell$.*

Proof. To prove the *if* part, assume that t_0 is the smallest value such that $e_\ell \leq t_0 \leq D_\ell$ and $\delta(\mathcal{T}, t_0, \mathcal{F}) = 0$. If $\delta(\mathcal{T}, t, \mathcal{F}) = 0$ for every $t = 0, \ldots, t_0$, then $EDF^{\mathcal{F}}(\mathcal{T})$ and $EDF(\mathcal{T})$ are identical from $t = 0$ to $t = t_0$, which implies that τ_ℓ completes by $e_\ell \leq D_\ell$ in both schedules. If, however, $\delta(\mathcal{T}, t, \mathcal{F}) > 0$ for some $0 \leq t < t_0$, then let $\bar{t}$ be the latest time before t_0 such that $\delta(\mathcal{T}, \bar{t}, \mathcal{F}) > 0$. Note that $\bar{t} < e_\ell$ (since t_0 is the first value after e_ℓ at which $\delta = 0$) and that $\delta(\mathcal{T}, \bar{t}+1, \mathcal{F}) = 0$ (by the definition of $\bar{t}$). Hence, by Theorem 1, all tasks that are ready before $\bar{t}$ finish execution by $\bar{t}$ in both $EDF(\mathcal{T})$ and $EDF^{\mathcal{F}}(\mathcal{T})$. Moreover, $\delta(\mathcal{T}, t, \mathcal{F}) = 0$ for $t = \bar{t}+1, \ldots, t_0$, which means that $W(\mathcal{T}, t) = W^{\mathcal{F}}(\mathcal{T}, t)$ and, thus, $EDF(\mathcal{T})$ is identical to $EDF^{\mathcal{F}}(\mathcal{T})$ in that period. But, τ_ℓ completes in $EDF(\mathcal{T})$ at e_ℓ, which means that it also completes at e_ℓ in $EDF^{\mathcal{F}}(\mathcal{T})$.

We prove the *only if* part by contradiction: Assume that $\delta(\mathcal{T}, t, \mathcal{F}) > 0$ for all $e_\ell \leq t \leq D_\ell$ and yet τ_ℓ finishes in $EDF^{\mathcal{F}}(\mathcal{T})$ at $\bar{t}$ for some $e_\ell \leq \bar{t} \leq D_\ell$. The fact that the lowest priority task, τ_ℓ, executes between time $\bar{t}-1$ and $\bar{t}$ means that no other task is available for execution at $\bar{t}-1$ and, thus, $W^{\mathcal{F}}(\mathcal{T}, \bar{t}-1) = 1$. Given the assumption that $\delta(\mathcal{T}, \bar{t}-1, \mathcal{F}) > 0$, Property 3 implies that $W(\mathcal{T}, \bar{t}-1) = 0$, which, by Property 1, implies that $EDF(\mathcal{T}, \bar{t}-1) = \uparrow$ and (2) leads to $\delta(\mathcal{T}, \bar{t}, \mathcal{F}) = 0$, which is a contradiction. □

The next corollaries provide conditions for the feasibility of $EDF^{\mathcal{F}}(\mathcal{T})$ for the entire task set, $\mathcal{T}$.

Corollary 1. *A* necessary *and* sufficient *condition for the feasibility of $EDF^{\mathcal{F}}(\mathcal{T})$ for a given $\mathcal{T}$ and a given $\mathcal{F}$ can be obtained by applying Theorem 2 to the n task sets $\mathcal{T}_j$, $j = 1, \ldots, n$, where $\mathcal{T}_j$ contains the j highest priority tasks in $\mathcal{T}$.*

Proof. The proof is by induction. The base case is trivial, when $j = 1$, since there is only a single task. For the induction step, assume that $EDF^{\mathcal{F}}(\mathcal{T}_j)$ is feasible and consider $\mathcal{T}_{j+1} = \mathcal{T}_j \bigcup \{\tau_\ell\}$, where τ_ℓ has a lower priority than any task in $\mathcal{T}_j$. In $EDF^{\mathcal{F}}(\mathcal{T}_{j+1})$, all tasks in $\mathcal{T}_j$ will finish at exactly the same time as in $EDF^{\mathcal{F}}(\mathcal{T}_j)$ since τ_ℓ has the lowest priority. Hence, the necessary and sufficient condition for the feasibility of $EDF^{\mathcal{F}}(\mathcal{T}_{j+1})$ is equivalent to the necessary and sufficient condition for the completion of τ_ℓ by D_ℓ. □

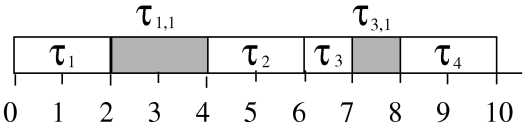

Fig. 2. The fault-tolerant schedule for the task set in Fig. 1.

Corollary 2. *A* sufficient *(but not* necessary*) condition for the feasibility of $EDF^{\mathcal{F}}(\mathcal{T})$ for a given $\mathcal{T}$ and a given $\mathcal{F}$ is*

$$\forall \tau_i \in \mathcal{T}, \ \exists\, e_i \leq t_i \leq D_i \ \text{ such that } \delta(\mathcal{T}, t_i, \mathcal{F}) = 0.$$

Proof. Note that the proof of the *only if* part in Theorem 2 relies on the property that τ_ℓ is the lowest priority task, which, in EDF, means the task with the latest deadline. The *if* part of the theorem, however, is true even if τ_ℓ is not the lowest priority task. Hence, any $\tau_i \in \mathcal{T}$ completes by D_i in $EDF^{\mathcal{F}}(\mathcal{T})$ if $\delta(\mathcal{T}, t, \mathcal{F}) = 0$ for some t, $e_i \leq t \leq D_i$, which proves the corollary. □

We clarify the conditions of the above corollaries by examples. First, we show that the condition given in Corollary 2 is not necessary for the feasibility of $EDF^{\mathcal{F}}(\mathcal{T})$. That is, we show that, for any given τ_i, it is not necessary that $\delta(\mathcal{T}, t_i, \mathcal{F}) = 0$ for some $e_i \leq t_i \leq D_i$, in order for τ_i to finish by D_i in $EDF^{\mathcal{F}}(\mathcal{T})$. This can be seen from the example task set and fault pattern shown in Fig. 1. The value of $\delta(\mathcal{T}, t, \mathcal{F})$ is not zero between $e_1 = 2$ and $D_1 = 4$ or between $e_2 = 5$ and $D_2 = 7$. Yet, as shown in Fig. 2, τ_1 and τ_2 will finish by their deadlines in $EDF^{\mathcal{F}}(\mathcal{T})$. In other words, the condition that $\delta(\mathcal{T}, t, \mathcal{F}) = 0$ between e_i and D_i, as stated in Theorem 2, is necessary and sufficient for the feasibility of only the lowest priority task in $EDF^{\mathcal{F}}(\mathcal{T})$ (task τ_4 in the above example).

Next, we show that, as stated in Corollary 1, we have to repeatedly apply Theorem 2 to all task sets $\mathcal{T}_j$, $j = 1, \ldots, n$, to obtain a sufficient condition for the feasibility of the entire task set. In other words, it is not sufficient to apply Theorem 2 only to $\mathcal{T}$. This can be demonstrated by modifying the example of Fig. 1 such that $D_3 = 7$. Clearly, this change in D_3 may still result in the same EDF schedule for $\mathcal{T}$ and thus will not change the calculation of $\delta(\mathcal{T}, t, \mathcal{F})$. Although the application of Theorem 2 guarantees that τ_4 will finish by its deadline in $EDF^{\mathcal{F}}(\mathcal{T})$, the recovery of τ_3 will not finish by $D_3 = 7$, as seen in Fig. 2.

Assume, without loss of generality, that the tasks in a given task set, $\mathcal{T}$, are numbered such that $e_1 \leq e_2 \leq \ldots \leq e_n$ and define $\delta_i(\mathcal{T}, \mathcal{F}) = \delta(\mathcal{T}, e_i, \mathcal{F})$ to be the extra work that still needs to be done due to a fault pattern $\mathcal{F}$, at time $t = e_i$. Noting that $\delta(\mathcal{T}, t, \mathcal{F})$ increases only at $t = e_1, \ldots, e_n$, (2) can be rewritten using the $slack()$ function defined in Section 2 as follows:

$$\delta(\mathcal{T}, t, \mathcal{F}) = \delta_i(\mathcal{T}, \mathcal{F}) \dot{-} slack(e_i, t) \quad e_i \leq t < e_{i+1}, \qquad (3)$$

where

$$\delta_i(\mathcal{T},\mathcal{F}) = \begin{cases} V_1 \cdot f_1 & i = 1 \\ \delta_{i-1}(\mathcal{T},\mathcal{F}) \doteq slack(e_{i-1}, e_i) + V_i \cdot f_i & i = 2, 3, \ldots \end{cases} \quad (4)$$

The application of Theorem 2 for a given $\mathcal{T}$ and $\mathcal{F}$ requires the simulation of $EDF(\mathcal{T})$ and the computation of e_i, $i = 1, \ldots, n$ as well as $slack(e_{i-1}, e_i)$. The values of δ_i computed from (3) and (4) can then be used to check the condition of the theorem. Each step in the above procedure takes $O(n)$ time, except for the simulation of the EDF schedule. Such simulation may be efficiently performed by using a heap which keeps the tasks sorted by deadlines. Each task is inserted into the heap when it is ready and removed from the heap when it completes execution. Since each insertion into and deletion from the heap takes $O(logn)$ time, the total simulation of EDF takes $O(nlogn)$ time. Thus, the time complexity of the entire procedure is $O(nlogn)$.

Hence, given a task set, $\mathcal{T}$, and a specific fault pattern, $\mathcal{F}$, a sufficient and necessary condition for the feasibility of $EDF^{\mathcal{F}}(\mathcal{T})$ can be computed using Corollary 1 in $O(n^2logn)$ time steps. This is less efficient than simulating $EDF^{\mathcal{F}}(\mathcal{T})$ directly, which can be done in $O(nlogn)$ steps. However, as will be described in the next section, simulating $EDF(\mathcal{T})$ only is extremely advantageous when we consider arbitrary fault patterns rather than a specific fault pattern.

5 Feasibility Test for a Task Set under Any Fault Pattern

We now turn our attention to determining the feasibility of a given task set for *any* fault pattern with k or less faults. We use $\mathcal{F}^w$ to denote a fault pattern with exactly w faults. That is,

$$\sum_{f_i \in \mathcal{F}^w} f_i = w.$$

We also define the function $\delta^w(\mathcal{T}, t)$, which represents the maximum extra work at time t induced by exactly w faults that occurred at or before time t. In other words, it is the extra work induced by the worst-case fault pattern of w faults:

$$\delta^w(\mathcal{T}, t) = \max_{\mathcal{F}^w}\{\delta(\mathcal{T}, t, \mathcal{F}^w)\}.$$

Note that, although the use of $\mathcal{F}^w$ in the above definition does not specify that all w faults will occur at or before time t, the value of $\delta(\mathcal{T}, t, \mathcal{F}^w)$ will reach its maximum when all possible w faults occur by time t.

Theorem 3. *For a given task set, $\mathcal{T}$, a given number of faults w, and any fault pattern, $\mathcal{F}^w$, the lowest priority task, τ_ℓ, in $\mathcal{T}$ completes by D_ℓ in $EDF^{\mathcal{F}^w}(\mathcal{T})$, if and only if $\delta^w(\mathcal{T}, t) = 0$ for some t, $e_\ell \le t \le D_\ell$.*

Proof. This theorem is an extension of Theorem 2 and can be proven in a similar manner. □

In order to compute $\delta^w(\mathcal{T}, t)$ efficiently, we define the values

$$\delta_i^w(\mathcal{T}) = \max_{\mathcal{F}^w}\{\delta_i(\mathcal{T}, \mathcal{F}^w)\} \quad i = 1, 2, \ldots$$

and use them to compute

$$\delta^w(\mathcal{T}, t) = \delta_i^w(\mathcal{T}) \doteq slack(e_i, t) \quad e_i \le t < e_{i+1}, \quad (5)$$

which is directly derived from (3).

The value of each $\delta_i^w(\mathcal{T})$ is defined as the maximum extra work at $t = e_i$ induced by any fault pattern with w faults. This maximum value can be obtained by considering the worst scenario in each of the following two cases:

- All w faults have already occurred in $\tau_1, \ldots, \tau_{i-1}$. Hence, the maximum extra work at e_i is the maximum extra work at e_{i-1} decremented by the slack available between e_{i-1} and e_i.
- $w - 1$ faults have already occurred in $\tau_1, \ldots, \tau_i$ and one additional fault occurs in τ_i. In this case, the maximum extra work at e_i is increased by V_i, the recovery time of τ_i.

Hence, noting that $e_1, \ldots, e_n$ and the function $slack()$ are derived from $EDF(\mathcal{T})$ and do not depend on any particular fault pattern, the values of $\delta_i^w(\mathcal{T})$ can be computed for $i = 1, \ldots, n$ and $w = 1, \ldots, k$ using the following recursive formula:

$$\delta_i^w(\mathcal{T}) = \begin{cases} 0 & w = 0 \\ w\, V_1 & i = 1 \\ max\{\delta_{i-1}^w(\mathcal{T}) \doteq slack(e_{i-1}, e_i)\ , \delta_i^{w-1}(\mathcal{T}) + V_i\} & \text{otherwise.} \end{cases} \quad (6)$$

The computations in (6) can be graphically represented using a graph, G, with n columns and k rows, where each row corresponds to a particular number of faults, w, and each column corresponds to a particular e_i (see Fig. 3b). The node corresponding to row w and column e_i will be denoted by N_i^w. A vertical edge between N_i^w and N_i^{w+1} represents the execution of one recovery block of task τ_i and thus is labeled by V_i. A horizontal edge between N_{i-1}^w and N_i^w means that no faults occur in task τ_i, and thus is labeled by $\doteq slack(e_{i-1}, e_i)$ to indicate that the extra work that remained at e_{i-1} is decremented by the slack available between e_{i-1} and e_i. Then, each path starting at N_1^0 in G represents a particular fault pattern (see Fig. 4). The value of δ_i^w corresponding to the worst case pattern of w faults at $t = e_i$ is computed from (6), which corresponds to a dynamic programming algorithm to compute the longest path from N_1^0 to N_i^w.

Fig. 3 depicts an example of the computation of δ_i^w for a specific task set and $w = 0, 1, 2$. The value of δ_i^w is written inside node N_i^w. We can see that, for this example, $\delta_4^2(\mathcal{T}) = 3$ and, thus, from (5), $\delta^2(\mathcal{T}, 10) = 0$, which satisfies the condition of Theorem 3, and thus, the lowest priority task, τ_3, will finish before $D_3 = 10$ in the presence of up to any two faults.

Similar to Corollary 1 discussed in the last section, a necessary and sufficient condition for the feasibility of $EDF^{\mathcal{F}}(\mathcal{T})$ requires the repeated application of Theorem 3.

Corollary 3. *A* necessary *and* sufficient *condition for the feasibility of $EDF^{\mathcal{F}}(\mathcal{T})$ for a given $\mathcal{T}$ and any fault pattern $\mathcal{F}$ with k or less faults can be obtained by applying Theorem 3 to the n task sets $\mathcal{T}_j$, $j = 1, \ldots, n$, where $\mathcal{T}_j$ contains the j highest priority tasks in $\mathcal{T}$.*

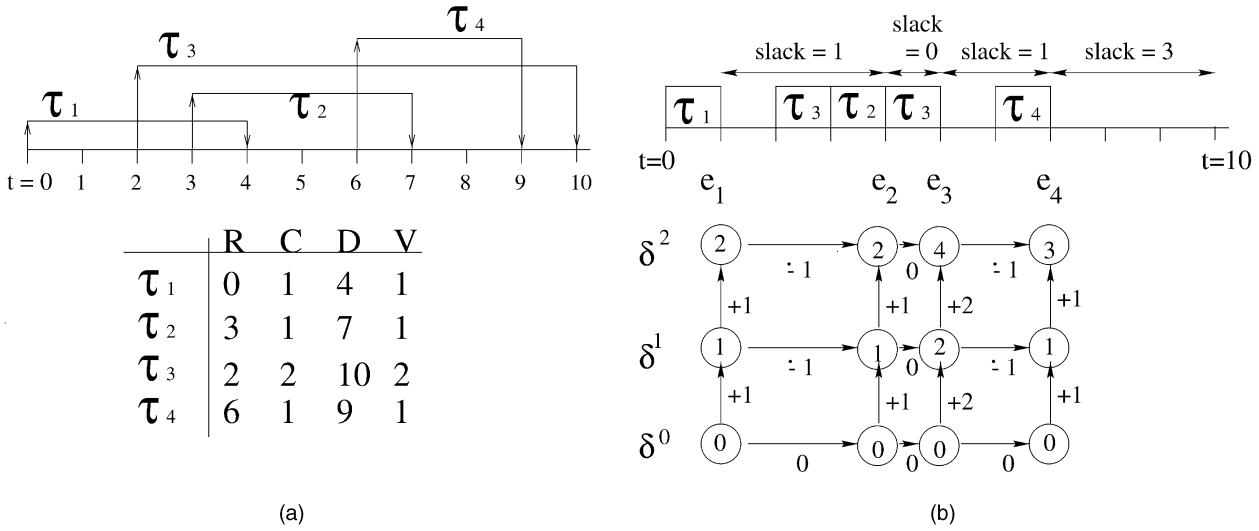

Fig. 3. The calculation of δ_i^1 and δ_i^2 for $i = 1, 2, 3, 4$. (a) The task set. (b) The fault-free schedule and the computation of δ.

Fig. 5 shows the computation of δ for an example with three tasks. Note that, although the application of Theorem 3 to this example shows that the lowest priority task, τ_3, will finish by its deadline in the presence of any two faults, the set of three tasks is not feasible in the presence of two faults in τ_1 since, in this case, either τ_1 or τ_2 will miss the deadline. This is detected when Theorem 3 is applied to the task set $\mathcal{T}_2 = \{\tau_1, \tau_2\}$.

To summarize, given a task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ and the maximum number of faults, k, the following algorithm can be used to optimally check if $EDF^{\mathcal{F}}(\mathcal{T})$ is feasible for any fault pattern of at most k faults.

Algorithm "Exact"

- Let $\mathcal{T}_1 = \{\tau_1\}$, where τ_1 is the highest priority task in $\mathcal{T}$ /* the one with earliest deadline */,
- Let $\ell = 1$ /* τ_ℓ is the lowest priority task (the only task) in $\mathcal{T}_1$ */
- For $j = 1, \ldots, n$ do
  1. Simulate $EDF(\mathcal{T}_j)$ and compute $e_1, \ldots, e_j$ as well as $slack()$,
  2. Renumber the tasks in $\mathcal{T}_j$ such that $e_1 \leq \ldots \leq e_j$,
  3. Compute δ_i^w for $i = 1, \ldots, j$ and $w = 1, \ldots, k$ from (6),
  4. Let $e_{j+1} = D_\ell$ /* this is just for computational convenience */
  5. If $\delta_i^w \dot{-} slack(e_i, e_{i+1}) > 0$ for all $i = \ell, \ldots, j$, then $EDF^{\mathcal{F}}(\mathcal{T})$ is not feasible ; EXIT.
  6. If $(j = n)$, then $EDF^{\mathcal{F}}(\mathcal{T})$ is feasible ; EXIT.
  7. Let τ_ℓ be the highest priority task in $\mathcal{T} - \mathcal{T}_j$,
  8. $\mathcal{T}_{j+1} = \mathcal{T}_j \bigcup \{\tau_\ell\}$, /* note that τ_ℓ is the lowest priority task in $\mathcal{T}_{j+1}$ */

Hence, in order to determine if the lowest priority task in a task set can finish by its deadline in the presence of at most k faults, Steps 1-5 (which apply Theorem 3), requires $O(nlogn + nk)$ steps to both generate $EDF(\mathcal{T}_j)$ and apply (6). In order to determine the feasibility of $EDF^{\mathcal{F}}(\mathcal{T})$, we repeat the `for` loop n times for $\mathcal{T}_1, \ldots, \mathcal{T}_n$ for a complexity of $O(n^2 logn + n^2 k)$. Note, however, that with some care, $EDF(\mathcal{T}_{j+1})$ can be derived from $EDF(\mathcal{T}_j)$ in at most $O(n)$ steps, thus resulting in a total of $O(n^2 k)$ for the feasibility test. Compared with the $O(n^{k+1} logn)$ complexity required to simulate EDF under the possible $O(n^k)$ fault patterns, our

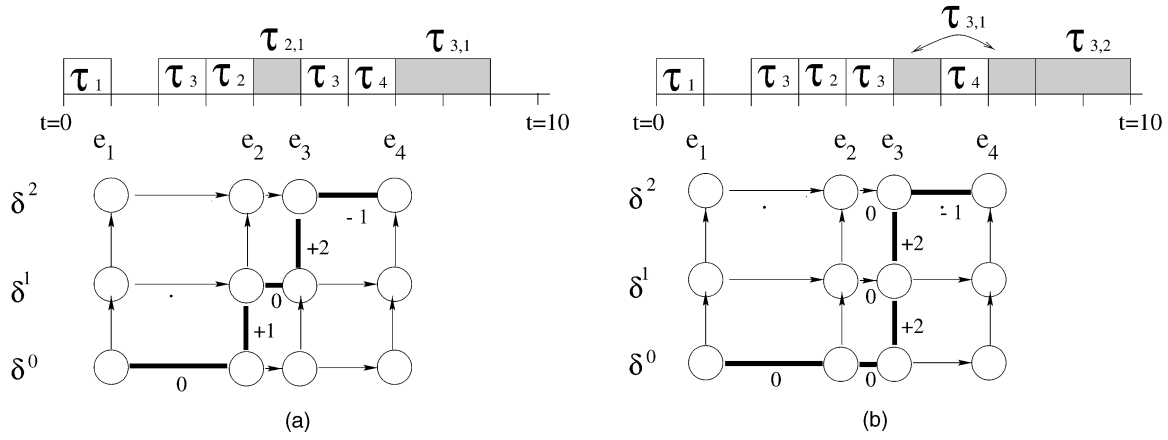

Fig. 4. Two fault patterns for the task set of Fig. 3 and the corresponding paths in G. (a) $f_1 = f_4 = 0$ and $f_2 = f_3 = 1$. (b) $f_1 = f_2 = f_4 = 0$ and $f_3 = 2$.

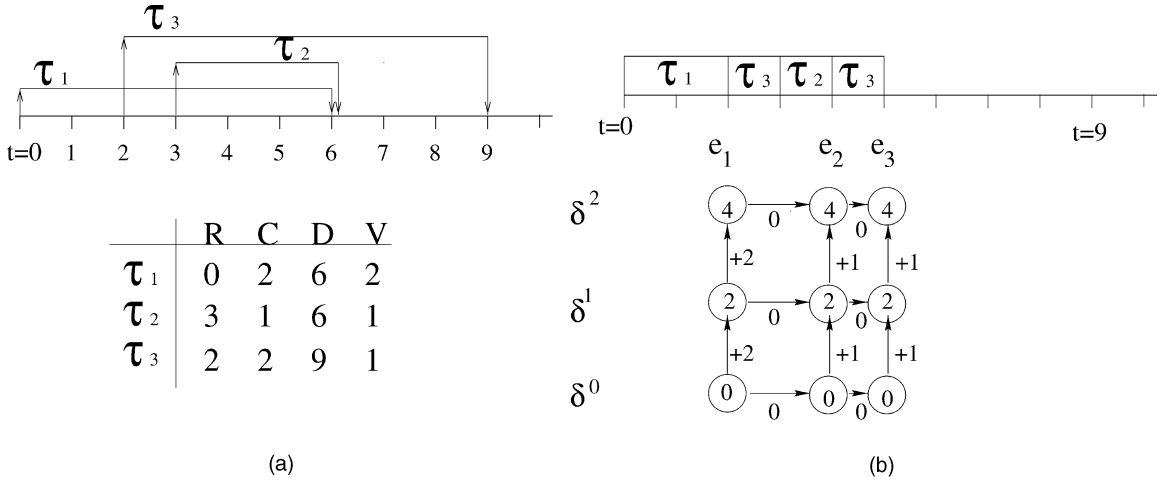

Fig. 5. An example with three tasks. (a) The task set. (b) The fault-free schedule and the computation of δ.

algorithm has a smaller time complexity, even for $k = 1$ (a single fault).

As indicated in Corollary 2, a sufficient but not necessary feasibility test may be obtained by computing δ from a simulation of $EDF(\mathcal{T})$ and then making sure that, for each task τ_i, δ is equal to zero between e_i and D_i. This can be completed in $O(nlog(n) + nk)$ time, as shown in the following algorithm.

Algorithm "Sufficient"

1. Simulate $EDF(\mathcal{T})$ and compute $e_1, \ldots, e_n$ as well as $slack()$,
2. Renumber the tasks in $\mathcal{T}_j$ such that $e_1 \leq \ldots \leq e_n$,
3. Compute δ_i^w for $i = 1, \ldots, n$ and $w = 1, \ldots, k$ from (6),
4. Let $e_{n+1} = D_n$ /* this is just for computational convenience */
5. For $i = 1, \ldots, n$ do
   If $\delta_i^w \doteq slack(e_i, e_{i+1}) > 0$, then declare $EDF^{\mathcal{F}}(\mathcal{T})$ not feasible; EXIT.
6. $EDF^{\mathcal{F}}(\mathcal{T})$ is feasible.

The example shown in Fig. 6 shows that a task τ_i, $i < n$, may complete by D_i in $EDF^{\mathcal{F}}(\mathcal{T})$ even if the value of δ computed from the simulation of $EDF(\mathcal{T})$ does not equal zero between e_i and D_i. In this example, $\delta_2^2 = 4$ and, thus, $\delta^2(\{\tau_1, \tau_2\}, t) \neq 0$ for $e_2 \leq t \leq D_2$. Yet, it is easy to see that the shown EDF schedule can tolerate any two faults (two faults in τ_1, two faults in τ_2, or one fault in each of τ_1 and τ_2). To intuitively explain this result, we note that, although δ_2^2 represents the maximum recovery work that needs to be done at $t = e_2$, no information is kept about the priority at which this recovery work will execute in $EDF^{\mathcal{F}}(\mathcal{T})$. Specifically, in the given example, some of the work in δ_2^2 will execute in $EDF^{\mathcal{F}}(\mathcal{T})$ at the priority of τ_1, which is lower than the priority of τ_2. Thus, it is not necessary that $\delta_2^2 = 0$ before D_2 for τ_2 to finish before its deadline. This, in general, may happen only because it is possible for a lower priority task to finish before a higher priority task. That is, if for some i and j, $e_i < e_j$ while $D_j < D_i$.

Finally, we note that, from the observation given in the last paragraph, algorithm "Sufficient" will provide a sufficient *and necessary* feasibility test in the special case where tasks complete execution in $EDF(\mathcal{T})$ in the order of their priorities (deadlines). That is, if $e_1, \ldots, e_n$ computed from $EDF(\mathcal{T})$ satisfy $e_i \leq e_{i+1}$ and $D_i \leq D_{i+1}$. In this case, the recovery work in any δ_i^w would have to execute in $EDF^{\mathcal{F}}(\mathcal{T})$ at a priority higher than or equal to that of τ_i and,

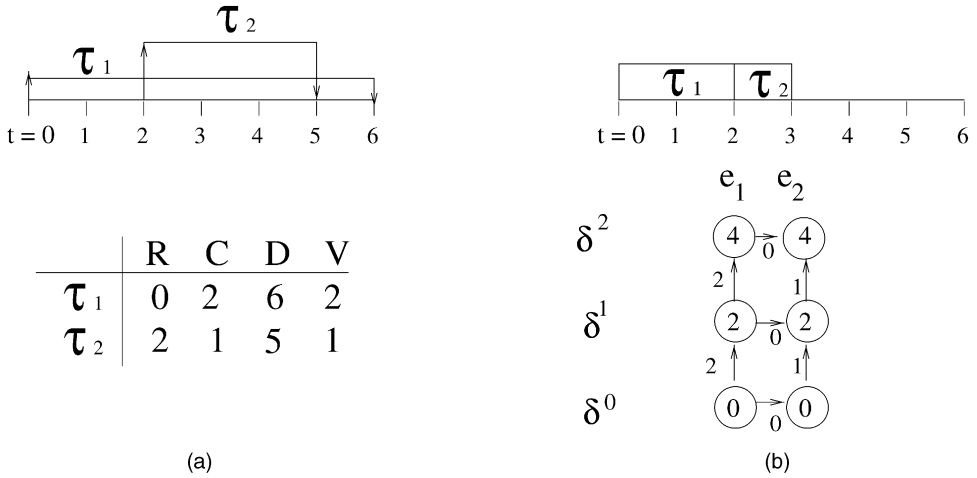

Fig. 6. An example in which $\{\tau_1, \tau_2\}$ can tolerate any two faults. (a) The task set. (b) The computation of δ.

thus, it is necessary for this work to be completed by D_i if τ_i is to complete by its deadline.

6 Related Work

Earlier work dealing with tolerance to transient faults for aperiodic tasks was carried out from the perspective of a single fault in the system [16], [14]. More recently, the fault models were enhanced to encompass a single fault occurring every interval of time, for both uniprocessors and multiprocessor systems [1], [6], [7]. Further, tolerance to transient faults for periodic tasks has also been addressed for uniprocessors [26], [27], [22], [24], [8] and multiprocessor systems [2], [23], [19].

In [14], processor failures are handled by maintaining contingency or backup schedules. These schedules are used in the event of a processor failure. To generate the backup schedule, it is assumed that an optimal schedule exists and the schedule is enhanced with the addition of "ghost" tasks, which function primarily as standby tasks. Since not all schedules will permit such additions, the scheme is optimistic. More details can be found in [15].

Duplication of resources has been used for fault tolerance in real-time systems [21]. However, the algorithm presented is restricted to the case where all tasks have the same period. Moreover, adding duplication for error recovery doubles the amount of resources necessary for scheduling.

In [1], a best effort approach to provide fault tolerance has been discussed in hard real-time distributed systems. A primary/backup scheme is used in which both the primary and the backup start execution simultaneously and if a fault affects the primary, the results of the backup are used. The scheme also tries to balance the workload on each processor.

More recently, work has been done on the problem of dynamic dispatching algorithms of frame-based computations with dynamic priorities when one considers a single fault. In [18], it was shown that simply generating n EDF schedules, one for each possible task failure, is sufficient to determine if a task set can be scheduled within their deadlines. Also, the work in [13] describes the approach taken by the Mars system in frame-based fault tolerance. Mars was a pioneer system in the timeline dispatching of tasks through the development of time-triggered protocols. It takes into account the scheduling overhead, as well as the need for explicit fault tolerance in embedded real-time systems. However, MARS requires special hardware to perform fault-tolerance related tasks such as voting and, thus, it cannot be used in a broad range of real-time systems.

7 Conclusion

We have addressed the problem of guaranteeing the timely recovery from multiple faults for aperiodic tasks. In our work, we assumed earliest-deadline-first scheduling for aperiodic preemptive tasks and we developed a necessary and sufficient feasibility-test for fault-tolerant admission control. Our test uses a dynamic programming technique to explore all possible fault patterns in the system, but has a complexity of $O(n^2 \cdot k)$, where n is the number of tasks to be scheduled and k is the maximum number of faults to be tolerated.

EDF is an optimal scheduling policy for any task set $\mathcal{T}$ in the sense that, if any task misses its deadline in $EDF(\mathcal{T})$, there is no schedule for $\mathcal{T}$ in which no deadlines are missed. EDF is also an optimal fault-tolerant scheduling policy. Specifically, $EDF^{\mathcal{F}}(\mathcal{T})$ for a fault pattern $\mathcal{F}$ is equivalent to $EDF(\mathcal{T}')$ where $\mathcal{T}'$ is obtained from $\mathcal{T}$ by replacing the computation time, C_i, of each task τ_i in $\mathcal{T}$ by $C_i + f_i \cdot V_i$. Hence, the work presented in this paper answers the following question optimally: **Given a task set, $\mathcal{T}$, is there a feasible schedule for $\mathcal{T}$ that will allow for the timely recovery from any combination of k faults?**

Acknowledgments

The authors would like to thank Sanjoy Baruah for proposing the problem of tolerating k faults in EDF schedules and for valuable discussions and feedback during the course of this work. The authors would also like to acknowledge the support of DARPA through contract DABT63-96-C-0044 to the University of Pittsburgh.

References

[1] S. Balaji, L. Jenkins, L.M. Patnaik, and P.S. Goel, "Workload Redistribution for Fault Tolerance in a Hard Real-Time Distributed Computing System," *Proc. IEEE Fault-Tolerant Computing Symp. (FTCS-19),* pp. 366-373, 1989.

[2] A.A. Bertossi, L.V. Mancini, and F. Rossini, "Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 9, Sept. 1999.

[3] J.L. Crowley, P. Bobet, and M. Mesrabi, "Layered Control of a Binocular Camera Head," *Int'l J. Pattern Recognition and Artifical Intelligence,* vol. 7, no. 1, pp. 109-122, Feb. 1993.

[4] A. Campbell, P. McDonald, and K. Ray, "Single Event Upset Rates in Space," *IEEE Trans. Nuclear Science,* vol. 39, no. 6, pp. 1,828-1,835, Dec. 1992.

[5] X. Castillo, S.R. McConnel, and D.P. Siewiorek, "Derivation and Caliberation of a Transient Error Reliability Model," *IEEE Trans. Computers,* vol. 31, no. 7, pp. 658-671, July 1982.

[6] S. Ghosh, R. Melhem, and D. Mossé, "Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System," *Proc. Int'l Parallel Processing Symp.,* Apr. 1994.

[7] S. Ghosh, D. Mossé, and R. Melhem, "Implementation and Analysis of a Fault-Tolerant Scheduling Algorithm," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 3, pp. 272-284, Mar. 1997.

[8] S. Ghosh, R. Melhem, D. Mossé, and D. Sen Sarma, "Fault-Tolerant Rate-Monotonic Scheduling," *J. Real-Time Systems,* vol. 15, no. 2, Sept. 1998.

[9] K.-H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers,* vol. 33, pp. 518-528, 1984.

[10] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randel, "A Program Structure for Error Detection and Recovery," *Lecture Notes in Computer Science,* vol. 16, pp. 177-193, 1974.

[11] R.K. Iyer and D.J. Rossetti, "A Measurement-Based Model for Workload Dependence of CPU Errors," *IEEE Trans. Computers,* vol. 35, no. 6, pp. 511-519, June 1986.

[12] R.K. Iyer, D.J. Rossetti, and M.C. Hsueh, "Measurement and Modeling of Computer Reliability as Affected by System Activity," *ACM Trans. Computer Systems,* vol. 4, no. 3, pp. 214-237, Aug. 1986.

[13] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic, 1997.

[14] C.M. Krishna and K. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Trans. Computers,* vol. 35, no. 5, pp. 448-455, May 1986.

[15] C.M. Krishna and K. Shin, *Real-Time Systems.* McGraw-Hill, 1997.

[16] A.L. Liestman and R.H. Campbell, "A Fault-Tolerant Scheduling Problem," *IEEE Trans Software Eng.*, vol. 12, no. 11, pp. 1,089-1,095, Nov. 1988.
[17] J.H. Lala and R.E. Harper, "Architectural Principles for Safety-Critical Real-Time Applications," *Proc. IEEE,* vol. 82, no. 1, pp. 25-40, Jan. 1994.
[18] F. Liberato, S. Lauzac, R. Melhem, and D. Mossé, "Global Fault Tolerant Real-Time Scheduling on Multiprocessors," *Proc. Euromicro Conf. Real-Time Systems,* 1999.
[19] S. Lauzac, R. Melhem, and D. Mossé, "An Efficient RMS Admission Control and Its Application to Multiprocessor Scheduling," *Proc. Int'l Parallel Processing Symp.,* pp. 511-518, 1998.
[20] L.D. Nguyen and A.M.K. Cheng, "An Imprecise Real-Time Image Magnification Algorithm," *Proc. Int'l Conf. Multimedia Systems,* 1996.
[21] Y. Oh and S. Son, "An Algorithm for Real-Time Fault-Tolerant Scheduling in a Multiprocessor System," *Proc. Fourth Euromicro Workshop Real-Time Systems,* June 1992.
[22] Y. Oh and S.H. Son, "Enhancing Fault-Tolerance in Rate-Monotonic Scheduling," *J. Real-Time Systems,* vol. 7, no. 3, pp. 315-329, Nov. 1994.
[23] Y. Oh and S. Son, "Allocating Fixed-Priority Periodic Tasks on Multiprocessor Systems," *Real-Time Systems J.,* vol. 9, pp. 207-239, 1995.
[24] M. Pandya and M. Malek, "Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks," *IEEE Trans. Computers,* vol. 47, no. 10, pp. 1,102-1,112, 1998.
[25] D.K. Pradhan, *Fault Tolerant Computing: Theory and Techniques.* Englewood Cliffs, N.J.: Prentice-Hall, 1986.
[26] S. Ramos-Thuel, "Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy," PhD thesis, Carnegie Mellon Univ., May 1993.
[27] S. Ramos-Thuel and J.K. Strosnider, "Scheduling Fault Recovery Operations for Time-Critical Applications," *Proc. Fourth IFIP Conf. Dependable Computing for Critical Applications,* Jan. 1994.
[28] Y.M. Yeh and T.Y. Feng, "Algorithm Based Fault Tolerance for Matrix Inversion with Maximum Pivoting," *J. Parallel and Distributed Computing,* vol. 14, pp. 373-389, 1992.

Frank Liberato: no biography or photo available.

Rami Melhem received a BE in electrical engineering from Cairo University in 1976, an MA degree in mathematics and an MS degree in computer science from the University of Pittsburgh in 1981, and a PhD degree in computer science from the University of Pittsburgh in 1983. He was an assistant professor at Purdue University prior to joining the faculty of The University of Pittsburgh in 1986, where he is currently a professor of computer science and electrical engineering and the chair of the Computer Science Department. His research interests include fault-tolerant and real-time systems, optical interconnection networks, high performance computing, and parallel computer architectures. Dr. Melhem served on program committees of numerous conferences and workshops on parallel, distributed, and fault-tolerant systems. He served as the general chair for the third International Conference on Massively Parallel Processing Using Optical Interconnections. He was on the editorial board of the *IEEE Transactions on Computers* and served on the advisory boards of the IEEE technical committees on parallel processing and computer architectures. He is the editor for the Plenum book series on computer science and is on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems.* Dr. Melhem is a fellow of the IEEE and a member of the ACM.

Daniel Mossé received a BS in mathematics from the University of Brasilia in 1986, and MS and PhD degrees in computer science from the University of Maryland in 1990 and 1993, respectively. He joined the faculty of The University of Pittsburgh in 1992, where he is currently an associate professor. His research interests include fault-tolerant and real-time systems, as well as networking. Dr. Mossé has served on program committees for all major IEEE-sponsored real-time related conferences and as and program chair for RTAS and RT Education Workshop. He is currently writing a book on real-time systems. Dr. Mossé is a member of the IEEE Computer Society and of the ACM.