Exact Fault-Sensitive Feasibility Analysis of Real-Time Tasks

Hakan Aydin, *Member*, *IEEE*

Abstract—In this paper, we consider the problem of checking the feasibility of a set of n real-time tasks while provisioning for timely recovery from (at most) k transient faults. We extend the well-known processor demand approach to take into account the extra overhead that may be induced by potential recovery operations under Earliest-Deadline-First scheduling. We develop a necessary and sufficient test using a dynamic programming technique. An improvement upon the previous solutions is to address and efficiently solve the case where the recovery blocks associated with a given task do not necessarily have the same execution time. We also provide an online version of the algorithm that does not require a priori knowledge of release times. The online algorithm runs in $O(m \cdot k^2)$ time, where m is the number of ready tasks. We extend the framework to *periodic* execution settings: We derive a sufficient condition that can be checked efficiently for the feasibility of periodic tasks in the presence of faults. Finally, we analyze the case where the recovery blocks are to be executed nonpreemptively and we formally show that the problem becomes intractable under that assumption.

Index Terms—Real-time scheduling, real-time systems, fault tolerance, deadline-driven systems, recovery blocks, processor demand analysis.

1 Introduction

In real-time systems, timeliness is as important as the correctness of the output. Thus, traditionally, hard real-time scheduling theory has aimed at achieving predictability by assuming worst-case scenarios such as the worst-case execution time and minimum interarrival rate [3], [27].

Providing reliability and availability in safety-critical applications is of paramount importance [30]. Further, in safety-critical *real-time systems*, the fault-tolerant techniques must take into account the timing constraints of the task set: Faults must be detected and appropriate recovery operations must be completed before the deadlines.

Permanent faults are usually tolerated by providing hardware redundancy in the form of hot-standby spares and/or by employing techniques such as triple modular redundancy [30]. *Transient* faults are typically short-lived; they can be caused by a variety of sources such as atmospheric nuclear particles (alpha-particles, protons, and neutrons) or electrical noise (power supply noise or electromagnetic interference). It is known that transient faults occur much more frequently than permanent faults [20], [21]. A recent study indicates that the emerging *low-power* design techniques [1] further increase the susceptibility of VLSI circuits to transient faults [28].

In real-time systems, transient faults are usually addressed through *time redundancy* and *backward error recovery* techniques in which extra CPU time is reserved in the schedule for potential recovery operations. To achieve this aim, in this paper, we adopt the *recovery block* approach [32] as the basis of error recovery mechanism. Fault detection tests (in the form of sanity or consistency checks [30]) are performed when a task completes. Should an error be detected, a recovery operation is undertaken by activating a recovery block. In the general case, there may be a need to execute *multiple* recovery blocks (or *secondaries*) one after another before a complete recovery is achieved or a *safe state* is loaded into the memory.

An alternative approach to recovery block execution is *checkpointing*, where important state information is saved periodically during task execution while error checking routines are run simultaneously [4], [26]. If an error is detected, the system state is rolled back to the last checkpoint and the computation is repeated. Though it may reduce the amount of recovery overhead compared to the recovery block approach, checkpointing has two drawbacks: increased runtime overhead due to the frequent checkpointing during the *fault-free* execution and inability to cope with cases where the error stems from the (unique) software implementation of the task. With recovery blocks, the designer can provide *alternate* implementations of the same task in the form of different recovery blocks and these can be activated if the error persists.

1.1 Contributions of This Paper

Our objective in this paper is to solve the following fundamental problem: **Given a set of n real-time tasks, is it possible to complete all tasks and potential recovery operations within timing constraints under any fault scenario with at most k transient faults?** In our analysis, we use the Earliest-Deadline-First (EDF) [27] scheduling policy, which is known to be optimal for uniprocessor systems under various conditions [13], [27].

After introducing our system model and assumptions in Section 2, we focus on the concept of *fault patterns*. Since we

• *The author is with the Computer Science Department, George Mason University, 4400 University Drive-MSN 4A5, Fairfax, VA 22030. E-mail: aydin@cs.gmu.edu.*

Manuscript received 18 Aug. 2006; revised 2 Dec. 2006; accepted 11 Apr. 2007; published online 5 June 2007.
Recommended for acceptance by L. Welch.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0321-0806.
Digital Object Identifier no. 10.1109/TC.2007.70739.

do not make any assumptions about the distribution of faults to individual tasks, the fault patterns enable us to characterize *all* possible execution scenarios with at most k faults. We show that traditional feasibility analysis techniques can lead to a combinatorial explosion if we consider each fault pattern *separately* (Section 3).

In Section 4, we derive the main theoretical result that is used as a basis for other contributions of the paper. Namely, we show how the **processor demand analysis** [3], [18], [33] technique can be extended to settings with transient faults. Processor demand analysis proved to be very useful in the feasibility analysis since its fundamental principles were introduced by Baruah et al. in the early 1990s [2], [3]. We believe that existing approaches to the feasibility problem in the presence of faults, especially for the EDF policy, remain somewhat disconnected from the well-known methodologies of conventional (that is, non-fault-tolerant) analysis techniques. In this aspect, we hope that our extensions will fill an important gap and lay ground for further research. In the same section, we also prove that the preemptive EDF scheduling policy is *optimal* for executing real-time tasks and recovery blocks in the presence of transient faults.

We show how the computation of worst-case recovery overhead function for k transient faults affecting n tasks can be achieved in $O(n \cdot k^2)$ time by the dynamic programming technique in Section 5. Based on this result, we present the algorithm FT-feasibility, which can be used to check the feasibility of the task set under all k-fault scenarios (Section 6). We note that the problem of checking the tolerance of a real-time task set to k faults was first addressed in [23]. The time complexity of our algorithm is $O(n^2 \cdot k^2)$. Our solution, unlike [23], does *not* assume that all of the recovery blocks associated with a given task have the same execution time. We believe that relaxing this assumption represents fault-tolerant real-time applications that rely on the recovery block mechanism more realistically. For instance, the first recovery block may try to reexecute the task and, in case another fault is detected at the end of the reexecution, an *alternate* module/version performing an approximate computation can be activated (or a *safe state* can be loaded into the memory). Moreover, there can be multiple alternate versions of the same task implemented at different sophistication levels. Clearly, in this realistic example, the worst-case execution times of recovery blocks do not need to be the same. Of course, one can still apply the feasibility analysis in [23] by setting the execution time of all recovery blocks to the largest execution time among all recovery blocks. In this case, the test becomes only sufficient but not necessary. By taking into account the variations in the overhead of recovery operations, we propose a necessary and sufficient feasibility test in the presence of transient faults.

In our study, we establish the fundamentals of a fault-sensitive feasibility analysis assuming an aperiodic task model with known timing parameters. Then, we use these results to derive a number of major extensions and analysis techniques.

In Section 7, we develop a fast *online* fault-sensitive admission test *that does not require a priori knowledge of task release times*. In some dynamic systems, this information may simply be unavailable and there may be a need to dynamically decide if the admission of the new task will compromise the timely completion of the existing tasks (with possible fault scenarios in mind). This extension is also motivated by the fact that, even if some information about release times is available, possible *jitter* sources can cause deviations in the actual task release times and this may limit the accuracy of the feasibility test. Our online solution also has the desirable feature of updating recovery-related information dynamically as faults occur. If we have to provision for a maximum of k faults during the execution, then the detection of a single fault will decrement the number of faults that we need to tolerate after this point. Moreover, the worst-case recovery overhead of the remaining recovery blocks is efficiently updated to incorporate the knowledge of the most recent fault's occurrence.

Although we build our fundamental results using an *aperiodic* real-time task model, we also show how the framework can be extended to *periodic* tasks in Section 8. We believe this is particularly relevant for practical applications as a significant number of safety-critical real-time tasks (including those in digital control applications) are invoked periodically. For this problem, we provide a *sufficient* condition that guarantees whether a set of periodic tasks can still meet their deadlines. The condition can be checked in time $O(n^2 \cdot k^2)$, where n is the number of periodic tasks.

Finally, in Section 9, we analyze the settings where the recovery blocks are to be executed *nonpreemptively* and show that the problem is intractable under that assumption. After reviewing related work in Section 10, we conclude in Section 11.

2 System Model and Assumptions

2.1 Task Model

We consider a uniprocessor system with a set of real-time tasks $\tau = \{\tau_1, \ldots, \tau_n\}$. The ready time (release time), deadline, and worst-case execution time of task τ_i are denoted by r_i, d_i, and c_i, respectively. All tasks are assumed to be independent. First, we derive our main results for *aperiodic* real-time tasks; in Section 8, we show how the framework can be extended to the case of *periodic* tasks.

Aperiodic and periodic tasks form the most common workloads in real-time applications [33]. If the real-time workload has both aperiodic and periodic tasks with hard deadlines, then the illustrated analysis technique is still applicable, but by considering each *job* of a periodic real-time task separately.

2.2 Fault Model

Our fault model assumes *transient faults* as they occur much more frequently than permanent faults [20], [21]. Further, we assume that each transient fault is of short duration [10], [31] and that it does not lead to a permanent fault. In general, it is not possible to tolerate permanent faults without using some form of *spatial redundancy*, for example, through the modular redundancy techniques [30].

The faults are detected at the end of the execution of each task through sanity or consistency checks [23], [30]. Upon the detection of a fault in task τ_i, the first recovery block associated with that task, namely, $B_{i,1}$, is activated. When

$B_{i,1}$ completes, the sanity/consistency checks are repeated on the output of $B_{i,1}$; if there is another fault, then the second recovery block, $B_{i,2}$, is initiated and so on. In other words, z faults affecting τ_i and its recovery blocks trigger $B_{i,1}, \ldots, B_{i,z}$. The result of the task/recovery block is *committed* only when the test indicates that it is consistent and/or within the allowed range. This can be achieved, for example, by substituting the task's output value to a function and checking if it satisfies a certain (easily verifiable) property [19] or by using checksums [30].

Following existing work on fault-tolerant real-time systems [23], [24], [34], we assume that a given transient fault affects only the task (or recovery block) that is executing on the CPU and not other tasks. This is a reasonable assumption since the transient faults that we consider are *short-lived* and the result of a task is not committed (hence, cannot propagate to other tasks) unless it passes a sanity/consistency check.

Throughout the paper, we consider scenarios in which at most k transient faults can occur during the execution of the aperiodic task set τ. We do not make any assumption about the *distribution* of faults: For instance, all $m \leq k$ faults can occur during the execution of τ_i and its recovery blocks, m distinct tasks can fail just once, and other combinations are equally possible. This allows consideration of a rather broad class of transient faults, including faults that arrive in *bursts* or systems where the mean time between consecutive faults can exhibit large variances because of environmental factors. A brief comparison with alternative models that assume knowledge about the probability distribution of fault arrivals is given in Section 10.

Since we assume that at most k faults can occur during the execution, there are k recovery blocks $B_{i,1}, \ldots, B_{i,k}$ associated with task τ_i. The worst-case execution time of recovery block $B_{i,j}$ is denoted by $b_{i,j}$. Note that the recovery blocks of a given task may have different execution times. If it is known in advance that the number of faults affecting a task τ_i and its recovery blocks will not exceed $z < k$, we can simply set $b_{i,z+1}, \ldots, b_{i,k}$ to 0 to reflect this fact. If not negligible, the time overhead of running the sanity/consistency checks can be incorporated into c_i and $b_{i,j}$ values.

Our extension to periodic task systems in Section 8 involves the analysis of fault scenarios with at most k faults affecting task instances (*jobs*) during each *hyperperiod*, which is defined as the least common multiple of task periods. This approach is called *hyperperiod-oriented fault tolerance* in [36].

2.3 Scheduling Model

We develop our framework by assuming preemptive EDF scheduling for tasks and recovery blocks. All recovery blocks associated with task τ_i will execute with deadline (hence, priority) d_i. Note that this points to the possibility of delaying and preempting the execution of recovery blocks: If a task τ_h arrives during the execution of τ_i's recovery block $B_{i,j}$ with $d_h < d_i$, then $B_{i,j}$ will be interrupted and resumed later. As we show later in the paper (Corollary 1), adopting a preemptive EDF policy for all of the tasks and recovery blocks turns out to be optimal from the scheduling point of view. The implications of executing the recovery blocks in a *nonpreemptive* manner are explored in Section 9.

3 Fault Patterns and Their Impact on Feasibility Analysis

Formally, a sequence of faults affecting tasks in τ and their recovery blocks are denoted by a *fault pattern* $f = \{f_1, \ldots, f_n\}$, where f_i denotes the number of faults affecting τ_i and its recovery blocks [23]. Further, we say that f is a *k-fault pattern (scenario)* if the total number of faults is exactly k, that is, if $\sum_{i=1}^{n} f_i = k$.

Note that each fault pattern will induce an extra workload (overhead) during recovery. Specifically, all of the recovery blocks associated with this fault pattern will need to be executed. We will use the notation x_f to denote the total execution time of the recovery blocks activated (or the **recovery overhead**) due the fault pattern f. That is,

$$x_f(\tau) = \sum_{i=1}^{n} \sum_{j=1}^{f_i} b_{i,j}.$$

Throughout our analysis, we will be interested in finding the *worst-case k-fault pattern W* that results in the largest recovery overhead for a (sub)set of tasks. The recovery overhead associated with the worst-case fault pattern W over a task set τ will be denoted by $w_k(\tau)$. Formally,

$$w_k(\tau) = \max_f \{x_f(\tau)\}.$$

Note that $w_k(\tau) \geq w_{k-1}(\tau)$ for $k \geq 1$ since all recovery blocks have nonnegative execution times. This implies that we can restrict our analysis to the *k-fault patterns* (where we have exactly k faults) when investigating the worst-case recovery overhead of fault scenarios with at most k faults.

Definition 1. *A task set τ is said to be k-*fault tolerant *(or, alternatively, τ can be scheduled in a k-*fault-tolerant *manner) if all tasks and potential recovery operations can be completed before their corresponding deadlines under* any *fault scenario with at most k transient faults.*

Note that this definition subsumes that of a *feasible* task set in conventional real-time scheduling theory: A task set is feasible if and only if it is zero-fault tolerant.

Well-established results of the hard real-time scheduling theory provide the means to analyze the feasibility under worst-case execution time and phasings. However, in that framework, each real-time task has a well-characterized worst-case workload which can be assessed through a worst-case execution time analysis. Hence, it is relatively easy to characterize the worst-case *total* workload before undertaking the feasibility analysis.

When the framework is extended to the k-fault settings, the designer faces the additional challenge of analyzing the feasibility for *all* of the k-fault scenarios, each of which may potentially create a distinct additional burden on timing constraints. In fact, the number of distinct k-fault patterns itself points to the considerable difficulty of the problem, even when one temporarily ignores the overhead of *checking* the feasibility of the schedule for each pattern.

Proposition 1. *The number of distinct k-fault patterns that can affect a task set with n tasks is $\binom{n+k-1}{k}$.*

TABLE 1
Example Task Set

id	r_i	d_i	c_i	$b_{i,1}$	$b_{i,2}$
τ_1	0	20	5	5	5
τ_2	10	40	3	1	3
τ_3	15	36	10	6	5
τ_4	25	50	10	10	5

The asymptotic behavior of binomial coefficients [12] implies that the number of distinct fault patterns grows rapidly with n; specifically, it is $\Omega((\frac{n}{k})^k)$.

Proposition 1 can be justified by observing that k indistinguishable balls can be distributed to n distinguishable bins in $\binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!}$ ways. As an example, if $n = 4$ and $k = 2$, there are $\binom{5}{2} = 10$ different two-fault patterns. The first four fault patterns correspond to the cases where both faults occur in τ_i, $i = 1, 2, 3, 4$. The last six fault patterns are given by the set {{1, 1, 0, 0}, {1, 0, 1, 0}, {1, 0, 0, 1}, {0, 1, 1, 0}, {0, 1, 0, 1}, {0, 0, 1, 1}}, which covers the cases where each task incurs at most one fault.

As a concrete example, consider the task set given in Table 1. When scheduled with EDF in the absence of any faults, all of the tasks complete well before their deadlines (Fig. 1). With some effort, the reader may check that the task set is 1-fault tolerant, that is, all tasks and one required recovery operation can be completed before the corresponding deadlines in case of any *single* fault. However, the picture changes significantly when we consider the tolerance to any combination of two faults: There are $\binom{5}{2} = 10$ different 2-fault scenarios that can occur during the execution of the task set. Clearly, a sound methodology is needed to address the combinatorial explosion to check the feasibility of a task set for *all* k-fault scenarios. Throughout the paper, the task set given in Table 1 will be used as a running example to illustrate the components of our solution framework.

4 Fault-Sensitive Processor Demand Analysis

Our approach to the feasibility analysis in the presence of faults will be based on extending the processor demand analysis technique [3], [7], [18]. First, we introduce some additional notation:

- $C(\tau)$ denotes the total execution time of the task set $\tau = \{\tau_1, \ldots, \tau_n\}$. Formally, $C(\tau) = \sum_{\tau_i \in \tau} c_i$.
- $\phi(t_1, t_2)$ denotes the subset of tasks that are released at or after t_1 and having deadlines less than or equal to t_2. Formally, $\phi(t_1, t_2) = \{\tau_i \in \tau | t_1 \leq r_i \leq d_i \leq t_2\}$.

A fundamental result in the feasibility analysis first obtained by Baruah et al. in the context of *sporadic* task sets [3] and later refined in many ways by other researchers [7], [17], [18], is the following:

Theorem 1 (from [3] and [18]). *A set of independent real-time tasks τ can be scheduled (by preemptive EDF) if and only if $C(\phi(t_1, t_2)) \leq t_2 - t_1$ for all intervals $[t_1, t_2]$.*

Note that the quantity $C(\phi(t_1, t_2))$ above represents the least upper bound on the amount of work that must be completed in interval $[t_1, t_2]$ and it is referred to as the **processor demand** of tasks in interval $[t_1, t_2]$ in [3], [7], [33].

Definition 2. *Given a set of real-time tasks τ and an interval of time $[t_1, t_2]$, the k-fault-sensitive processor demand of the task set in the interval is*

$$h_k(t_1, t_2) = C(\phi(t_1, t_2)) + w_k(\phi(t_1, t_2)).$$

Observe that the term $w_k(\phi(t_1, t_2))$ in the expression of fault-sensitive processor demand effectively represents the worst-case overhead of any k faults occurring exclusively in the subset of tasks $\phi(t_1, t_2)$. As a simple example, we revisit the task set given in Table 1. Consider the interval $[t_1 = 5, t_2 = 45]$. Applying the definitions above, we find that $\phi(t_1, t_2) = \{\tau_2, \tau_3\}$, and $C(\phi(t_1, t_2)) = 13$. Further, $w_2(\phi(t_1, t_2)) = 11$, which corresponds to the fault pattern $f = \{0, 0, 2, 0\}$, where τ_3 incurs faults twice (in the following section, we will illustrate how the function $w_k()$ can be computed efficiently for any task set).

A fundamental result of this research effort is given by the following theorem:

Theorem 2. *A set of real-time tasks τ can be scheduled in a k-fault-tolerant manner by EDF if and only if $h_k(t_1, t_2) \leq t_2 - t_1$ for every interval $[t_1, t_2]$.*

Proof. *Only if part*. Suppose the contrary, that is, there exists a task set τ that can be scheduled in a k-fault-tolerant manner and there is an interval $[t_1, t_2]$ such that $h_k(t_1, t_2) > t_2 - t_1$. Consider the worst-case k-fault pattern exclusively affecting tasks released at or after t_1 and with deadlines at or before t_2. The recovery overhead induced by this fault pattern is $w_k(\phi(t_1, t_2))$. Thus, $h_k(t_1, t_2) = C(\phi(t_1, t_2)) + w_k(\phi(t_1, t_2))$ represents the "augmented" workload that must be completed within that interval. If $h_k(t_1, t_2) > t_2 - t_1$, then *no scheduling algorithm* can meet all of the deadlines with this fault pattern since the total computational demand exceeds the available CPU time. The same holds for the EDF scheduling policy; hence, there is a contradiction.

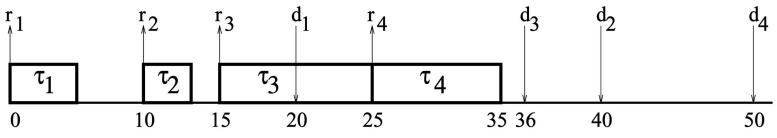

Fig. 1. Feasible EDF schedule without faults.

If part. Again, we proceed by contradiction. Suppose that there exists a task set τ such that $h_k(t_1, t_2) \leq t_2 - t_1$ for all intervals $[t_1, t_2]$, yet there is a k-fault pattern $(j \leq k)$ f^w resulting in deadline miss(es). Assume that the first deadline miss occurs at $t = d_i$. Now, let t_0 be the latest time preceding d_i such that

- the CPU is idle or
- a task (or recovery block) with deadline $> d_i$ executes.

Note that t_0 is well defined and it must correspond to a task release time. Further, during the interval $[t_0, d_i]$, the CPU is continuously busy executing only tasks (and recovery blocks) that are released at or after t_0 and with deadlines not exceeding d_i (namely, $\phi(t_0, d_i)$ and their recovery blocks induced by the fault pattern f^w).

Now, let $f^0 \subset f^w$ be the subset of faults affecting tasks in $\phi(t_0, d_i)$. Observe that the number of faults in f^0 cannot exceed k. Since EDF is a work-conserving scheduling algorithm that never leaves the CPU idle unless there are no ready tasks, the deadline miss at d_i and the definition of t_0 above imply that the available CPU time in the interval $[t_0, d_i]$ was not sufficient to accommodate the augmented processor demand $C(\phi(t_0, d_i)) + x_{f^0}(\phi(t_0, d_i))$ (the second term shows the extra recovery work induced by the fault pattern f^0).

Thus, we obtain, $d_i - t_0 < C(\phi(t_0, d_i)) + x_{f^0}(\phi(t_0, d_i))$. Noting that, for any task set β, the relation $w_k(\beta) \geq x_f(\beta)$ holds for all fault patterns with up to k faults, we get

$$d_i - t_0 < C(\phi(t_0, d_i)) + w_k(\phi(t_0, d_i)) = h_k(\phi(t_0, d_i)),$$

contradicting our assumption that $h_k(t_1, t_2) \leq t_2 - t_1$ for all intervals $[t_1, t_2]$. □

Recall that, in the "only if part," we established the necessary nature of the condition specified in the theorem for *any* scheduling algorithm that is able to recover from any k-fault scenario occurring in τ. Since, in the "if part," we established that the condition is also sufficient for EDF, we have the following:[1]

Corollary 1. *EDF is an optimal preemptive scheduling algorithm for meeting the timing constraints in the presence of transient faults.*

5 Computing the Worst-Case Recovery Overhead Function $w_k()$

In this section, we present an efficient technique to compute the worst-case recovery overhead for a given task set under any k-fault scenario. While developing our solution, we will temporarily ignore the timing constraints and show how to evaluate the function $w_k()$ for a set of n tasks that are simultaneously ready. However, the technique that we consider will be instrumental in later sections, where we develop the general fault-sensitive feasibility analysis framework.

1. We note that the optimality of EDF in the presence of faults was already given in [22]. However, we find it useful to derive it formally in light of fault-sensitive processor demand analysis.

Algorithm WC-Fault-Overhead

```
1 Compute w_j({τ_i}) for 1 ≤ j ≤ k and 1 ≤ i ≤ n using (2)
2 set β = {τ_1}
3 for m = 2 to n do
4   for h = 1 to k do
5     compute w_h(β ∪ {τ_m}) using (3)
6   endfor
7 set β = β ∪ {τ_m}
8 endfor
```

Fig. 2. Algorithm to compute the worst-case recovery overhead function.

Clearly, a straightforward way of tackling the problem would be to generate all different k-fault scenarios and compute the recovery overhead associated with each fault pattern. This brute-force technique is clearly infeasible from a computational point of view, as hinted by Proposition 1. Below, we outline the details of our solution that has a worst-case time complexity of $O(n \cdot k^2)$. Our solution will be based on a **dynamic programming** technique.

First, note that if we have a task set $\tau = \{\tau_x\}$ consisting of a single task τ_x, then the value of $w_j(\tau)$ can be trivially computed by observing that a fault pattern exclusively affecting τ_x will result in *consecutive* executions of recovery blocks $B_{x,j}$. That is,

$$w_0(\{\tau_x\}) = 0, \tag{1}$$

$$w_j(\{\tau_x\}) = w_{j-1}(\{\tau_x\}) + b_{x,j} \text{ for } 1 \leq j \leq k. \tag{2}$$

If the task set contains multiple tasks, then one can resort to a recursive formula. Suppose that we have evaluated the worst-case fault overhead functions $w_j(\tau)$ for $j = 1 \ldots k$ for a given task set τ. Consider the k-fault patterns affecting the *augmented* task set $\tau = \tau \cup \{\tau_x\}$. One possibility is that *all* k faults will occur in τ, whereas no fault occurs in τ_x. In this case, $w_k(\tau \cup \{\tau_x\})$ would simply be equal to $w_k(\tau)$. Alternatively, *exactly one* fault can affect τ_x (causing the recovery overhead of $b_{x,1} = w_1(\{\tau_x\})$) and the *remaining* $k-1$ faults can occur in τ, of which the worst-case recovery overhead is given by $w_{k-1}(\tau)$. Thus, in general, it is necessary and sufficient to consider k different cases (that correspond to the number of faults affecting τ_x): τ_x can incur j faults, whereas tasks in τ can collectively incur $k-j$ faults $(0 \leq j \leq k)$. Hence, the worst-case k-fault recovery overhead of $\tau \cup \{\tau_x\}$ can be evaluated by computing the maximum of these k cases. Formally,

$$w_k(\tau \cup \{\tau_x\}) = \max_{j=0}^{k} \{w_j(\{\tau_x\}) + w_{k-j}(\tau)\}. \tag{3}$$

Assuming that we have already computed $w_j(\tau)$ and $w_j(\{\tau_x\})$ for $j = 0, \ldots, k$, computing $w_k(\tau \cup \{\tau_x\})$ requires an additional $O(k)$ operations. The formula above immediately hints to an algorithm that computes the $w_j()$ values over an n-task set (see Fig. 2). The algorithm works in a bottom-up manner: It starts with the task set containing just one task. It augments the task set incrementally by adding one task at a time and by updating *all* of the $w_j()$ values $(j \leq k)$ along the way according to (3). It is important to note that we need to

compute *all* of the $w_j()$ values ($j < k$) in addition to $w_k()$ at every iteration: They are used to evaluate the worst-case recovery overhead functions in the *next* iteration. Finally, observe that the order in which the tasks are processed in this procedure does not affect the final outcome.

Complexity. First, we note that computing the worst-case overhead of fault patterns exclusively affecting single tasks requires $O(n \cdot k)$ time (line 1). The nested loop in lines 3-8 performs $O(\sum_{t=2}^{n} \sum_{h=1}^{k} h)$ operations, implying an overall time complexity of $O(n \cdot k^2)$. The space complexity of the algorithm is $O(n \cdot k)$.

Special cases. The analysis above addresses the most general case, that is, the values of the $w_k()$ function are computed for arbitrary recovery block execution times. However, we find it useful to further elaborate on two special cases that may be relevant for practical applications as the analysis is less complex.

5.1 Recovery Blocks with Identical Execution Times

Arguably, the simplest case corresponds to the one where the recovery blocks of a given task all have the same execution time, that is, $b_{i,j} = b_i \ \forall j$. This can happen if all the recovery blocks are functionally equivalent, for example, when they all consist of the reexecution of the main task. We note that this is the underlying model in [23].

Proposition 2. *If $b_{i,j} = b_i \ \forall j$, then $w_k(\tau) = k \cdot \max_{i=1}^{n}\{b_i\}$.*

Let τ_x be the the task with the largest recovery block execution time b_x (that is, $b_x \geq b_y \ \forall x, y$). Then, the scenario under which τ_x fails k times causes a recovery overhead of $k \cdot b_x$, which cannot be exceeded by the recovery overhead of any other fault scenario. Hence, Proposition 2 is easily justified.

Also, in this case, the problem of computing the worst-case recovery overhead function is reduced to finding the maximum among $b_1, \ldots, b_n$, which can be achieved in time $O(n)$.

5.2 Recovery Blocks with Nonincreasing Execution Times

As another interesting special case, we may have recovery blocks that satisfy the property $b_{i,j} \geq b_{i,j+1} \ \forall i, j$ for each task. This can be the case, for example, when the first recovery takes the form of reexecution of the task and successive recoveries (if needed) consist of implementing *increasingly simpler versions* of the original task, with decreasing execution times and decreasing output accuracies.

Since, in this case, the recovery blocks of a given task will already be ordered, then the algorithm to compute $w_k()$ may exploit this property. Let $\bar{b}_i(\tau)$ denote the ith *largest* value among *all* recovery block execution times $b_{1,1}, \ldots, b_{n,k}$.

Theorem 3. *If $b_{i,j} \geq b_{i,j+1} \ \forall i, j$, then $w_k(\tau) = \sum_{j=1}^{k} \bar{b}_j(\tau)$.*

The proof of Theorem 3, as well as an algorithm of time complexity $O(n \log n + k^2)$ to compute the worst-case recovery overhead function in this special case is presented in the Appendix.

6 Checking the Feasibility of the Task Set under Any k-Fault Scenario

Having extended the processor demand approach to incorporate the recovery overhead in Section 4 and having developed an algorithm to compute the worst-case recovery overhead function for a given task set in Section 5, we can address our fundamental problem: **Given a task set τ, can all tasks and recovery operations be completed within timing constraints under any k-fault scenario?**

At first, the application of Theorem 2 does not look promising from a computational point of view, because the number of intervals to be examined for a fault-sensitive processor demand can be fairly large. However, in real-time scheduling theory, it is well known that, when checking feasibility by the (non-fault-sensitive) processor demand approach, it is sufficient to check the intervals that begin at a task release time and end at a task deadline. In fact, the same holds for the fault-sensitive processor demand.

Proposition 3. *The fault-sensitive feasibility analysis of a real-time task set τ can be carried out exclusively on intervals $[t_1, t_2]$ such that t_1 is a task release time and t_2 is a task deadline, without compromising correctness.*

Proof. Suppose that the task set is not k-fault tolerant, that is, $h_k(t_1, t_2) > t_2 - t_1$ for an interval $[t_1, t_2]$. First, since $h_k(t_1, t_2)$ is always nonnegative and $t_2 > t_1$, it follows that there must be at least one task τ_i such that $t_1 \leq r_i \leq d_i \leq t_2$.

If t_2 is not a task deadline, then consider $h_k(t_1, d_j)$ such that d_j is the *latest* task deadline that does not exceed t_2. Clearly, $\phi(t_1, d_j) = \phi(t_1, t_2)$. Thus, $h_k(t_1, d_j) = h_k(t_1, t_2) > t_2 - t_1 > d_j - t_1$, implying that we would definitely be able to find another "overloaded" interval that ends at a task deadline. Thus, we can safely restrict our analysis to intervals $\{[t_1, t_2]\}$ such that t_2 is a task deadline. Similarly, if t_1 is not a task release time and $h_k(t_1, d_j) > d_j - t_1$ for an interval $[t_1, d_j]$, we can show that $h_k(r_y, d_j) = h_k(t_1, d_j) > d_j - t_1 > d_j - r_y$, where r_y is the *first* task release time after t_1. In other words, the interval $[r_y, d_j]$ would be "overloaded" as well. In conclusion, we can restrict our analysis to intervals that begin at a task release time and end at a task deadline.□

Proposition 3 implies that the number of intervals that need to be checked does not exceed n^2 for n tasks. However, a straightforward application of the algorithm WC-Fault-Overhead in Fig. 2 for each interval would result in an overall complexity of at least $O(n^3 \cdot k^2)$. On the other hand, a closer look reveals that the subsets of tasks that must be considered for each interval are not completely independent.

Proposition 4. *Suppose that r_a is a task release time and d_j, d_m are task deadlines such that $r_a \leq d_j < d_m$. If there are no other deadlines between d_j and d_m, then $\phi(r_a, d_m) = \phi(r_a, d_j) \cup \{\tau_i | d_i = d_m \text{ and } r_i \geq r_a\}$.*

Let us denote the set $\{\tau_i | d_i = d_m \text{ and } r_i \geq r_a\}$ by $\xi(r_a, d_m)$. Proposition 4 implies that, if one has already evaluated $h_k(r_a, d_j)$ and now is willing to evaluate $h_k(r_a, d_m)$, where d_m is the *first* deadline after d_j, then (s)he only has to

Algorithm FT-feasibility

```
1  Order the release times such that r_{a_1} ≤ r_{a_2} ... ≤ r_{a_n}
2  /* a_i is the index of the task with i^th earliest release time */
3  Order the deadlines such that d_{e_1} ≤ d_{e_2} ... ≤ d_{e_n}
4  /* e_i is the index of the task with i^th earliest deadline */
5  Status = feasible
6  /* Compute the recovery overhead of faults affecting single tasks */
7  for j = 1 to n do
8       set w_0(τ_j) = 0
9       for m = 1 to k do
10          set w_m({τ_j}) = w_{m-1}({τ_j}) + b_{j,m}
11      endfor
12 endfor
13 /* Compute the fault-sensitive processor demand for every interval [r_{a_i}, d_{e_j}] */
14 for i = 1 to n do
15   set C = 0
16   set γ = ∅
17   for j = 1 to n do
18     if r_{e_j} ≥ r_{a_i} then
19       /* τ_{e_j} falls in interval [r_{a_i}, d_{e_j}] */
20       set C = C + c_{e_j}
21       for h = 1 to k compute w_h(γ ∪ {τ_{e_j}}) using (3)
22       if C + w_k(γ ∪ {τ_{e_j}}) > d_{e_j} - r_{a_i} then set Status = infeasible
23       set γ = γ ∪ {τ_{e_j}}
24     endif
25   endfor
26 endfor
```

Fig. 3. Algorithm to check the feasibility in the presence of multiple faults.

- add the computational demand of tasks having release times $\geq r_a$ and deadline $= d_m$ (namely, $C(\xi(r_a, d_m))$) and
- recompute $w_g(\phi(r_a, d_j) \cup \xi(r_a, d_m))$ for $g = 1 \ldots k$.

Note that the computation of $w_g(\phi(r_a, d_j) \cup \xi(r_a, d_m))$ for $g = 1 \ldots k$ can be achieved in $O(|\xi(r_a, d_m)| \cdot k^2)$ total time by using the dynamic programming technique in Section 5, assuming that we already have $w_g(\phi(r_a, d_j))$ values.

In summary, if we exclusively focus on intervals $[r_a, d_j]$ that start at $t = r_a$, then we can first consider the interval $I_1 = [r_a, d_{e_1}]$, where d_{e_1} is the *first* task deadline after r_a, and evaluate $h_k(r_a, d_{e_1})$. Then, we can iteratively evaluate $h_k(r_a, d_{e_i})$ by using $h_k(r_a, d_{e_{i-1}})$ and the procedure outlined above.

It is clear that computing $h_k(r_a, d_{e_1}), \ldots, h_k(r_a, d_{e_n})$ cannot require more than $O(n \cdot k^2)$ operations since, at most, a total of n tasks can be added throughout the iterations at distinct deadlines. This observation immediately yields an algorithm of total complexity $O(n^2 \cdot k^2)$ since we can repeat the iterations over deadlines for *every* release time.

The algorithm FT-feasibility, shown in Fig. 3, first orders and puts task release times and deadlines into two lists, $A = [r_{a_1}, \ldots, r_{a_n}]$ and $E = [d_{e_1}, \ldots, d_{e_n}]$. We note that a_i denotes the index of the task with the ith earliest release time, whereas e_i denotes the index of task with the ith earliest deadline. Clearly, both lists can be obtained in $O(n \cdot \log n)$ time.

After computing the recovery overhead functions of single tasks, a nested loop is invoked. The outer loop iterates over increasing release times, whereas the inner loop iterates over increasing deadlines. When we consider an interval $[r_{a_i}, d_{e_j}]$, the set $\phi(r_{a_i}, d_{e_{j-1}})$ is updated by tasks having the deadline d_{e_j}, provided that their release times are not before r_{a_i} (start point of the interval). Once the set of tasks to be added is determined in this way, the worst-case recovery overhead functions, cumulative execution time functions, and, hence, the fault-sensitive processor demand function can be updated efficiently by the technique outlined above. The overall time complexity is $O(n^2 \cdot k^2)$, as explained above.

Now, we return to the example task set in Table 1 and check if it is 2-fault tolerant using the algorithm FT-Feasibility. When we focus on the recovery overhead of faults affecting exclusively single tasks, we easily compute the values of $w_0()$, $w_1()$, and $w_2()$ functions:

$$\begin{array}{lll} w_0(\{\tau_1\}) = 0 & w_1(\{\tau_1\}) = 5 & w_2(\{\tau_1\}) = 10 \\ w_0(\{\tau_2\}) = 0 & w_1(\{\tau_2\}) = 1 & w_2(\{\tau_2\}) = 4 \\ w_0(\{\tau_3\}) = 0 & w_1(\{\tau_3\}) = 6 & w_2(\{\tau_3\}) = 11 \\ w_0(\{\tau_4\}) = 0 & w_1(\{\tau_4\}) = 10 & w_2(\{\tau_4\}) = 15. \end{array}$$

TABLE 2
Trace of FT-feasibility on the Example Problem

Interval	length	γ	$C(\gamma)$	$w_1(\gamma)$	$w_2(\gamma)$	$h_2(I)$
$[r_1, d_1]$	20	$\{\tau_1\}$	5	5	10	15
$[r_1, d_3]$	36	$\{\tau_1, \tau_3\}$	15	6	11	26
$[r_1, d_2]$	40	$\{\tau_1, \tau_3, \tau_2\}$	18	6	11	29
$[r_1, d_4]$	50	$\{\tau_1, \tau_3, \tau_2, \tau_4\}$	28	10	16	44
$[r_2, d_1]$	10	$\emptyset$	0	0	0	0
$[r_2, d_3]$	26	$\{\tau_3\}$	10	6	11	21
$[r_2, d_2]$	30	$\{\tau_3, \tau_2\}$	13	6	11	24
$[r_2, d_4]$	40	$\{\tau_3, \tau_2, \tau_4\}$	23	10	16	39
$[r_3, d_1]$	5	$\emptyset$	0	0	0	0
$[r_3, d_3]$	21	$\{\tau_3\}$	10	6	11	21
$[r_3, d_2]$	25	$\{\tau_3\}$	10	6	11	21
$[r_3, d_4]$	35	$\{\tau_3, \tau_4\}$	20	10	16	36
$[r_4, d_3]$	11	$\emptyset$	0	0	0	0
$[r_4, d_2]$	15	$\emptyset$	0	0	0	0
$[r_4, d_4]$	25	$\{\tau_4\}$	10	10	15	25

Then, we iterate over $[r_{a_i}, d_{e_j}]$ intervals, defined over all (release time, deadline) pairs. In Table 2, we provide the information about each interval I, its length, the tasks that form the fault-sensitive processor demand (the set γ in the algorithm), the values of $w_1(\gamma)$, $w_2(\gamma)$, and $C(\gamma)$, and the total fault-sensitive demand $h_2(I) = C(\gamma) + w_2(\gamma)$. Note that the computation of $w_1(\gamma)$ and $w_2(\gamma)$ is performed by the dynamic programming technique outlined in Section 5.

We observe that there is in fact one (and only one) 2-fault scenario for which the fault-sensitive processor demand exceeds the length of the interval: This is the interval $[r_3, d_4]$. In fact, as can be seen in Fig. 4, τ_4 will miss its deadline if τ_3 and τ_4 both fail just once. In other words, the only 2-fault scenario that can cause a deadline miss is given by the fault pattern $f = \{0, 0, 1, 1\}$ and the task set is not 2-fault tolerant.

7 Extension to Online Settings

The algorithm presented in Section 6 solves the feasibility problem in the presence of k transient faults. However, it has a limitation: It requires *exact* information about task release times a priori. In many real-time applications, this information may not be a priori available. Even if some information about the release times is available, the effects of *jitter* can shift the release times considerably. Thus, an online algorithm that performs fault-sensitive feasibility analysis *as tasks arrive* would be very useful for practical real-time applications. The algorithm that we present in this section is effectively equivalent to an online *admission test* for real-time tasks that can be subject to transient faults at runtime. One additional advantage of the algorithm is its ability to dynamically perform adjustments for tasks that complete without presenting their worst-case workload (a common situation in many applications).

The concepts and techniques developed in the previous sections can be adopted to dynamic settings as follows: Whenever a task is released, the fault-sensitive feasibility analysis will be carried out over the set of *current ready tasks* by taking into account their *remaining* computational demands (and recovery overheads). We note that our approach is parallel to the algorithm presented by Buttazzo and Stankovic [7] in their overload management framework that uses (non-fault-sensitive) processor demand analysis.

The algorithm Online-Feasibility is shown in Fig. 5. It is invoked at every task release time t_a. The algorithm assumes that the ready tasks are ordered according to their deadlines (in fact, the task ready queue of the operating system will directly provide this information when using EDF). At every release time $t = t_a$, the new fault-sensitive processor demand is evaluated by taking into account only the computational demand (and the recovery overheads) in intervals $[t_a, d_1], [t_a, d_2], \ldots, [t_a, d_m]$, where m is the number of ready tasks, in a fashion similar to Algorithm FT-feasibility. One major difference is that, since we run this test dynamically, we consider each ready task *as if it was released at* $t = t_a$ *but with remaining computational time* c_i when reevaluating the new fault-sensitive processor demand.

Online-Feasibility can be invoked at each release time to check the tolerance to k faults. However, it still has a drawback: It repeats the test at every release time **anew** for k faults, regardless of the number of faults detected previously. If (and according to our framework, since) we know that at most k faults will occur during the entire execution interval, then we can decrement k *whenever we detect a fault*. Further, if this is the zth fault encountered in τ_i, we must update the *indices* of the recovery blocks associated with that task since the recovery block $B_{i,z}$ will immediately be activated and the following recovery blocks will be executed in case $B_{i,z}$ fails too.

The procedure Update-FT-parameters (Fig. 6) decrements k, the number of faults to be tolerated after this point. Then, it recomputes the remaining worst-case workload of task τ_i: If the fault is encountered in τ_i, then $B_{i,1}$ will be definitely executed as the immediate workload of the task (this is established by setting c_i to $b_{i,1}$). Further, the indices of the subsequent recovery blocks $B_{i,2}, B_{i,3} \ldots$ are decremented to reflect their running order in the case of any additional faults that might occur in the future. Finally, we update the $w_h(\{\tau_i\})$ values as a preparation for the reinvocation of the algorithm at the next release time.

Complexity. When invoked, Online-Feasibility considers only one *release time*, which is in fact the *current time* (as

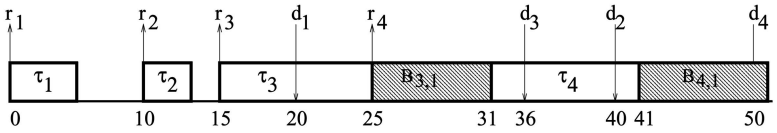

Fig. 4. A worst-case 2-fault scenario resulting in a deadline miss.

Algorithm On-line-Feasibility

Notes:

- The algorithm is invoked at $t = t_a$ when a new task τ_a arrives to the system.
- Ready tasks $\tau_1, \ldots, \tau_m$ are indexed according to their deadlines (τ_1 is the task with the earliest deadline).
- k is the maximum number of faults that must be tolerated in interval $[t_a, d_m]$.
- c_i and $b_{i,j}$ represent the remaining worst-case execution times of task τ_i and recovery block $B_{i,j}$, respectively.

```
1  Insert d_a to the ordered deadline list
2  set w_0(τ_a) = 0
3  for h = 1 to k do
4    set w_h(τ_a) = w_{h-1}(τ_a) + b_{a,h}
5  endfor
6  set C = 0
7  set γ = ∅
6  set Status = feasible
9  /* Compute iteratively the processor demand with recovery overhead for
   every interval [t_a, d_i] */
10 for i = 1 to m do
11   set C = C + c_i
12   for h = 1 to k compute w_h(γ ∪ {τ_i})  using (3)
13   if C + w_k(γ ∪ {τ_i}) > d_i − t_a then set Status = infeasible
14   set γ = γ ∪ {τ_i}
15 endfor
```

Fig. 5. The online algorithm to check the feasibility in the presence of faults.

Algorithm Update-FT-parameters

Notes:

- The algorithm is invoked when a fault is detected in task τ_i (or in recovery block $B_{i,j}$).
- c_i and $b_{i,j}$ represent the remaining worst-case execution times of task τ_i and recovery block $B_{i,j}$, respectively.

```
1  set k = k − 1 /* Decrement the number of faults that must be tolerated after
   this point */
2  set c_i = b_{i,1} /* the recovery block that needs to be executed becomes the current
   computational demand of τ_i */
3  /* Accordingly, update the indices of recovery blocks and w_h(τ_i) values */
4  set w_0(τ_i) = 0
5  for j = 1 to k do
6    set b_{i,j} = b_{i,j+1}
7    set w_j(τ_i) = w_{j-1}(τ_i) + b_{i,j}
8  endfor
```

Fig. 6. Algorithm to update the recovery-related parameters.

opposed to FT-Feasibility that iterates over all release times and deadlines during the offline analysis). Thus, the time complexity of the algorithm Online-Feasibility is $O(m \cdot k^2)$ per invocation, where m is the number of ready tasks and k is the maximum number of faults to be tolerated after this point. The procedure Update-FT-parameters requires $O(k)$ operations.

8 Extension to Periodic Tasks

Up to this point, our framework assumed a general real-time task model in which each task has its distinct ready time, arrival time, and worst-case execution time. However, a large number of real-time applications are *periodic* in nature; especially, widespread digital control and audio/video processing tasks are invoked periodically. Hence, it is worthwhile to investigate the extension of the framework to the periodic execution settings.

A *periodic task* generates a sequence of task instances, each with its own release time and deadline. The inter-arrival time of the instances (that is, the time difference between the arrivals of instances) is given by the *period* of the task. Specifically, we consider a set of independent periodic tasks $\pi = \{\pi_1, \ldots, \pi_n\}$. The period and the worst-case execution time of π_i are denoted by p_i and c_i, respectively. The jth instance of π_i is denoted by $\pi_{i,j}$. The

release time and deadline of $\pi_{i,j}$ are given by $r_{i,j} = r_{i,1} + (j-1)\,p_i$ and $d_{i,j} = r_{i,j} + p_i$, respectively. Without loss of generality, we assume that all periodic tasks are ready at $t = 0$ (that is, $r_{i,1} = 0\ \forall i$). We still commit to the preemptive EDF policy whose optimality in the presence of faults was proven in Section 4.

Each of the π_is instances has the same set of recovery blocks $B_{i,1}, \ldots, B_{i,k}$, with corresponding execution times $b_{i,1}, \ldots, b_{i,k}$. That is, a single task instance $\pi_{i,j}$ may fail up to k times. We assume that the faults affecting different instances of π_i are independent. Our objective is again to determine whether π can be scheduled in a k-fault-tolerant manner, that is, whether all of the task instances can complete in a timely manner under *any* fault pattern with at most k faults.

Since a periodic task generates (potentially) an infinite sequence of task instances, we require that the task set tolerate any k faults that occur in the interval $[0, P]$, where P is the least common multiple of all the periods (commonly known as the *hyperperiod*). Since all of the tasks become simultaneously ready again at $t = P$, it is easy to see that this will enable the system to tolerate any k-fault pattern in any interval $[qP, (q+1)P]$, where q is an integer.

To start with, observe that the methodology developed in earlier sections can still be used by treating each instance of π_i as a different aperiodic task in the interval $[0, P]$. In particular, the necessary and sufficient condition for k-fault tolerance given in Theorem 2 remains valid. This condition can be rewritten as

$$\frac{h_k(t_1, t_2)}{t_2 - t_1} \leq 1 \text{ for all intervals } [t_1, t_2]. \tag{4}$$

Though accurate, the main problem with this straightforward approach is the computational complexity: Since there may be a very large (potentially exponential) number of task instances in the hyperperiod, the overhead may be prohibitive. Specifically, if there are a total of N different task instances (belonging to n periodic tasks), then the application of the algorithm in Section 6 will involve a time complexity of $O(N^2 \cdot k^2)$, which may be exponential if N is large compared to n. Therefore, a simpler and computationally more feasible approach is warranted.

A well-known result established by Liu and Layland's seminal paper [27] is that, in the absence of faults, a periodic task set π is feasible if and only if its total *utilization*, defined as $U_{tot}(\pi) = \sum_{i=1}^{n} \frac{c_i}{p_i}$, does not exceed 100 percent when scheduled by preemptive EDF. Along the same lines, our research for k-fault tolerance of periodic tasks involves a simple bound that can be computed efficiently while providing a *sufficient* condition.

The main result of this section will involve an additional definition: For every periodic task π_i, we define a *corresponding aperiodic task* τ_i with execution time $\frac{c_i}{p_i}$ and k recovery blocks with execution times $\frac{b_{i,1}}{p_i}, \ldots, \frac{b_{i,k}}{p_i}$. Let us denote by τ^π the set of corresponding aperiodic tasks $\{\tau_1, \ldots, \tau_n\}$ obtained from π in this way. We can now present our main result on k-fault-tolerant *periodic* tasks.

Theorem 4. *A set of periodic real-time tasks π is k-fault tolerant if $U_{tot}(\pi) + w_k(\tau^\pi) \leq 1.0$.*

Before giving the proof details, we remark that the sufficient condition above can still be checked in time $O(n^2 \cdot k^2)$, where the dominant term comes from the computation of the worst-case recovery overhead of the corresponding aperiodic task set τ^π.

Proof. We begin by introducing some additional notation. Given an interval $[t_1, t_2]$, we define:

- $Y_i(t_1, t_2)$: The set of instances of the periodic task π_i whose release times and deadlines are fully contained in the interval $[t_1, t_2]$.
- $\eta_i(t_1, t_2)$: The number of instances of the periodic task π_i whose release times and deadlines are fully contained in the interval $[t_1, t_2]$. Formally, $\eta_i(t_1, t_2) = |Y_i(t_1, t_2)|$.

In the following discussion, we omit the parameters of $Y_i()$ and $\eta_i()$ when the interval in question is clear.

From Definition 2, the k-fault-sensitive processor demand in an interval $[t_1, t_2]$ is defined as $h_k(t_1, t_2) = C(\phi(t_1, t_2)) + w_k(\phi(t_1, t_2))$. Using the periodic invocation pattern and the definitions of $Y_i()$ and $\eta_i()$, we can write

$$h_k(t_1, t_2) = \sum_{i=1}^{n} \eta_i(t_1, t_2) \cdot c_i + w_k\left(\bigcup_{i=1}^{n} Y_i(t_1, t_2)\right). \tag{5}$$

Using (4), we can rewrite the necessary and sufficient condition for k-fault tolerance as

$$\frac{\sum_{i=1}^{n} \eta_i(t_1, t_2) \cdot c_i}{t_2 - t_1} + \frac{w_k(\bigcup_{i=1}^{n} Y_i(t_1, t_2))}{t_2 - t_1} \leq 1.0 \quad \text{for all intervals } [t_1, t_2]. \tag{6}$$

We will show the sufficient nature of the bound given in the theorem by proving that, for each interval $[t_1, t_2]$,

$$\frac{\sum_{i=1}^{n} \eta_i \cdot c_i}{t_2 - t_1} \leq U_{tot}^{\pi} \tag{7}$$

and

$$\frac{w_k(\bigcup_{i=1}^{n} Y_i)}{t_2 - t_1} \leq w_k(\tau^\pi). \tag{8}$$

Consequently, whenever the inequality $U_{tot}(\pi) + w_k(\tau^\pi) \leq 1.0$ holds, the k-fault tolerance will be guaranteed.

We begin by showing the inequality in (7). Observe that $\eta_i = \lfloor \frac{t_2 - t_1}{p_i} \rfloor$. Hence,

$$\frac{\sum_{i=1}^{n} \eta_i \cdot c_i}{t_2 - t_1} = \frac{\sum_{i=1}^{n} \lfloor \frac{t_2 - t_1}{p_i} \rfloor \cdot c_i}{t_2 - t_1} \leq \frac{\sum_{i=1}^{n} \frac{t_2 - t_1}{p_i} \cdot c_i}{t_2 - t_1} = \sum_{i=1}^{n} \frac{c_i}{p_i} = U_{tot}^{\pi},$$

establishing the claim.

Validating the inequality in (8) is slightly more involved. Let $\bar{f} = \{f_1, \ldots, f_n\}$ be the k-fault pattern with the largest recovery overhead in interval $[t_1, t_2]$, where f_i denotes the number of faults affecting π_i's instances whose release times and deadlines are totally confined in the same interval. Observe that $\sum f_i = k$. We can write

$$\frac{w_k(\bigcup_{i=1}^{n} Y_i)}{t_2 - t_1} = \frac{\sum_{i=1}^{n} w_{f_i}(Y_i)}{t_2 - t_1} = \sum_{i=1}^{n} \frac{w_{f_i}(Y_i)}{t_2 - t_1} \leq \sum_{i=1}^{n} \frac{w_{f_i}(Y_i)}{p_i \cdot \eta_i}. \quad (9)$$

The latter inequality is justified by the fact that $p_i \cdot \eta_i$ is the *minimum* interval length that can accommodate exactly η_i instances of π_i in $Y_i(t_1, t_2)$.

Moreover, $w_{f_i}(Y_i)$ is certainly smaller than the recovery overhead of the f_i η_i-fault pattern, where each of the η_i instances in Y_i incurs *exactly* f_i faults. However, the latter is equal to $w_{f_i}(\{\pi_{i,m}\}) \cdot \eta_i$, where $w_{f_i}(\{\pi_{i,m}\})$ is simply the (worst-case) recovery overhead of f_i faults, all occurring in a single instance of π_i. Hence,

$$\frac{w_{f_i}(Y_i)}{p_i \cdot \eta_i} \leq \frac{w_{f_i}(\{\pi_{i,m}\}) \cdot \eta_i}{p_i \cdot \eta_i} = \frac{w_{f_i}(\{\pi_{i,m}\})}{p_i}, \quad (10)$$

$$\sum_{i=1}^{n} \frac{w_{f_i}(Y_i)}{p_i \cdot \eta_i} \leq \sum_{i=1}^{n} \frac{w_{f_i}(\{\pi_{i,m}\})}{p_i}. \quad (11)$$

Recalling the definition of the *corresponding aperiodic task* τ_i, we observe that the quantity $\frac{w_{f_i}(\{\pi_{i,m}\})}{p_i}$ is equivalent to $w_{f_i}(\tau_i)$. Combining this observation with (9) and (10), we prove (8) and the theorem. □

9 Nonpreemptive Execution of Recovery Blocks

Our framework assumed a priority-driven real-time execution environment where preemption is allowed for both main tasks and recovery blocks. As a result, a recovery block activated upon the detection of a transient fault may be preempted if a task with high priority (earlier deadline) arrives. The recovery block will resume its execution only when it again becomes the highest priority task in the system. This approach has the merit of considering the timing constraints as the sole criterion for priority assignment and it leads to the result regarding the optimality of the preemptive EDF policy for real-time task systems that rely on a recovery block mechanism to tolerate transient faults (Corollary 1).

However, it is also possible to imagine an environment where the system, upon the detection of a fault, enters the recovery mode and attempts to complete the execution of the recovery block(s) as soon as possible, without any preemption. This gives rise to a model where the recovery blocks (unlike the main tasks) are executed in a nonpreemptive manner. The problem of determining whether a given task set is k-fault tolerant in this new model can be formally stated as follows:

Problem Nonpreemptive-FT. Consider a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of n real-time tasks, where each task τ_i has release time r_i, deadline d_i, worst-case execution time c_i, and recovery blocks $\{B_{i,j}\}$ $j = 1, \ldots, k$. The execution time of $B_{i,j}$ is given by $b_{i,j}$. Is it possible to complete all tasks and potential recovery operations within timing constraints under any fault scenario with at most k faults with the additional constraint that the recovery blocks are to be executed in a nonpreemptive manner?

The general problem of nonpreemptive real-time scheduling is known to be intractable [14]. On the other hand, Nonpreemptive-FT imposes the nonpreemptive execution requirement only for recovery blocks; further, a given recovery block is activated/scheduled only when a transient fault is detected at the end of the main task, which can be executed preemptively and which, by definition, has a nonzero execution time. Nevertheless, this new problem can be shown to be intractable as well.

Theorem 5. *Nonpreemptive-FT is NP-hard.*

Proof. We prove that Nonpreemptive-FT is NP-hard by reduction from PARTITION, which is known to be NP-complete [14]:

Problem PARTITION. Consider a set α of n items, $\{a_1, \ldots, a_n\}$, where there is an integer "size" s_i associated with each item a_i and $\sum_{i=1}^{n} s_i = 2A$. Is it possible to partition α into two disjoint subsets, α_1 and α_2, such that $\sum_{a_i \in \alpha_1} s_i = \sum_{a_i \in \alpha_2} s_i = A$?

We claim that if there were a polynomial-time solution to Nonpreemptive-FT, then it would be possible to solve PARTITION (which is NP-complete) in polynomial time. Given an instance of PARTITION, we will construct a corresponding instance of Nonpreemptive-FT and show that the PARTITION instance admits a YES answer if and only if the instance of Nonpreemptive-FT admits a YES answer.

We define the corresponding Nonpreemptive-FT instance as follows: We will have a set of real-time tasks, τ, where each task τ_i $(i = 1, \ldots, n)$ is given by

$$\begin{aligned} c_i &= \epsilon \quad \text{where } 0 < \epsilon < \frac{1}{n} \\ r_i &= r = 0 \\ d_i &= d = 2A + n\epsilon + 2 \\ b_{i,1} &= b_i = s_i \\ b_{i,j} &= 0 \ (j = 2, \ldots, k). \end{aligned}$$

As can be seen, each task τ_i has only one recovery block, $B_{i,1} = B_i$, whose execution time $b_{i,1} = b_i$ is equal to the "size" s_i of the corresponding item in the PARTITION instance. That is, this construction implies that a given task may be subject to at most one transient fault. The total number of faults the tasks can incur, that is, the parameter k, is set to n. Since $b_{i,j} = 0$ $(j \geq 2)$, there is only one k-fault (n-fault) scenario that we need to consider: This is the scenario where all tasks $\tau_1, \ldots, \tau_n$ fail exactly once.

In addition, we have a last task, τ_{n+1}, with the following parameters:

$$\begin{aligned} c_{n+1} &= 2 \\ r_{n+1} &= A + n\epsilon \\ d_{n+1} &= A + n\epsilon + 2 \\ b_{n+1,j} &= 0 \ (j = 1, \ldots, k). \end{aligned}$$

We first show that if the Nonpreemptive-FT instance has a YES answer, then the corresponding PARTITION instance admits a YES answer as well.

Observe that the release time and deadline of τ_{n+1} are different from those of $\tau_1, \ldots, \tau_n$. Further, it is assumed that it will not be affected by any faults ($b_{n+1,j} = 0 \ \forall j$). Also, τ_{n+1} arrives at $t = n\epsilon + A$ and its execution time $c_{n+1} = 2$ is equal to its laxity. It is clear that, under any

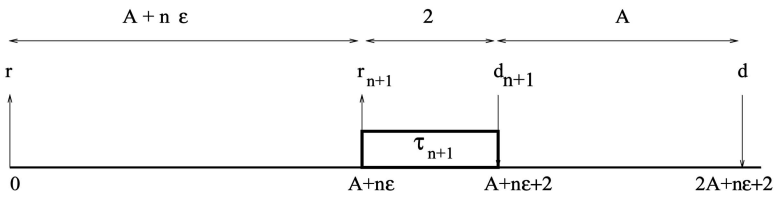

Fig. 7. The timeline for the task set with nonpreemptive recovery blocks.

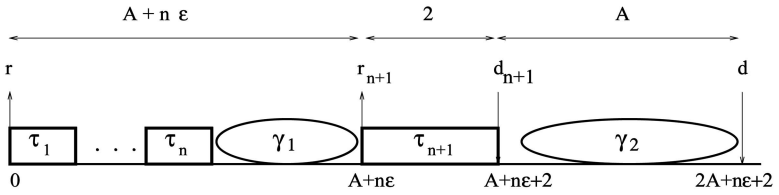

Fig. 8. The hypothetical k-fault-tolerant schedule.

k-fault-tolerant schedule, the time interval $[A + n\epsilon, A + n\epsilon + 2]$ will need to be exclusively reserved for τ_{n+1}. Hence, $\tau_1, \ldots, \tau_n$ and, in the worst-case fault scenario, their recovery blocks $B_1, \ldots, B_n$ need to be executed in intervals $[0, A + n\epsilon]$ and $[A + n\epsilon + 2, 2A + n\epsilon + 2]$ (see Fig. 7). Note that the *entire timeline* $[0, 2A + n\epsilon + 2]$ will have to be used for task and recovery block executions (that is, there will not be any idle CPU time) in this specific fault scenario.

We claim that all of the tasks $\tau_1, \ldots, \tau_n$ need to be fully executed in interval $[0, A + n\epsilon]$ if the task set is k-fault tolerant. To see this, first observe that the total amount of CPU time needed for task executions is $n \cdot \epsilon < 1$. Assume that $\tau_1, \ldots, \tau_n$ are not fully executed in $[0, A + n\epsilon]$. Then, the total amount of available CPU time for the *nonpreemptive* execution of recovery blocks in the interval $[0, A + n\epsilon]$, denoted by W, would be $A + n\epsilon - y$, where $0 < y < n\epsilon < 1$. This gives $A < W < A + 1$, where A is an integer from the assumption. In turn, this implies that one needs to come up with a subset of recovery blocks whose total execution time W in $[0, A + n\epsilon]$ is a noninteger. However, this is not possible in view of the fact that all of the recovery block execution times are integers (from the PARTITION instance) and any subset of integer numbers has an integer sum.

Consequently, $\tau_1, \ldots, \tau_n$ should all be scheduled in interval $[0, A + n\epsilon]$. Then, it becomes clear that the task set is k-fault tolerant (in the nonpreemptive sense) only when it is possible to partition all the recovery blocks $B_1, \ldots B_n$ (with execution times $b_i = s_i \; i = 1, \ldots, n$) in two subsets, each with total execution time exactly A. This is true only if the original PARTITION instance admits a YES answer.

Conversely, assume that the PARTITION instance has a YES answer, that is, we can partition α into two disjoint subsets α_1 and α_2 such that $\sum_{a_i \in \alpha_1} s_i = \sum_{a_i \in \alpha_2} s_i = A$. Further, define $\gamma_1 = \{B_i | a_i \in \alpha_1\}$ and $\gamma_2 = \{B_i | a_i \in \alpha_2\}$. We will show that this implies a YES answer for the corresponding Nonpreemptive-FT instance.

Consider the schedule where $\tau_1, \ldots, \tau_n$ are scheduled one after the other in the interval $[0, n\epsilon]$. Also, to provision for the fault pattern where each task incurs exactly one fault, we can execute the recovery blocks in γ_1 in the interval $[n\epsilon, n\epsilon + A]$ and those in γ_2 in the interval $[A + n\epsilon + 2, 2A + n\epsilon + 2]$ in a *nonpreemptive* manner. The resulting schedule meets the timing constraints even with k faults (see Fig. 8); hence, the corresponding Nonpreemptive-FT instance also has a YES answer. □

10 Related Work

There is a significant body of literature on fault-tolerant real-time systems. Our work falls along the lines of [23], [25], [35], [36], where a maximum number of faults need to be tolerated during the execution of the task set, without any extra assumption about the fault distribution. The approach is named *job-level fault tolerance* in [36] and *fault resilience* in [25]. A number of studies assume that the fault arrival rate follows a probability distribution (for example, Poisson or Weibull distribution [10]) and/or that two successive faults are separated by a minimum known time interval [24], [29], [31]. The first approach (which is adopted in our work) appears to be more general for settings where the fault arrival pattern (or distribution) cannot be predicted accurately, for example, on settings where faults can occur in *bursts*. The second approach has the advantage of exploiting a priori information about the fault arrivals in order to reserve the system resources less conservatively for potential recovery operations. Zhang and Chakrabarty discuss the connections and possible transitions between these two alternative fault models in [36].

It would be fair to state that most of the existing fault-tolerant real-time scheduling literature deals with *fixed-priority settings*. Ghosh et al. extended Rate Monotonic Scheduling (RMS) feasibility bounds to account for the overhead of transient faults in [15]. In [29], Pandya and Malek adopted a recovery model where all unfinished tasks are reexecuted upon the detection of a fault and they derived a utilization bound equal to 50 percent for RMS. Ramos-Thuel and Strosnider extended the *slack stealing* technique to reserve time for potential recovery operations [34]. Han et al. presented an algorithm for settings where each periodic task has two versions: primary and alternate [16]. Their algorithm attempts to complete as many

primaries as possible while attempting to complete the alternate versions before the deadlines in the case of faults.

Researchers from the Real-Time Systems Group at the University of York investigated various aspects of fault-tolerant fixed-priority real-time computing. A well-known response time analysis is extended to account for recovery overhead in [4]. A framework to provide probabilistic timing guarantees in fault-sensitive settings is proposed in [5]. The effects of checkpointing on fixed-priority scheduling analysis were studied in [31].

Lima and Burns presented an algorithm to find the optimal priority assignment during recovery operations, again for fixed-priority systems [24] by assuming a minimum time interval between faults. In [25], they extended the framework by removing this restriction: They perform the schedulability analysis as a function of a *fault resilience* metric, which represents the maximum number of faults that the task may experience.

A framework to provide probabilistic timing guarantees in fault-sensitive settings is proposed in [5]. In [6], Burns et al. provide a generalized framework that assumes probabilistic fault and task arrival patterns, as well as worst-case execution time estimations. Their analysis is formulated in terms of the probability of failure for each task execution.

Research studies built on the EDF policy were much less numerous. Among such studies, we can mention the early work of Chetto and Chetto [11], where an offline schedule for *alternates* is generated and modified at runtime, depending on the occurrence (or absence) of faults. The work in [23] was the first to characterize the feasibility of a real-time task set with EDF under transient faults, although the assumption was that all recovery blocks of a given task have the same execution time. Caccamo and Buttazzo exploited the paradigm of *skippable* real-time tasks to provision for faults while improving the user-perceived quality of service whenever possible [9]. Existing fault-tolerant real-time resource management techniques were revisited to incorporate the effects of low-power design techniques in [26], [35].

After its introduction in [2], [3], the processor demand analysis for deadline-driven systems was extended by Jeffay and Stone to various settings such as scheduling with shared resources [17] and accounting for interrupt handling costs [18]. Buttazzo and Stankovic proposed an online scheme to perform an admission test as tasks arrive dynamically, using the fundamental principles of the processor demand analysis [7]. One main advantage of this technique is that it does not suppose a priori knowledge of release times. More recently, Buttazzo et al. made use of the processor demand analysis to validate their *elastic scheduling* framework [8].

11 Conclusion

In this paper, we considered and effectively solved the problem of checking the feasibility of a real-time task set under any k-fault scenario. We showed how the well-known processor demand analysis can be extended to take into account the CPU time that must be reserved for (potential) recovery operations. That is, we provided an exact characterization of fault-tolerant real-time schedules under worst-case fault scenarios.

Based on this characterization, we provided an efficient and dynamic programming-based algorithm that checks whether an aperiodic real-time task set is k-fault tolerant. Unlike previous solutions, our approach does not assume that the recovery blocks of a given task have the same worst-case execution time. We also provided an online extension of the algorithm that performs the feasibility test at every task release time without requiring the knowledge of future arrivals. The algorithm has the additional feature of adjusting its recovery-related parameters on the fly as faults are detected and recovered from.

Building on this framework, we extended the solution to the periodic execution settings and derived a sufficient condition for the k-fault tolerance of periodic tasks. The time complexity of the solution is $O(n^2 \cdot k^2)$ for n periodic tasks. Finally, we proved that the problem becomes intractable if the recovery blocks are forced to execute in a nonpreemptive fashion.

Our research has some distinct features in that it does *not* assume 1) identical execution times for recovery blocks, 2) any specific fault arrival pattern, or 3) fault-free execution for recovery blocks. However, the analysis is limited to independent tasks. We hope that it will stimulate further research on the analysis of extended task models in the presence of transient faults, such as those given with complex precedence constraints. Nevertheless, another research avenue is to consider extension to multiprocessor systems. For settings where the recovery blocks have to be executed nonpreemptively, further research on polynomial-time approximation algorithms is warranted.

Appendix

Case of Recovery Blocks with Nonincreasing Execution Times

In this part of the paper, first we restate and prove Theorem 3. Then, we show how a fast solution can be devised using the property stated in the theorem.

Theorem 6. *If* $b_{i,j} \geq b_{i,j+1}\ \forall i, j$, *then* $w_k(\tau) = \sum_{j=1}^{k} \bar{b}_j(\tau)$.

Proof. We will show the theorem by induction on n, the number of tasks in τ. Recall that $\bar{b}_i(\tau)$ denotes the ith *largest* value among all recovery block execution times $b_{1,1}, \ldots, b_{n,k}$. Observe that, for a single task τ_x, if $b_{x,j} \geq b_{x,j+1}$, as was assumed, then $\bar{b}_i(\{\tau_x\}) = b_{x,i}$. Then, the base case is easily established: For $n = 1$ (that is, for a task set containing only one task τ_x), $w_k(\{\tau_x\}) = \sum_{j=1}^{k} b_{x,j} = \sum_{j=1}^{k} \bar{b}_j(\{\tau_x\})$.

Assume that the statement holds for all task sets containing less than n tasks. We will show that it remains valid for any task set τ' with n tasks.

τ' can be written as $\tau \cup \{\tau_x\}$ where τ is a set with $n-1$ tasks. From (3), we can write

$$w_k(\tau') = \max_{j=0}^{k} \{w_{k-j}(\{\tau_x\}) + w_j(\tau)\}. \tag{12}$$

Let h be the value of j for which $w_{k-j}(\{\tau_x\}) + w_j(\tau)$ is maximized. In other words, in the worst-case fault

Algorithm WC-Fault-Overhead-2

1 set $w_0(\tau) = 0$
2 sort $b_{1,1}, b_{2,1} \ldots b_{n,1}$ to obtain the non-increasing sequence β
3 for h = 1 to k do
4 set M = the maximum (i.e. first) element ($b_{y,i}$) in β
5 set $w_h(\tau) = w_{h-1}(\tau) + M$
6 set $\beta = \beta - \{M\}$
7 insert $b_{y,i+1}$ to β in the correct order
8 endfor

Fig. 9. Algorithm to compute the worst-case recovery overhead function for recovery blocks with nonincreasing execution times.

pattern $\bar{f}$ for τ', τ incurs exactly h faults and τ_x incurs exactly $k-h$ faults. Since τ has $n-1$ tasks, by induction assumption, $w_h(\tau) = \sum_{j=1}^{h} \bar{b}_j(\tau)$. Similarly, $w_{k-h}(\{\tau_x\}) = \sum_{j=1}^{k-h} \bar{b}_j(\{\tau_x\}) = \sum_{j=1}^{k-h} b_{x,j}$.

Hence, the proof will be complete if we show that $\bar{b}_1(\tau) \ldots \bar{b}_h(\tau)$ and $\bar{b}_1(\{\tau_x\}) \ldots \bar{b}_{k-h}(\{\tau_x\})$ together correspond to $\bar{b}_1(\tau') \ldots \bar{b}_k(\tau')$. Observe that showing the validity of the following two propositions is sufficient for that purpose:

1. All of the elements in $\bar{b}_1(\{\tau_x\}) \ldots \bar{b}_{k-h}(\{\tau_x\})$ are all greater than or equal to $\bar{b}_{h+1}(\tau) \ldots \bar{b}_k(\tau)$ (when $h < k$).
2. All of the elements in $\bar{b}_1(\tau) \ldots \bar{b}_h(\tau)$ are all greater than or equal to $\bar{b}_{k-h+1}(\{\tau_x\}) \ldots \bar{b}_k(\{\tau_x\})$ (when $h > 0$).

To justify proposition 1 of the proof, we need to show that the smallest element in $\bar{b}_1(\{\tau_x\}) \ldots \bar{b}_{k-h}(\{\tau_x\})$, namely, $b_{k-h}(\{\tau_x\})$, is greater than the largest element in $\bar{b}_{h+1}(\tau) \ldots \bar{b}_k(\tau)$, which is $\bar{b}_{h+1}(\tau)$. This is valid for the following reason: Suppose otherwise, that is, $\bar{b}_{k-h}(\{\tau_x\}) = b_{x,k-h} < \bar{b}_{h+1}(\tau)$ when $h < k$, then, the recovery overhead of another k-fault scenario $w_{h+1}(\tau) + w_{k-h-1}(\{\tau_x\})$ would be larger than $w_h(\tau) + w_{k-h}(\{\tau_x\})$, contradicting the assumption that h is the value that maximizes (12) above.

Similarly, to show proposition 2 of the proof, we need to establish that $\bar{b}_{k-h+1}(\{\tau_x\}) \leq \bar{b}_h(\tau)$ when $h > 0$. Suppose otherwise, that is, $\bar{b}_{k-h+1}(\{\tau_x\}) = b_{x,k-h+1} > \bar{b}_h(\tau)$ when $h > 0$. Then, consider the recovery overhead of another k-fault scenario, namely, $w_{h-1}(\tau) + w_{k-h+1}(\{\tau_x\})$. This would again be larger than $w_h(\tau) + w_{k-h}(\{\tau_x\})$, contradicting the definition of h. □

According to Theorem 3, for any task set τ, the worst-case recovery overhead of k-fault patterns may be computed by finding the summation of the largest k recovery blocks over all of the tasks in τ if $b_{i,j} \geq b_{i,j+1}\ \forall i, j$. Algorithm WC-Fault-Overhead-2 (Fig. 9) provides a fast solution. The algorithm exploits the fact that the recovery blocks of a given task τ_i are already in a nonincreasing order by assumption. Hence, sorting the execution times of all *first* recovery blocks (step 2) will also enable us to choose the *largest* among *all* recovery block execution times (say, $b_{y,1}$). By using the same property, we can infer that the second largest recovery block execution time will be either $b_{y,2}$ or a *first* recovery block of another task τ_z, $z \neq y$. In this way, we can identify and sum up the largest k recovery blocks in k iterations (Steps 3-8).

Initial sorting takes $O(n \log n)$ time, whereas the loop (Steps 3-8) can be executed in time $O(n \cdot k)$. The total time complexity is therefore $O(n \log n + n \cdot k)$.

ACKNOWLEDGMENTS

A preliminary version of this paper was published in the *Proceedings of the IEEE Real-Time Systems Symposium* (*RTSS*), 2004.

REFERENCES

[1] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez, "Power-Aware Scheduling for Periodic Real-Time Tasks," *IEEE Trans. Computers,* vol. 53, no. 10, pp. 584-600, Oct. 2004.
[2] S. Baruah, A. Mok, and L. Rosier, "Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor," *Proc. IEEE Real-Time Systems Symp. (RTSS '90),* Dec. 1990.
[3] S. Baruah, R. Howell, and L. Rosier, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor," *Real-Time Systems,* vol. 2, no. 4, pp. 301-324, Nov. 1990.
[4] A. Burns, R. Davis, and S. Punnekkat, "Feasibility Analysis of Fault-Tolerant Real-Time Task Sets," *Proc. Eighth Euromicro Workshop Real-Time Systems,* June 1996.
[5] A. Burns, S. Punnekkat, L. Strigini, and D.R. Wright, "Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems," *Proc. Seventh IFIP Int'l Working Conf. Dependable Computing for Critical Applications (DCCA '99),* Jan. 1999.
[6] A. Burns, G. Bernat, and I. Broster, "A Probabilistic Framework for Schedulability Analysis," *Proc. Third ACM Int'l Embedded Software Conf. (EMSOFT '03),* pp. 1-15, Oct. 2003.
[7] G. Buttazzo and J. Stankovic, "Adding Robustness in Dynamic Preemptive Scheduling," *Responsive Computer Systems: Steps toward Fault-Tolerant Real-Time Systems,* D.S. Fussell and M. Malek, eds., Kluwer Academic, 1995.
[8] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic Scheduling for Flexible Workload Management," *IEEE Trans. Computers,* vol. 51, no. 3, pp. 289-302, Mar. 2002.
[9] M. Caccamo and G. Buttazzo, "Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems," *Proc. Fifth Int'l Workshop Real-Time Computing Systems and Applications (RTCSA '98),* Oct. 1998.
[10] X. Castillo, S.P. McConnel, and D.P. Siewiorek, "Derivation and Calibration of a Transient Error Reliability Model," *IEEE Trans. Computers,* vol. 31, no. 7, pp. 658-671, July 1972.
[11] H. Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Trans. Software Eng.,* vol. 15, no. 10, pp. 1261-1269, Oct. 1989.
[12] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* MIT Press, 1997.
[13] M.L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Information Processing,* 1974.
[14] M. Garey and D. Johnson, *Computers and Intractability.* W.H. Freeman, 1979.
[15] S. Ghosh, R. Melhem, D. Mossé, and J. Sansarma, "Fault-Tolerant Real-Time Scheduling," *Real-Time Systems J.*, vol. 15, no. 2, pp. 149-182, Sept. 1998.
[16] C. Han, K.G. Shin, and J. Wu, "A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults," *IEEE Trans. Computers,* vol. 52, no. 3, pp. 362-372, Mar. 2003.
[17] K. Jeffay, "Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems," *Proc. IEEE Real-Time Systems Symp. (RTSS '92),* Dec. 1992.
[18] K. Jeffay and D.L. Stone, "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems," *Proc. IEEE Real-Time Systems Symp. (RTSS '93),* Dec. 1993.
[19] K.-H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers,* special issue on reliable and fault-tolerant computing, vol. 33, no. 6, pp. 518-528, June 1984.

[20] 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.
[21] 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.
[22] F. Liberato, S. Lauzac, R. Melhem, and D. Mossé, "Global Fault-Tolerant Real-Time Scheduling on Multiprocessors," *Proc. 11th Euromicro Workshop Real-Time Systems,* June 1999.
[23] F. Liberato, R. Melhem, and D. Mossé, "Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems," *IEEE Trans. Computers,* vol. 49, no. 9, pp. 906-914, Sept. 2000.
[24] D.A. Lima and A. Burns, "An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault-Tolerant Hard Real-Time Systems," *IEEE Trans. Computers,* vol. 52, no. 10, pp. 217-231, Oct. 2003.
[25] G. Lima and A. Burns, "Scheduling Fixed-Priority Hard Real-Time Tasks in the Presence of Faults," *Proc. Second Latin-American Symp. Dependable Computing (LADC '05),* pp. 154-173, Oct. 2005.
[26] R. Melhem, D. Mossé, and E. Elnozahy, "The Interplay of Power Management and Fault Recovery in Real-Time Systems," *IEEE Trans. Computers,* vol. 53, no. 2, pp. 217-231, Feb. 2004.
[27] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 46-61, 1973.
[28] A. Maheshwari, W. Burleson, and R. Tessier, "Trading Off Transient Fault Tolerance and Power Consumption in Deep Submicron (DSM) VLSI Circuits," *IEEE Trans. Very Large Scale Integration Systems,* vol. 12, no. 3, pp. 299-312, Mar. 2004.
[29] M. Pandya and M. Malek, "Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks," *IEEE Trans. Computers,* vol. 47, no. 10, pp. 1102-1112, Oct. 1998.
[30] D.K. Pradhan, *Fault-Tolerant Computer System Design.* Prentice Hall, 1996.
[31] S. Punnekkat, A. Burns, and R. Davis, "Analysis of Checkpointing for Real-Time Systems," *J. Real-Time Systems,* vol. 20, no. 1, pp. 83-102, Jan. 2001.
[32] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.,* vol. 1, no. 2, pp. 220-232, June 1975.
[33] *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms,* J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, eds. Kluwer Academic, 1998.
[34] S. Ramos-Thuel and J.K. Strosnider, "Scheduling Fault Recovery Operations for Time-Critical Applications," *Proc. Fifth IFIP Int'l Conf. Dependable Computing for Critical Applications (DCCA '95),* Sept. 1995.
[35] Y. Zhang, K. Chakrabarty, and V. Swaminathan, "Energy-Aware Fault Tolerance in Fixed-Priority Real-Time Embedded Systems," *Proc. Int'l Conf. Computer-Aided Design (ICCAD '03),* Nov. 2003.
[36] Y. Zhang and K. Chakrabarty, "A Unified Approach for Fault Tolerance and Dynamic Power Management in Real-Time Embedded Systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 1, pp. 111-125, Jan. 2006.

Hakan Aydin received the BS and MS degrees in control and computer engineering from Istanbul Technical University in 1991 and 1994, respectively, and the PhD degree in computer science from the University of Pittsburgh in 2001. He is currently an assistant professor in the Computer Science Department at George Mason University, Fairfax, Virginia. He has served on the program committees of several conferences and workshops, including the IEEE Real-Time Systems Symposium and IEEE Real-time Technology and Applications Symposium. He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2006. His research interests include real-time systems, low-power computing, and fault tolerance. He is a member of the IEEE.