On Fault-Sensitive Feasibility Analysis of Real-Time Task Sets

Hakan Aydin
Computer Science Department
George Mason University
Fairfax, VA 22030 - USA
aydin@cs.gmu.edu

Abstract

In this paper, we consider the problem of checking the feasibility of a set of n aperiodic 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 dynamic programming technique. An improvement upon the previous solutions is to address and efficiently solve the case where the recovery blocks associated with faults of a given task do not have necessarily the same execution time. Further, we provide an on-line version of our algorithm that does not require a priori knowledge of release times. The on-line algorithm runs in $O(m \cdot k^2)$ time where m is the number of ready tasks. We also show how to quickly adjust the recovery-related parameters of the algorithm for the remaining part of the execution when a fault is detected.*

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 worst-case execution time and minimum interarrival rate [3, 17].

Providing reliability and availability in safety-critical applications is of paramount importance [21]. Further, in safety-critical *real-time systems*, the fault tolerance techniques must also 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 [21]. *Transient* faults 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 [11, 12]. A recent study indicates that the emerging *low-power* design techniques further increase the susceptibility of VLSI circuits to transient faults [18].

In real-time systems, transient faults are usually addressed through the *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 [22] as the basis of error recovery mechanism. Fault detection tests (in the form of sanity or consistency checks [21]) are performed when a task completes. Should an error be detected, a recovery operation is undertaken by activating a recovery block. In most general case, there may be a need to execute *multiple* recovery blocks (or, *secondaries*) one after the another before a complete recovery is achieved, or a "safe state" is loaded into the memory.

Fault-tolerant real-time scheduling has been addressed by the research community in many ways, in the context of various task/processor models and different scheduling policies [1, 4, 7, 13, 14, 15, 16, 19, 20, 24]. 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 faults?**

We will use Earliest Deadline First (EDF) [17] scheduling policy, because it is known to be optimal for uniprocessor systems under various conditions [8, 17], even in the presence of transient faults [14] (assuming that preemption is allowed). Moreover, our approach will be based on **Processor Demand Analysis** [3, 10, 23]. 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 90's [2, 3]. Later, it was extended by Jeffay et al. to various settings such as scheduling with shared resources [9] and accounting for interrupt handling costs [10]. Buttazzo and Stankovic proposed an on-line scheme to perform an admission test as tasks arrive dynamically, using the fundamental principles of Processor Demand Analysis [5]. One main advantage of this technique is in that it does not suppose a priori knowledge of release times. More recently, Buttazzo et al. made use of Processor Demand Analysis to validate their "elastic scheduling" framework [6].

Contributions of this paper: A first contribution of this research effort is the extension of Processor Demand Analysis for deadline-driven systems to account for recovery overhead due to transient faults. In this aspect, it lays ground for exact fault-sensitive feasibility analysis of extended task models with EDF scheduling. On the other hand, the specific problem that we consider, as specified above, is addressed in [15]. In that work, the authors provide a solution of time complexity $O(n \cdot k^2)$. Our work improves upon [15] in two major ways.

First, unlike [15], we do *not* assume that all the recovery blocks associated with a given task have the same execution time. We believe that relaxing this assumption represents better fault-tolerant real-time applications that rely on recovery block mechanism. For instance, the first recovery block may try to re-execute the task, and in case that another fault is detected at the end of the re-execution, an "alternate" module/version performing an "approximate" computation can be activated (or, a "safe state" can be loaded into 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 of [15] 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. Our algorithm runs in time $O(n^2 \cdot k^2)$.

The second improvement upon [15] is the proposal of a fast, on-line 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 already admitted tasks (with potential 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 actual task release times and this may limit the accuracy of the feasibility test. Our on-line solution has also 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 remaining recovery blocks is efficiently updated to incorporate the knowledge of most recent fault's occurrence.

Although this work deals with aperiodic task systems, it can be extended to *frame-based* systems where a set of real-time tasks having the same period are invoked at regular intervals. In case of non-identical periods, one can always consider each task invocation during the hyperperiod (least common multiple of all periods) as a separate aperiodic request and apply the technique. Finally, on-line extension is applicable to the case of *sporadic* task systems where exact release times of task instances are not known: In systems with n sporadic tasks, at most n task instances can be active at a given time, and our on-line algorithm will be able to check the tolerance to k faults in $O(n \cdot k^2)$ time.

The remainder of this paper is organized as follows: In Section 2, we introduce our system model and notation. In Section 3, we develop the theoretical framework that will serve as the basis for the following sections, namely the extension of Processor Demand Analysis to settings with transient faults. We show how the computation of worst-case recovery overhead function can be achieved in $O(n \cdot k^2)$ time by dynamic programming technique in Section 4. Findings of these sections are combined in algorithm FT-feasibility, that can be used to check the feasibility of the task set under all k-fault scenarios (Section 5). We present the on-line variation of the algorithm in Section 6. We finalize the paper by concluding remarks in Section 7.

2 System Model and Notation

We consider a uniprocessor system with an aperiodic real-time task set $\tau = \{\tau_1, \ldots, \tau_n\}$. 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. Preemption is allowed and we ignore preemption costs.

Our fault model involves *transient faults* that are detected at the end of the execution of each task through sanity or consistency checks. 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.

Throughout the paper, we consider scenarios in which at most k transient faults can occur during the execution of the 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.

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 we know 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 due to running the sanity/consistency checks can be incorporated to c_i and $b_{i,j}$ values.

Formally, a sequence of faults affecting tasks in τ and their recovery blocks is 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 [15]. Further, we say that f is a *k-fault pattern (scenario)* if the number of encountered 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 the recovery blocks associated with this fault pattern will need to be executed. We will use x_f to denote the total execution time of 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,

$$w_k(\tau) = \operatorname*{Max}_{f}\{x_f(\tau)\}$$

Note that $w_k(\tau) \geq w_{k-1}(\tau)$ for $k \geq 1$, since all recovery blocks have non-negative execution times[1]. 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.

We assume preemptive Earliest Deadline First (EDF) scheduling for tasks and recovery blocks throughout the paper. 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 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.

Definition 1 *A task set τ is said to be k-*fault tolerant *(or alternatively, τ can be scheduled in* 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 *feasible* task set in traditional real-time scheduling framework: a task set is feasible if and only if it is 0-fault tolerant.

As a motivational example, consider the task set given in Table 1. When scheduled with EDF in the absence of any faults, all tasks complete well before their deadlines (Figure 1). Further it can be checked 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 the case of a *single* fault. But suppose that we want to know whether this task set can tolerate *any* combinations of two faults. There are $4 * 4 = 16$ different 2-fault scenarios that can occur during the execution of the task set, and generating all the EDF schedules corresponding to the 16 different fault scenarios is clearly not a computationally scalable solution. The reader can easily check that the number of different k-fault scenarios increases *exponentially* with k for any task set τ.

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

Table 1. Example Task Set

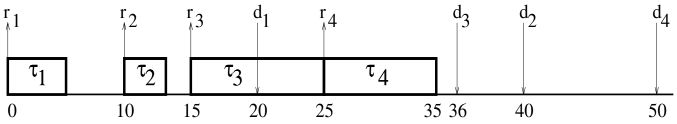

Figure 1. Feasible EDF schedule without faults

3 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 [3, 5, 10]. 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 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 [5, 9, 10] is the following:

Theorem 1 (from [3, 10]) *A set of independent real-time tasks τ can be scheduled (by 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, 5, 23].

[1] Define $w_0(\tau) = 0$ for each task set τ.

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).

Theorem 2 *A set of real-time tasks τ can be scheduled in* 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 k-fault-tolerant manner while 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 affecting exclusively 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 be able to meet all the deadlines with this fault pattern since the total computational demand exceeds available CPU time. The same holds for EDF scheduling policy, hence, a contradiction.

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 $j-$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 selection 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, 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 "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 "if part" we established that the condition is also sufficient for EDF, we have the following[2]:

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

4 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 the later sections for the fault-sensitive feasibility analysis of tasks with timing constraints.

Note that one trivial way of solving this problem would be to generate all $\Theta(n^k)$ potential fault scenarios and compute the recovery overhead associated with each fault pattern. This brute-force technique is clearly infeasible from computational point of view. Below, we outline the details of our solution which has a worst-case time complexity of $O(n \cdot k^2)$. Our solution will be based on **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 affecting exclusively τ_x will result in *consecutive* executions of recovery blocks $B_{x,j}$. That is:

[2] We note that the optimality of EDF in the presence of faults was already given in [14]. However, we find useful to derive it formally in the light of fault-sensitive processor demand analysis.

$$w_0(\{\tau_x\}) = 0 \quad (1)$$
$$w_j(\{\tau_x\}) = w_{j-1}(\{\tau_x\}) + b_{x,j} \text{ for } 1 \leq j \leq k \quad (2)$$

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 k-fault patterns affecting the *augmented* task set $\tau = \tau \cup \{\tau_x\}$. One possibility is that *all* k faults will occur in τ while no fault occurs in τ_x. In this case, $w_k(\tau \cup \{\tau_x\})$ would be simply 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, while tasks in τ can incur collectively $k-j$ 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 <u>maximum</u> of these k cases. Formally:

$$w_k(\tau \cup \{\tau_x\}) = \operatorname*{Max}_{j=0}^{k} \{w_j(\{\tau_x\}) + w_{k-j}(\tau)\} \quad (3)$$

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 Figure 2). The algorithm works bottom-up: 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* the $w_j()$ values ($j \leq k$) along the way according to Equation (3). It is important to note that we need to compute *all* 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 *following* iteration. Finally, observe that the order at which the tasks are processed in this procedure does not affect the final outcome.

Complexity: First, we note that computing worst-case overhead of fault patterns affecting exclusively 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 $O(n \cdot k)$.

5 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 3 and having developed an algorithm to compute the worst-case recovery overhead function for a given task set in Section 4, we can now address the following question: **Given a task set τ, can all tasks and recovery operations be completed within timing constraints under any k-fault scenario?**

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

At first, the application of Theorem 2 does not look promising from computational point of view, because the number of intervals to be examined for 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 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 1 *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 $t_2 > t_1$ 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 non-negative 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 complete at a task deadline.

□

Proposition 1 implies that the number of intervals that need to be checked does not exceed n^2 for n tasks. However, blindly applying the algorithm WC-Fault-Overhead of Figure 2 for each interval would result in an overall complexity of at least

$O(n^3 \cdot k^2)$. On the other hand, a close look reveals that the subsets of tasks that must be considered for each interval are not completely independent:

Proposition 2 *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 $\eta(r_a, d_m)$. Proposition 2 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 following d_j, then (s)he has to only:

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

Note that computation of $w_g(\phi(r_a, d_j) \cup \eta(r_a, d_m))$ for $g = 1 \ldots k$ can be achieved in $O(|\eta(r_a, d_m)| \cdot k^2)$ total time by using the dynamic programming technique of Section 4, assuming that we have already $w_g(\phi(r_a, d_j))$ values.

In summary, if we exclusively focus on intervals $[r_a, d_j]$ that start at $t = r_a$, then we can start with the interval $I_1 = [r_a, d_{e_1}]$ where d_{e_1} is the *first* task deadline following 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 Figure 3, first orders release times and deadlines in 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 task with the i^{th} earliest release time, while e_i denotes the index of task with the i^{th} 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, while 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. Overall the time complexity is $O(n^2 \cdot k^2)$ as explained above.

Now we return to the example task set of Table 1, and check if it is 2-fault-tolerant using the algorithm FT-Feasibility.

Algorithm FT-feasibility

Order the release times such that $r_{a_1} \leq r_{a_2} \ldots \leq r_{a_n}$
/* a_i is the index of the task with i^{th} earliest release time */
Order the deadlines such that $d_{e_1} \leq d_{e_2} \ldots \leq d_{e_n}$
/* e_i is the index of the task with i^{th} earliest deadline */
Status = feasible
/* Compute the recovery overhead of faults affecting single tasks */
for j = 1 to n do
    set $w_0(\tau_j) = 0$
    for m = 1 to k do
        set $w_m(\{\tau_j\}) = w_{m-1}(\{\tau_j\}) + b_{j,m}$
    endfor
endfor
/* Compute the fault-sensitive processor demand for every interval $[r_{a_i}, d_{e_j}]$ iteratively */
for i = 1 to n do
  set $C = 0$
  set $\gamma = \emptyset$
  for j = 1 to n do
    if $r_{e_j} \geq r_{a_i}$ then
      /* τ_{e_j} falls in interval $[r_{a_i}, d_{e_j}]$ */
      set $C = C + c_{e_j}$
      for h = 1 to k compute $w_h(\gamma \cup \{\tau_{e_j}\})$ using (3)
      if $C + w_k(\gamma \cup \{\tau_{e_j}\}) > d_{e_j} - r_{a_i}$ then set Status = infeasible
      set $\gamma = \gamma \cup \{\tau_{e_j}\}$
    endif
  endfor
endfor

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

When we focus on recovery overhead of faults affecting exclusively single tasks, we easily compute the value of $w_0(), w_1()$ and $w_2()$ functions:

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

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),

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

Table 2. Trace of FT-feasibility on the example problem

the values of $w_1(\gamma)$,$w_2(\gamma)$, $C(\gamma)$ 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 4.

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 it can be seen in Figure 4, τ_4 will miss its deadline if τ_3 and τ_4 both fail just once.

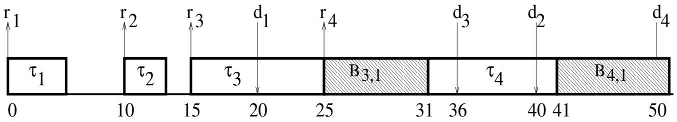

Figure 4. A worst-case 2-fault scenario resulting in deadline miss

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.

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

Figure 5. On-line algorithm to check the feasibility in the presence of faults

6 Extension to On-line Settings

The algorithm presented in Section 5 solves the feasibility problem in the presence of k transient faults. However, it has some limitations: in many real-time applications the exact release times of tasks are not known a priori. Even if some information about the release times is available, the effects of *jitter* can shift the release times considerably. Thus, an on-line algorithm that performs fault-sensitive feasibility analysis *as tasks arrive* would be very useful for practical real-time applications. One additional advantage of the algorithm is that it can 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 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 [5] in their overload management framework that uses (non-fault-sensitive) processor demand analysis.

The algorithm On-line-Feasibility is shown in Figure 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 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 re-evaluating the new fault-sensitive processor demand.

On-line-Feasibility can be invoked at each release time to check the tolerance to k-faults. However, it has still 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 z^{th} fault encountered in τ_i, we must update the *indices* of recovery blocks associated with that task, since the recovery block $B_{i,z}$ will be immediately activated and the following recovery blocks will be executed in case that $B_{i,z}$ fail to update the recovery-related parameters of the task set, when a fault is detected.

The procedure Update-FT-parameters (Figure 6) decrements k, the number of faults to be tolerated after this point. Then it re-computes 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 to the fault-sensitive feasibility analysis at next release time.

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.

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

Figure 6. Procedure to update recovery-related parameters dynamically

Complexity: Note that when invoked, On-line-Feasibility considers only one *release time*, which is in fact *current time* (as opposed to FT-Feasibility that iterates over all release times and deadlines in off-line analysis). Thus the time complexity of the algorithm On-line-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.

7 Conclusion

In this paper, we considered and effectively solved the problem of checking the feasibility of an aperiodic 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 which, unlike previous solutions, does not assume that the recovery blocks of a given task have the same worst-case execution time. We also provided an on-line 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.

We believe that the exact characterization of fault-tolerant schedules under EDF may be particularly helpful in extending this work to different settings, such as sporadic tasks for which minimum interarrival times are known a priori. Yet another research avenue would be to consider other fault models where the minimum time interval between any two consecutive faults is available to the designer.

Acknowledgements: The author would like to thank the anonymous reviewers whose suggestions helped to improve the paper.

References

[1] H. Aydin, R. Melhem and D. Mossé. Tolerating Faults while Maximizing Reward. *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'00)*, 2000.

[2] S. Baruah, A. Mok, and L. Rosier. Preemptively Scheduling Hard-real-time Sporadic tasks on One Processor. In *Proceedings of the Real-Time Systems Symposium (RTSS'90)*, Orlando, Florida. December 1990.

[3] S. Baruah, R. Howell and L. Rosier. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-time Tasks on One Processor. In *Real-Time Systems (2)*, 1990.

[4] A. Bertossi and L. Mancini. Scheduling algorithms for Fault-Tolerance in Hard Real-Time Systems. *Real Time Systems Journal*, 7, 1994.

[5] G. Buttazzo and J. Stankovic. Adding Robustness in Dynamic Preemptive Scheduling. In *Responsive Computer*

Systems: Steps Toward Fault-Tolerant Real-Time Systems, edited by D. S. Fussell and M. Malek, Kluwer Academic Publishers, Boston, 1995.

[6] G. Buttazzo, G. Lipari, M. Caccamo and L. Abeni. Elastic Scheduling for Flexible Workload Management. In *IEEE Transactions on Computers, Vol. 51, No. 3*, March 2002.

[7] M. Caccamo and G. Buttazzo. Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications, RTCSA'98*, Hiroshima, Japan, October 1998.

[8] M.L. Dertouzos. Control Robotics: the Procedural Control of Physical Processes. In *Information Processing*, 1974.

[9] K. Jeffay. Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS'92)*, December 1992.

[10] K. Jeffay and D. L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS'93)*, December 1993.

[11] R.K. Iyer and D.J. Rossetti. A Measurement-based Model for Workload Dependence of CPU Errors. In *IEEE Transactions on Computers*, Vol 33: 518-528, 1984.

[12] R.K. Iyer, D.J. Rossetti and M.C. Hsueh. Measurement and Modeling of Computer Reliability as Affected by System Activity. In *ACM Transactions on Computer Systems*, 4(3): 214-237, Aug 1986.

[13] C. M. Krishna and K. Shin. On Scheduling Tasks with a Quick Recovery From Failure. In *IEEE Transactions on Computers*, 35(5): 448-455, 1986.

[14] F. Liberato, S. Lauzac, R. Melhem and D. Mossé. Global Fault-Tolerant Real-Time Scheduling on Multiprocessors. In *Euromicro Workshop on Real-time Systems*, 1999.

[15] F. Liberato, R. Melhem and D. Mossé. Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-time Systems. *IEEE Transactions on Computers*, 49(9), September 2000.

[16] D. A. Lima and A. Burns. An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault-Tolerant Hard Real-Time Systems. *IEEE Transactions on Computers*, 52: 10, pp. 1332-1346, October 2003.

[17] C.L. Liu and J.W.Layland. Scheduling Algorithms for Multiprogramming in Hard Real-time Environment. *Journal of ACM* 20(1), 1973.

[18] A. Maheshwari, W. Burleson and R. Tessier. Trading Off Transient Fault Tolerance and Power Consumption in Deep Submicron (DSM) VLSI Circuits. *IEEE Transactions on Very Large Scale Integration Systems*, 12(3): 299-312, March 2004.

[19] Y. Oh and S. H. Son. An Algorithm for Real-Time Fault-Tolerant Scheduling in a Multiprocessor System. In *4th Euromicro Workshop on Real-Time Systems*, June 1992.

[20] M. Pandya and M. Malek. Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks. *IEEE Transactions on Computers*, 47(10):1102-1112, 1998.

[21] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, New Jersey, 1996.

[22] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering,* 1(2):220–232, June 1975.

[23] J. Stankovic, M. Spuri, K. Ramamritham and G. Buttazzo (eds). *Deadline Scheduling for Real-Time Systems : EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[24] S. Ramos-Thuel and J.K. Strosnider. Scheduling Fault Recovery Operations for Time-Critical Applications. In *4th IFIP Conference on Dependable Computing for Critical Applications*, 1995.