Computación y Sistemas Vol. 6 No. 1 pp. 051 - 061
© 2002, CIC - IPN. *ISSN 1405-5546* Impreso en México

Scheduling Fault Recovery Operations in Real-Time Systems

Planificación de Operaciones de Recuperación de Fallas en Sistemas de Tiempo Real

Pedro Mejía Álvarez
Centro de Investigación y Estudios Avanzados
Sección de Computación, Av. IPN 2508
Col. Zacatenco, 07300 México D.F.
E-mail: pmejia@cs.cinvestav.mx

Article received on July 06, 2001; accepted on July 02, 2002

Abstract

This paper introduces an integrated framework for the scheduling of non-deterministic workloads, demanded on time-critical recovery operations triggered by the detection of errors in a real-time system. The framework will be developed within the context of fixed-priority preemptive systems. A dynamic analysis for recovery workloads is introduced by developing a criterion for responsiveness of fault recovery operations. This is motivated by the need to verify the timing correctness of real-time workloads under transient recovery workloads and provide graceful degradation to the real-time workload during recovery.

We hypothesize that a scheduler suited to this environment should dispatch tasks using only response time and slack as an admission control mechanism for recovery requests, as long as all deadlines can be met, and that in the presence of transient recovery overloads, the best the scheduler can do is criticality-driven load shedding. The performance of the responsive algorithm is measured quantitatively with simulations using synthetic task sets.

Keywords: Real-Time Scheduling, Fault-Tolerance

Resumen

En este artículo, se presenta un esquema para la planificación de tareas de tiempo real no deterministas, en las que se producen operaciones de recuperación de fallos de tiempo crítico debidas a la detección de errores. Este esquema se desarrolla bajo el contexto de sistemas de tiempo real con prioridades fijas y expulsividad. Se desarrolla un análisis dinámico para cargas de trabajo de recuperación, mediante la introducción de un enfoque de responsividad para operaciones de recuperación de fallos. Este enfoque es motivado por la necesidad de verificar la ejecución a tiempo de las tareas de tiempo real bajo condiciones de fallos, y por la necesidad de proveer una degradación controlada de las tareas durante la recuperación.

El comportamiento de los algoritmos de responsividad ha sido medido en forma cuantitativa mediante extensas simulaciones sobre tareas de tiempo real sintéticas.

Palabras Clave: Planificación de Sistemas de Tiempo Real, Tolerancia a Fallos

1 Introduction

Modern real-time scheduling research has mostly concentrated on generating efficient algorithms for guaranteeing that tasks meet their deadlines without considering faults. Although this is interesting to enable studying different aspects of scheduling theory, the ability to tolerate faults is essential if deadlines are to be met. Work on fault masking achieves this goal by using spatial redundancy, which is expensive. A less expensive approach is that of using time redundancy, which typically does not require a large amount of extra resources and is amenable to uniprocessors. Therefore, this paper focuses on the problem of scheduling real-time tasks along with transient recovery requests in a uniprocessor environment, using time redundancy.

Scheduling recovery time in a real-time system to enhance its fault tolerance is a problem which requires the development of efficient scheduling policies, as well as fault-tolerant mechanisms that resolve resource contention conflicts among different tasks requesting access to a shared resource (Ramos-Thuel, 1994). This is a complex problem because real-time scheduling policies and fault-tolerant mechanisms must allow the scheduler *to meet two competing objectives*, namely, (a) to ensure that the timing requirements of the real-time application workload are met, and (b) ensure that the timing requirements of any time-consuming recovery operation are met and do not violate (a). While the scheduler strives to meet both objectives (a) and (b), there may be situations in which it may be unable to do so. These situations encompass transient overloads, high utilization of the real-time workload, higher than assumed rate of faults, and temporal unpredictability of fault recovery operations.

In this paper, a solution is presented for solving this problem, striving, when possible, to meet the objectives (a) and (b). Our solution comprises an algorithm for scheduling recovery operations to meet the dynamic and widely changing recovery requirements of a real-time system. The amount of intrusiveness of the recovery operations is controlled by the scheduler by servicing the recovery operation at different *levels of responsiveness* (see Section 2), thus enabling run-time criticality-

based control of load shedding. The Responsiveness Algorithm (RA) calculates a responsiveness level for handling a recovery request, and provides a set of priorities at which the recovery could be serviced.

1.1 Background and Motivation

In recent years, computing systems have been used in several applications which have stringent timing constraints, such as embedded robotic industrial systems, computerized intensive care units, autopilot systems, air traffic control, mobile telephones, or communication satellites. *Real-Time systems* are systems whose correctness depends not only on their logical and functional behavior, but also on the temporal properties of this behavior. They can be classified as *hard* real-time systems, in which the consequences of missing a deadline may be catastrophic, and *soft* real-time systems in which missing some deadlines may be allowed. In many real-time applications, fault-tolerance is also an important issue. A system is *fault-tolerant* if it produces correct results even in the presence of faults. Due to the critical nature of tasks supported by by many real-time systems, it is essential that tasks complete before their deadlines even in the presence of failures.

Due to the typical small amount of available resources in embedded real-time systems (Koopman, 1996) and due to the large number of transient faults in such systems (Siewiorek et.al, 1978), we focus our work on the extra time that is required to support recovery from faults. Redundant time can be scheduled prior to execution, in order to achieve predictable timeliness and guarantee that timing constraints are not violated. Such *static* allocation methods require that a specific characterization of the recovery workload be developed before run-time. On the other hand, it is possible to allocate redundant time for recovery *dynamically*. In this approach, there is no *a-priori* notion of pre-allocated processing time for servicing recovery requests. Rather, the scheduler allocates time dynamically when requests for recovery arrive, which has the advantage of not degrading the system performance in case there are no faults.

Recovering from errors in a real-time system is a two step process requiring the *allocation of time* for recovery operations and the subsequent *management* of the allocated time to ensure that the timing constraints of the recovery operations are met (Ramos-Thuel, 1991). Allocation mechanisms may be *intrusive* or *transparent* depending on whether or not performance degradation is imposed on the application workload during recovery. Management of recovery time may be *passive* or *active* depending on the ability of the scheduler to distribute recovery time for faulty tasks. A passive management approach allocates time when the resource is idle, or whenever the application tasks execute less than their estimated worst case computation times. An active management approach is able to schedule recovery operations upon arrival by determining the priority at which recovery is serviced.

Recently, a few studies have begun to emerge in developing schedulability analysis for fault-tolerant real-time task sets for fixed-priority preemptive systems. Some of these studies have dealt with real-time fault-tolerant scheduling (Ghosh et.al, 1995; Betatti et.al, 1993; Gonzalez et.al, 1997; Fohler, 1997; Kopetz, 1990); since these studies do not relate to Rate Monotonic Scheduling (RMS), they are not discussed further.

In (Ramos-Thuel, 1993), Ramos-Thuel presented time reservation algorithms for the static and dynamic scheduling of fixed priority recovery modules; it is one of the first developments of the concept of slack scheduling. Ramos-Thuel also studied dynamically scheduling of recovery operations (Ramos-Thuel, 1991), where a *criticality-driven transient server* is developed. The transient server creates server tasks in response to transient recovery requests. Its implementation requires the creation of a table before run-time to assist the scheduler in *performing on-line decisions during recovery*. Punnekkat (Punnekkat, 1997) extended the analysis in (Joseph et.al, 1986) to develop a worst-case response time analysis for different fault-tolerant mechanisms, and developed probabilistic guarantees for fault-tolerant real-time tasks. In (Ghosh, 1996; Ghosh, 1998), a recovery scheme is proposed to extend the RMS scheme for single and multiple transient faults (FT-RMS). In that model, one fault is tolerated by re-execution of the faulty task, if faults occur with a certain minimum inter-arrival time. A technique called *overloading* is used to share the time interval among several tasks.

Previous studies providing fault tolerance in RMS-based real-time systems have some shortcomings, some of which are addressed in this paper. Toward that, we develop a dynamic analysis for recovery operations by introducing the concept of *levels of responsiveness* for recovery operations. Previous definitions of responsiveness has been based on server policies (e.g., Deferrable and Sporadic Servers) (Sprunt, 1989), where responsiveness is measured using worst-case/average *response time* of aperiodic service requests.

The slack stealing scheduling approach developed in (Ramos-Thuel, 1993; Ramos-Thuel, 1994) offers further improvements in response time over the server approach. The amount of *slack time* "stolen" during a time interval $[t_1, t_2]$ can be used to execute aperiodic requests without causing any periodic request to miss its deadlines. A probabilistic definition of responsive-

ness for fault-tolerant real-time systems was given by Malek (Malek, 1993), within a consensus-based model. This model included timeliness requirements for several algorithms, such as synchronization, reliable communication, diagnosis, scheduling, checkpointing and reconfiguration. However, the model in (Malek, 1993) does not provided analytical arguments for its application.

Previous definitions of responsiveness have yielded a boolean result (yes/no) to the question of whether or not an aperiodic operation can be serviced. In this paper, we extend this concept to define *levels of responsiveness* as the different priorities at which a recovery request can be serviced. Therefore, the question of whether or not a recovery request can be serviced, will depend on the system characteristics and the level of responsiveness. Some levels indicate that recovery can be serviced without harming any task deadline, while some other levels indicate that recovery can only be serviced in an intrusive manner, making some tasks miss their deadlines. It is also possible that the responsiveness level indicate that there is no time for recovery. Overall, our approach based on responsiveness levels focuses in providing run-time flexibility to the scheduler for controlling the amount of intrusiveness in the scheduling of recovery requests.

The remainder of this paper is organized as follows. In Section 2, a framework for integrating responsiveness levels into the scheduling of recovery requests is developed by introducing a responsiveness algorithm, RA. Section 3 provides an evaluation of the performance of the responsiveness algorithm. Finally, Section 4 presents concluding remarks.

2 Dynamic Analysis of Fault Recovery

2.1 Framework and Assumptions

A fixed priority Rate Monotonic Scheduling model (Liu et.al, 1973) is considered, with a set of n independent periodic preemptive [1] tasks. In this model, ϕ_i is the initiation time (or phase), T_i is the period and C_i is the worst case execution time of task τ_i. It is assumed that a task executes *correctly* if its results are produced according to its specification, and delivered before its deadline. A fault occurs when either of these conditions does not hold.

The Rate Monotonic model assigns a higher priority to a task with shorter period, and the priorities remains fixed for the complete execution of the tasks. In this model, tasks are periodic and preemptive, so they execute every T_i units of time. Therefore, a periodic task τ_i give rise to an infinite sequence of jobs. The k^{th} job is ready to execute at time $\phi_i + (k-1)T_i$ and its C_i units of required execution must be completed by time $\phi_i + (k-1)T_i + D_i$, or else a timing fault will occur.

Liu and Layland (Liu et.al, 1973) proved that a task τ_i is guaranteed to be schedulable (meets its deadline) if the deadline of its first job is met when it is initiated at the same time as all higher priority tasks, i.e., $\phi_k = 0, for k = 1,...i$. This is because the time between the arrival of a task's job and the completion of its execution, referred to as its *response time*, is maximized when it arrives at the same instant at which all tasks of equal and higher priority arrive. The phasing scenario in which the initiation times for all tasks are equal is know as the *critical instant*, which is the worst-case phasing. It follows that a set of tasks (workload) is schedulable under Rate Monotonic if the deadline of the first job of every task starting at a critical instant is met.

Liu and Layland also developed a sufficient test for the schedulability of a task set in which task deadlines are equal to the periods, $D_i = T_i$. They proved that if the utilization of the tasks set ($U_t = \sum U_i = \sum C_i/T_i$) is less than 69%, the no task will miss its deadline.

Lehoczky, et.al. 1989, extended these results by deriving a necessary and sufficient schedulability condition for fixed-priority workloads under critical instant phasing. This condition dictates that task τ_i is schedulable if the following condition is met,

$$min_{\{0 \le t \le D_i\}}\{W_i(t)/t\} \le 1 \qquad (1)$$

where $W_i(t)$ is the cumulative work that has arrived from priority levels 1 to i in the time interval $[0, t]$ under critical instant phasing and is computed as,

$$W_i(t) = \sum_{j=1}^{i} C_j \cdot \lceil t/T_j \rceil \qquad (2)$$

The condition $W_i(t)/t \le 1$ is condition is true if for some t in the range $0 \le t \le D_i$ the supply of processing time is more than or equal to the demand for processing time for τ_i.

It follows that the entire task set is schedulable if the maximum value of $W_i(t)/t$ over the minima computed for each task $\tau_i, i = 1, ..., n$, is also less or equal to one, as indicated by,

$$max_{\{1 \le i \le n\}} min_{\{0 \le t \le D_i\}} \{W_i(t)/t\} \le 1 \qquad (3)$$

This analytical framework allows us to evaluate the schedulability of a task set assuming that no processing time is reserved for the recovery of faults.

[1] *Preemptiveness* is a property of real-time tasks in which each tasks can be interrupted from execution at any time by some higher priority task. If tasks are non-preemptive, they will not be interrupted until they finish their current execution.

In this paper we will extend this framework to evaluate the timing impact caused by the the occurrence of faults in the real-time system.

Our fault model considers the following assumptions. There exists a fault detection mechanism to detect transient faults; only one task is affected by each fault. We assume that the sequence of recovery arrivals is not known in advance and that a recovery request is initiated when a fault is detected during the execution of task τ_i. The computation requirement of the recovery request is known at the instant of its arrival and its deadline is the same of the faulty task. The recovery operations can be preempted by either high priority or recovery tasks.

2.2 Problem Formulation

In this paper, we address the problem of dynamically scheduling recovery requests under RMS (Liu, 1973), using the *responsiveness approach*. As all systems have finite resources, their ability to service recovery requests while meeting the temporal requirements is limited. Clearly, transient overload conditions may arise if more tasks and recovery requests have to be scheduled than the available processor resources can handle. Under such conditions, the decision of whether or not a recovery request can be serviced, will depend on the system characteristics and a level of responsiveness.

In our approach, when recovery requests arrive, the scheduler dynamically performs a feasibility test to determine whether or not a recovery request can be accepted. A recovery request is accepted if it can meet its deadline, while being serviced at an assigned *responsiveness level*. Otherwise, it is rejected. The levels of responsiveness are based on the slack available in the schedule and on the criticality of the recovery request. Moreover, the recovery request can be serviced in a transparent or intrusive manner (based on the criticality of the recovery request), that is, without affecting other tasks or by shedding some of the application workload. Hence, the main advantages of our approach are that it: (1) allows the scheduler to assign priorities to service recovery requests dynamically, (2) provides an approach for *criticality-driven* recovery service and load shedding, (3) causes no performance degradation to the application workload in the absence of recovery requests.

Solution Approach

First, it is necessary to compute the slack needed for servicing the recovery request at all levels of priority (see Section 2.3). Once an error is detected, the computed slack available in the schedule is used to determine the feasibility of the recovery request. Second, the responsiveness level for the recovery request should be computed (see section 2.5). The responsiveness level will indicate a set of priorities at which the recovery request can be serviced.

In order to control the level of intrusiveness needed by the recovery workload, the scheduler needs to know **(a)** the amount of slack time available at different priority levels, **(b)** the criticality of the faulty task and of the other real-time tasks, and **(c)** the computation time needed by the recovery operation. Once **(a)**, **(b)** and **(c)** are known, the scheduler has enough information for determining different *levels of responsiveness* for recovery. After a responsiveness level is determined, an *algorithm for handling recovery requests* is invoked dynamically (see section 2.8).

2.3 Slack for Recovery

In order to compute the level of responsiveness of a recovery request, we need to compute the slack available for each task τ_i. Slack can be computed either statically (Ramos-Thuel, 1993) or dynamically (Davis, 1995). These methods schedule a ready aperiodic request at the highest priority whenever the value of the slack is not zero, while (Tia, 1996) uses the value of the slack to determine the priority of the aperiodic request.

The function of our *slack stealer* is augmented by a *slack reclaimer*. A slack reclaimer cooperates with the slack stealer by making available for recovery any processing time unused by the periodic tasks when a fault occurs. In case the task executes for its worst case execution time, the reclaimed time is zero.

Let d_{ij} and t_F denote the deadline and the instant of time in which the fault occurred in job τ_{ij}, respectively. Job τ_{ij} denotes the j^{th} request of task τ_i. At time t_F, we need to compute $SL_i(t_F)$, the slack available within $[t_F, d_{ij}]$ for servicing the recovery request.

According to the analysis provided by Lehoczky (Lehoczky, 1990), it is possible to compute the total cumulative processing made by all jobs of $\tau_1,...,\tau_{i-1}$ and the first job of τ_i during $[0,t]$, under critical instant phasing by

$$W_i(t) = \sum_{j=1}^{i-1} \left(C_j \cdot \left\lceil \frac{t}{T_j} \right\rceil \right) + c_i(t) \qquad (4)$$

where $c_i(t)$ denotes the computation time of the completed portion of τ_i.

From Equation (4) we can compute the amount of computation time in the interval $[t_a, t_b]$ required by periodic requests in the set of tasks with higher priority than τ_i. The cumulative workload CW_i in $[t_a, t_b]$ can be computed by

$$CW_i(t_a, t_b) \leq \sum_{k=1}^{i} C_k \cdot (\lceil t_b/T_k \rceil - \lfloor t_a/T_k \rfloor) - c_k(t_a) - cr_{k,t_b} \quad (5)$$

where $c_k(t_a)$ is the completed execution time of task τ_k at time t_a and cr_{k,t_b} is the remaining computation time of task τ_k at time t_b (i.e., $cr_{k,t_b} = C_k - c_k(t_b)$).

Hence, letting d_F denote the next deadline of the faulty task after time t_F, the slack SL_i for τ_{ij} at time t_F can be computed as below.

$$SL_i(t_F) = d_i - t_F - CW_i(t_F, d_i) + cr_{i,t_f} \quad (6)$$

where cr_{i,t_f} is the remaining computation time of the faulty task.

Equation (6) implies that at time t_F the scheduler must be able to extract the remaining execution time cr_{i,t_f} from task τ_i in order to allocate it to the recovery operation. We assume that during $[t_F, d_F]$ only one recovery request can be serviced for τ_i.

To schedule recovery requests dynamically, our approach is to run a feasibility test using the information given by the slack. This test dictates that if a fault occurs during execution of task τ_i at time t_F, it is schedulable if the following condition holds

$$SL_i(t_F) \geq C_i^F \quad (7)$$

where C_i^F is the timing requirement of the recovery request. Equation (7) forms the basis for the development of the *levels of responsiveness* (see section 2.5), and it does not consider lower priority tasks.

	Fault-Free Workload			Re-execution
	T_i	C_i	D_i	C_i^F
τ_1	20	7	20	5
τ_2	40	10	40	8
τ_3	75	20	75	11

Table 1: Timing Characteristics of the Real-Time Workload and its Recovery

Example of Slack Calculations

For the real-time workload given in Table 1, a given set independent fault arrivals and their associated slack values is shown in Table 2. These values of fault arrivals have been chosen for illustration purposes only.

With the information calculated in Table 2, it is possible to observe that *only if* a fault occurs at $t_F = 5$ or $t_F = 67$, it could happen that task τ_3 misses its deadline. This situation occurs because C_3^F is equal to 11 when re-execution is used for recovery. Therefore, $C_3^F = 11 > SL_3(t_F)$, for $t_F = 5$ or $t_F = 67$. On the other hand, if the fault occurs at one of $t_F = 12, 18, 22, 35, 42, 52$ during execution of task τ_3, there will be enough slack time for recovery using re-execution without missing any deadline.

t_F	Faulty Job	$SL_1(t_F)$	$SL_2(t_F)$	$SL_3(t_F)$
5	$\tau_{1,1}$	15	18	9
12	$\tau_{2,1}$	21	21	12
18	$\tau_{3,1}$	15	31	26
22	$\tau_{1,2}$	18	34	12
35	$\tau_{3,1}$	18	21	16
42	$\tau_{1,3}$	18	21	12
52	$\tau_{2,2}$	21	21	12
67	$\tau_{3,1}$	26	29	8

Table 2: Slack Calculations for Different Values of t_F

2.4 Levels of Responsiveness

In this section we will describe how to compute the *levels of responsiveness*, that is, the set of priorities at which a recovery request should be serviced. Computing several levels of responsiveness gives the scheduler the flexibility needed for servicing recovery requests with dynamic timing constraints. At each level of responsiveness, slack information is computed from the schedule to verify whether or not a recovery request can be serviced. If slack time is not available, it may be necessary to extract time at the expense of some other tasks. This gives the scheduler the flexibility to schedule critical recovery requests at a specific priority, even if this jeopardizes the timely execution of other tasks.

In the following classification, we introduce different levels of responsiveness. We assume that $SL_j(t_F)$ is the amount of slack available for servicing recovery requests for task τ_j at time t_F, as computed above.

- **Too late Recovery.** Indicates that recovery is not possible, even scheduling the recovery operation at the highest priority. This level of responsiveness arises when $(d_F - t_F) < 0$.
- **Non-intrusive Recovery**

  Fair. At this level, slack is computed to verify if it is possible to schedule recovery operations at the lowest possible priority, such that the faulty task, and other tasks do not miss their deadlines.

$$FA_i(t_F) = \begin{cases} min_{j=1}^{n} SL_j(t_F) & if\ SL_j(t_F) \geq C_i^F (j = 1, .., n) \\ 0 & \text{otherwise} \end{cases}$$

At this level, recovery operations can be scheduled at priorities $1, ..., n$.

Greedy early. At this level, slack is computed to verify if recovery can be scheduled at higher priorities without making any task miss its deadline.

$$GE_i(t_F) = \begin{cases} min_{j=1}^{i} SL_j(t_F) & if\ SL_j(t_F) \geq C_i^F (j = 1, .., n) \\ 0 & \text{otherwise} \end{cases}$$

At this level, recovery operations can be scheduled at priorities $1, ..., i$.

- **Intrusive Recovery**

Gracefully late. Indicates that scheduling recovery operations is only possible at higher priorities. At this responsiveness level, lower priority tasks possibly miss their deadlines.

$$GL_i(t_F) = \begin{cases} min_{j=1}^{i} SL_j(t_F) & if\ SL_j(t_F) \geq C_i^F (j = 1, .., i) \\ 0 & \text{otherwise} \end{cases}$$

At this level, recovery operations can be scheduled only at priorities $1, ..., i$.

Critically late indicates that scheduling recovery operations is only possible at the highest priority. This could make higher priority tasks miss their deadlines.

$$CL_i(t_F) = \begin{cases} (d_F - t_F) & if\ ((d_F - t_F) \geq C_i^F) \\ 0 & \text{otherwise} \end{cases}$$

At this level, recovery operations can be scheduled only at priority 1.

Note that the levels above are ordered such that if there is enough slack at one level, there is always enough slack at a level below. For example, if there is enough slack to schedule a recovery request at the Fair Level, it can also be scheduled at lower levels of responsiveness. But, if there is not enough slack, it will be necessary to check in the lower levels to verify the next level at which the recovery request can be scheduled. We assume that the intrusive responsiveness levels are not used when there is enough slack at higher levels to schedule the recovery operations.

When computing the levels of responsiveness, the algorithm gathers information that includes the time of the fault (t_F), the computation time of the recovery operation (C_i^F), and the slack computed for every level of responsiveness. We introduce this computation via an example, as below.

Example of Levels of Responsiveness

Consider the workload described in Table 1. Assume that the following sequence of faults occur: (1) the first fault occurs at time $t_F = 5$, (2) the second at time $t_F = 22$, and (3) the third at time $t_F = 52$.

The resulting execution sequence from this example (Gantt charts are shown in Figure 1) can be explained as follows. After a fault is detected, the levels of responsiveness are computed dynamically[2] (each line of Table 3). In the first fault, recovery of task τ_1 produces a delay of 3 units of time for tasks τ_2 and τ_3. This is because the recovery operation adds 3 units of time to the response time of task τ_1. Job $\tau_{1,2}$ suffers the second fault and its recovery operation is serviced also at some non-intrusive level. In this case, job $\tau_{1,2}$ does not increase its response time because the scheduler reclaims the unused computation time. Therefore, it does not delay lower priority tasks. In the third fault, slack time is computed to verify that job $\tau_{2,2}$ does not harm any task deadline, at any level of responsiveness.

The example above illustrates that the our approach may be followed for guaranteeing the schedulability of a recovery request. For every level of responsiveness the scheduler is able to determine a *set of priorities* for which the recovery can be serviced. The information we gather provides run-time support for the real-time scheduler which allows it to reason about the servicing of recovery operations and introduces a criterion for handling different fault semantics. The order at which the responsiveness levels are calculated is given by the algorithm for scheduling recovery requests developed in Section 2.6.

t_F, C_i^F	CL	GL	GE	FA
$t_F = 5, C_1^F = 5$	15	15	9	9
$t_F = 22, C_1^F = 5$	18	18	12	12
$t_F = 52, C_2^F = 8$	23	21	12	12

Table 3: Levels of responsiveness for different faults.

2.5 Criticality

Task criticality defines how important each task is when compared with the rest of the workload. We assume that this parameter is fixed and given by the application (see Burns et.al 2000, for a method to derive this value), and we use it in our approach to perform scheduling decisions about what task must suffer graceful degradation under overload. The criticality set $\mathbf{M} = \{m(\tau_1), m(\tau_2), ..., m(\tau_n)\}$ defines the relative

[2]Note that, since in our approach the recovery request is serviced at the least intrusive level, it may not be necessary to compute all levels of responsiveness.

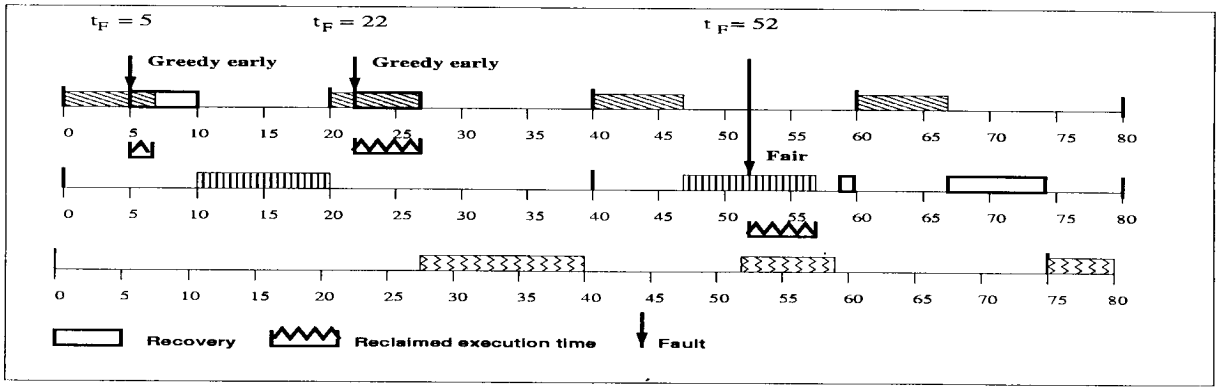


Figure 1: Illustrating Slack Calculations for Multiple Fault Arrivals

criticality of tasks. Let us consider the set of tasks defined in Table 1, τ_1, τ_2, τ_3, with criticality values given by $\mathbf{M} = \{2, 1, 3\}$, which are listed in decreasing order. Thus, τ_2 is the most critical task in the system.

Criticality is used in our approach only when recovery is intrusive. If there is enough slack to service a recovery operation without violating any deadlines, there is no need to include this parameter. However, if not enough slack is available, criticality will provide information to aid the decision of whether to shed the recovery request or other tasks in the system. Therefore, this parameter is used as a decision mechanism for computing the GL and the CL responsiveness levels (the GE and FA levels do not take into consideration the criticality of the tasks). This issue will be described in more detail when the algorithm for scheduling recovery requests is introduced in next section.

2.6 Responsiveness Algorithm

When a recovery request arrives, the scheduler checks first if it is too late for recovery. This is done to expedite the algorithm, because this step only requires to verify if $(d_F - t_F) < C_i^F$. If the request is not immediately denied, the algorithm computes only the levels of responsiveness needed, to determine if the deadline of the recovery request can be met, starting at the FA level[3]. If the responsiveness level indicates that there is enough slack to service a recovery request, it means that it can be serviced at a priority within a range of priorities given at that level, and the algorithm stops. If there is not enough slack at that level, the subsequent level must be calculated. Note that criticality of a task is checked only on intrusive levels (GL,CL), in which the less critical task is always degraded (delayed).

The pseudo-code of the Responsiveness Algorithm (RA) is given in Figure 2.

[3]Note that in the algorithm, the GE level has been skipped, for simplicity.

Event:
A recovery request for task τ_i arrives at time t_F with computation time C_i^F
compute SL_i for $i = 1, ..., n$

Algorithm:
compute $CL_i(t_F)$
/*check if it is too late for recovery */
if $CL_i = 0$ **then** reject the recovery request; exit;
compute $FA_i(t_F)$
if $FA_i(t_F) \geq C_i^F$ **then**
  service the recovery request at some priority given by the FA level
else /* check the criticality of tasks */
  if criticality of faulty task τ_i is not greater than criticality of task τ_j (for $j = i+1, ..., n$) **then**
    reject the recovery request; exit;
  compute $GL_i(t_F)$
  if $GL_i(t_F) \geq C_i^F$ **then**
    service the recovery request at some priority given by the GL level
  else /* check the criticality of tasks */
    if criticality of faulty task τ_i is not greater than criticality of task τ_j (for $j = 1, ..., i-1$) **then**
      reject the recovery request; exit;
    if $CL_i(t_F) \geq C_i^F$ **then**
      service the recovery request at some priority given by the CL level

Figure 2: Responsiveness Algorithm (RA) for Servicing Recovery Requests

Note that a responsiveness level does not indicate the exact priority at which the recovery request can be serviced, instead it gives a set of priorities at which a recovery request can be serviced.

The responsiveness algorithm is based on the following assumptions,

- recovery requests can be preempted by other recovery requests or by periodic tasks.
- calculation of the responsiveness levels will be made progressively, according to the order given by the RA.
- recovery requests have the same deadline of its corresponding faulty task.

3 Performance Evaluation

In this section, several simulations have been conducted to measure the performance of the Responsiveness Algorithm (RA), when using re-execution as the recovery technique. The scheduler dispatches tasks using only the RMS algorithm as long as all deadlines can be met, and that in the presence of fault recovery workloads the scheduler must search for slack available in the schedule for the recovery operation. If there is not enough slack in the schedule the Responsiveness Algorithm performs a criticality-driven load shedding mechanism. The performance of the Responsiveness Algorithm described above is tested through simulations using a synthetic workload, and compared against the performance of:

- RMS without Faults (RMS-NoF). No faults occur, so all admitted tasks run to completion. This is an upper-bound of the performance of the scheduling algorithm. We use the schedulability analysis developed by Lehoczky (Lehoczky, 1989) to verify whether tasks will meet their deadlines.
- RMS with Faults (RMS-F). Within RMS, the occurrence of a fault flags a missed deadline but the task continues to execute. No slack or recovery calculations are performed in this case.
- RMS with Recovery (RMS-Rec). Within RMS, when a fault occurs, the recovery operation is executed at the same priority as the faulty task. The recovery operation (re-execution) is executed without considering if there is enough slack for its successful execution. Therefore, the scheduler does not anticipate a missed deadline due to recovery.
- RMS with Slack (RMS-Slack). Within RMS, when a fault occurs, the slack of the faulty task is compared against the computation time of the recovery operation. If slack is greater, the recovery operation is executed. Otherwise, a deadline is missed, but the unused computation time is reclaimed (in case the fault occurred before the end of the task). This case is important for the comparison because it is the closest to the RA, but it does not consider the criticality of the tasks, as in the RA. Hence it provides a measure for evaluating the importance of the criticality factor. RMS-Slack is equivalent to running RA with the FA level, but without a criticality-driven load shedding mechanism.

Each data point on the graphs represents the average of a set of 50 independent simulations, the duration of each simulation is chosen to be 50,000 time units. The algorithms are executed on tasks sets consisting of 10 periodic tasks, whose parameters are generated as follows. The worst-case execution time C_i is chosen as a random variable with uniform distribution between 5 and 20 time units. The period T_i is computed as a value equal to $T_i = NC_i/U$, where N is the total number of tasks and $U = \sum U_i$ is the load of the task set (recall that $U_i = C_i/T_i$). U is a parameter that varies between 75% and 110%. The task set is sorted based on its computation time, in a way that the lowest priority task has also the lowest computation time.

For each simulation run, we added a recovery workload to the load of the system. The recovery workload is periodic and varied in our experiments between 1% and 10% of the CPU load. We only show the results for 10% for presentation purposes. For the other values, the curves get closer as the recovery workload decreases (i.e., the performance of all algorithms is approximately the same when the recovery workload is 1%). We chose 10% as the upper-bound on the recovery workload following (Siewiorek et.al, 1978). When an instance of task i is chosen as faulty, the exact time for the fault is chosen as a random variable with uniform distribution between 1 and C_i time units (the execution time of the faulty task).

The recovery operation considered is re-execution, which is always carried out at the same priority as the faulty task[4].

The performance of RA was measured by computing the *Deadline Ratio* and the *Value Ratio*. In the Deadline Ratio, criticality is not considered; instead every task in the system has a criticality value equal to one.

Therefore, it is defined as

$$DeadlineRatio = \frac{\text{TnD} - \text{NmD}}{\text{TnD}} \tag{8}$$

The Value Ratio is computed including the criticality value for every task, $m(\tau_i)$.

$$ValueRatio = \frac{\sum_{i=1}^{N} \{\text{m}(\tau_i) * (\text{TnD of } \tau_i - \text{NmD of } \tau_i)\}}{\sum_{i=1}^{N} \{\text{m}(\tau_i) * (\text{TnD of } \tau_i)\}} \tag{9}$$

where: TnD = Total Number of Deadlines and NmD = Number of Missed Deadlines.

The following sections will describe the performance studies carried out to evaluate RA with respect to the RMS Algorithm and its variants. We will show how the use of criticality can affect the performance of the algorithms, and how much the reclaiming mechanism and the behavior of RA can adaptively compensate for

[4] A search for the best possible priority for recovery might give better performance to the algorithm, but it may introduce a large amount of overhead because of the search procedure. For this reason, in the RA algorithm we allow the recovery at the same priority of the faulty task, serving as a lower-bound of its performance.

the performance degradation due to faults, even under overload conditions. In the first study, performance of RA is measured without considering criticality. In the second study, criticality is added to the workload, in which higher priority tasks always have higher criticality values, $m(\tau_0) > m(\tau_n)$. Finally, in the third study, criticality is also included into the workload, but higher priority tasks will have lower criticality values, $m(\tau_n) > m(\tau_0)$.

3.1 RA without Criticality

The first performance study (shown in Figure 3), measures the behavior of the RA algorithm without considering criticality. That is, every task in the system has the same criticality value. In this study, it can be noticed that RA achieves better performance[5] (misses less deadlines) than RMS-F and RMS-Rec.

However, RA achieves the same performance than RMS-Slack. This is because RMS-Slack is equivalent to running RA with the FA level, but without a criticality-driven load shedding mechanism.

Notice that RA achieves a deadline ratio close to that of the RMS-NoF. This means that extracting the unused computation time while recovering introduces an improvement on the Deadline Ratio, despite the extra functionality of recovery from faults.

RMS-F has the lowest performance among the algorithms tested (much lower Deadline Ratio), reflecting the recovery workload (10%). Such situation does not occur with RMS-Rec, RMS-Slack or RA, because they can execute recovery operations.

3.2 RA with Decreasing Criticality

In the second performance study (shown in Figure 4), criticality of the tasks is included. In this case, higher priority tasks have higher criticality: $m(\tau_0) > m(\tau_n)$. Criticality of task τ_i is equal to the computation time of task τ_{n-i} considering $(C_0 < C_1 < \ldots < C_n)$.

Notice from Figures 3 and 4 that the performance of RA with Decreasing Criticality is better than that of RA without criticality, since the addition of criticality allows the algorithm to search for more possible ways to recover a faulty task. More critical tasks (which also have higher priority) will be allowed to recover instead of low-criticality tasks.

In this study RA perform better than RMS-F because RA has the ability to perform recovery and a reclaiming mechanism for handling the unused computation time from faulty tasks.

[5]Note that, when criticality values are the same for all tasks, deadline ratio is the same as value ratio.

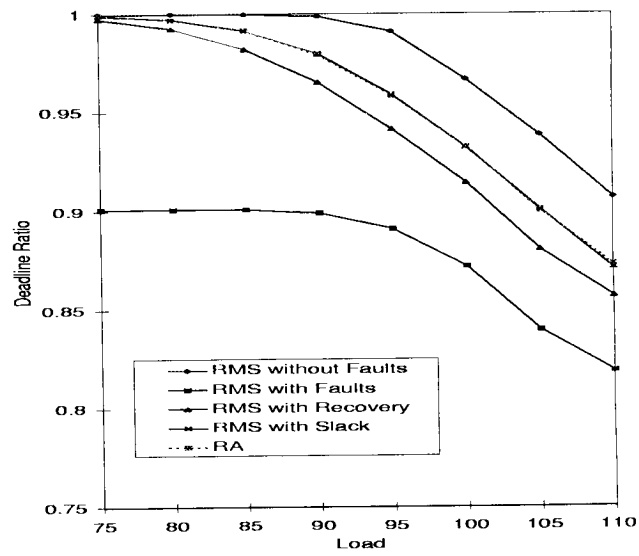


Figure 3: Performance Evaluation without Criticality, 10% Fault Rate.

It is interesting to compare the performance of RA with those of RMS-Rec and RMS-Slack. Good performance can be observed from RMS-Rec because it always executes recovery, even when it may cause a deadline to be missed. This behavior allows RMS-Rec better performance than RMS-F, even under overload conditions. RMS-Slack achieves better performance than RMS-Rec because RMS-Slack can reclaim the unused computation time from faulty tasks, and because it can compute the slack of the faulty tasks, which allows it to decide whether or not a recovery request can be serviced, without harming any task deadline.

However, RA achieves better performance than RMS-Slack, since the reclaiming mechanism of RA and its criticality-driven load shedding properties allows it to perform a fine-tuned search of the time available for recovery and consider the most important tasks.

Further, it is important to note that the responsiveness algorithm is aware of *overloads*, because it detects when there is no slack in the system at any of the responsiveness levels. In this case, the unused execution time of the faulty task is *reclaimed* by the scheduler and re-distributed to other tasks in the system.

3.3 RA with Increasing Criticality

In the third performance study (shown in Figure 5), criticality of the tasks is also considered. In this case, higher priority tasks have lower criticality: $m(\tau_n) > m(\tau_0)$. Criticality of task τ_i is equal to the computation time of task τ_i considering $(C_0 < C_1 < \ldots < C_n)$.

The difference between this study and the study with Decreasing Criticality, is that here we observed worse

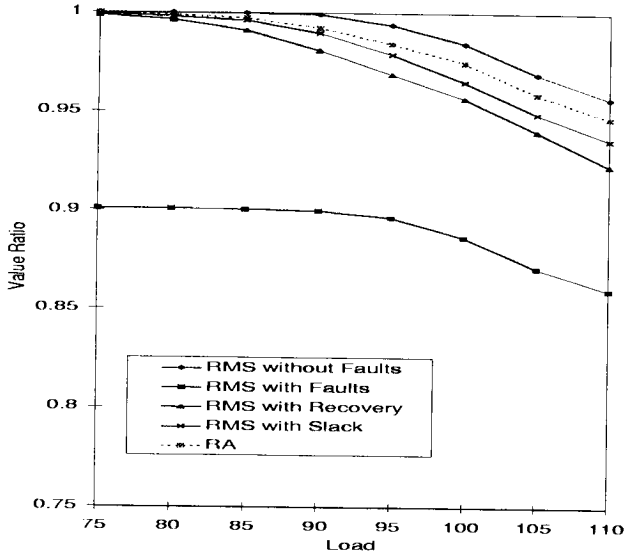


Figure 4: Performance Evaluation with Decreasing Criticality, 10% Fault Rate.

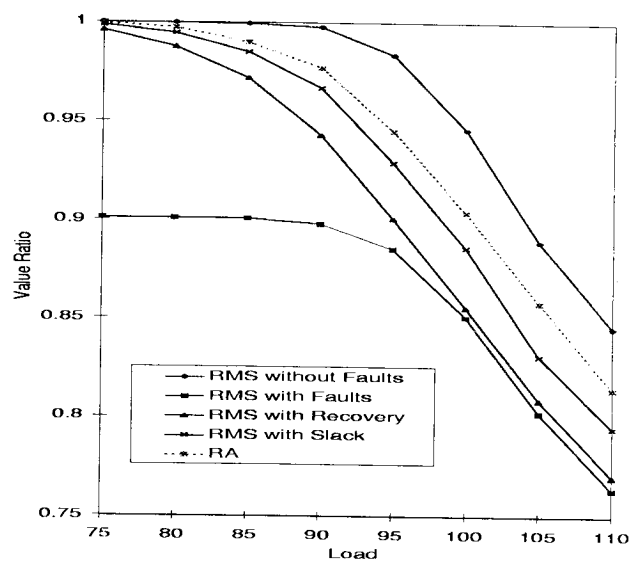


Figure 5: Performance Evaluation with Increasing Criticality, 10% Fault Rate.

performance (Value Ratio decreases more rapidly).

The reason for this behavior is because if a high priority task suffers a fault and there isn't enough slack, the task will not recover because the criticality of the lower priority tasks is always greater. Since higher priority tasks execute more frequently and faults are injected periodically, there will be more chances that they may miss their deadlines.

However, it is important to note that this study is similar to RA with Decreasing Criticality, showing that every algorithm follow the same pattern of behavior (i.e., RA has better performance than RMS-Slack, RMS-Rec, and RMS-F, in that order).

4 Conclusions

The dependability of real-time software can be improved by enhancing the robustness of the scheduler in predicting and controlling the occurrence of timing failures during recovery. This may be achieved by developing strategies which allow the scheduler to dynamically control the manner in which real-time tasks and its time-critical recovery operations are handled in a timely fashion.

In this paper, a scheme was presented to provide scheduling guarantees for fault recovery techniques based on the primary/backup paradigm. A criterion for providing responsiveness to a fault-tolerant scheduler was discussed and some approaches were developed. An algorithm for supporting dynamic scheduling of recovery requests has been developed. Some of the issues involved in supporting run-time scheduling decisions for recovery were illustrated by simulation studies with synthetic workloads. The studies illustrate the effectiveness of the Responsiveness Algorithm with respect to the Rate Monotonic Scheduling Algorithm and its variants. It shows how the use of criticality can affect the performance of the algorithms, and how much reclaiming mechanisms and the behavior of RA can adaptively compensate the performance degradation due to faults, even under overload conditions.

Currently, this scheduling approach is being extended to consider dynamic-priority real-time systems and more in-depth studies of overloaded systems.

References

Betatti R., Bowen N.S., Chung J.Y. "On-Line Scheduling for Checkpointing Imprecise Computation", *Proceedings of the Fifth Euromicro Workshop on Real-Time Systems*, pp. 238-243", 1993.

Burns A., Prasad D., Bondavalli A., Giandomenico F, Ramamritham K, Stankovic J., Strigini L. " The Meaning and Role of Value in Scheduling Flexible Real-Time Systems", *Journal of Systems Architecture, Vol. 46, pp. 46-77, 2000*

Davis R.I. "On Exploiting Spare Capacity in Real Time Systems", *PhD. Thesis, Dept. CS. U. of York, U.K*, 1995.

Fohler G. "Adaptive Fault Tolerance with Statically Scheduled Real-Time Systems", *Proceedings EuroMicro Workshop on Real-Time Systems*, Toledo, Spain 1997.

Ghosh S. "Guaranteeing Fault-Tolerant Through Scheduling in Real-Time Systems", *PhD. Thesis, Dept. Computer Science. University of Pittsburgh*, , 1996.

Kopetz H., Kantz H., Grunsteidl G., Pushner P., Reisinger J. "Tolerating Transient Faults in MARS", *Proceedings Symp. on Fault Tolerant Computing (FTCS-20)*, pp. 446-473. IEEE 1990.

Koopman P. "Embedded System Design", *Proceedings of the International Conference on Computer Design (ICCD 96)*, October 7-9, 1996 - Austin, TX. IEEE.

Ghosh S., Melhem R., Mossé D. "Fault Tolerant Rate Monotonic Scheduling", *J. of Real-Time Systems*, October 1998.

Ghosh S., Melhem R., Mossé D. "Enhancing Real-Time Schedules to Tolerate Transient Faults", *Proceedings of the IEEE Real Time Systems Symposium*, 1995. pp. 120-129.

Gonzalez O., Shrikumar H., Stankovic J., Ramamritham K. "Adaptive Fault Tolerance and Graceful Degradation under Dynamic Hard Real-Time Systems", *Proceedings of the IEEE Real Time Systems Symposium*, 1997.

Joseph M., Pandya P. "Finding Response Times in a Real-Time System", *Computer Journal*, pp-390-395, October 1986.

Lehoczky J. "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines", *Proceedings of the IEEE Real Time Systems Symposium*, pp. 201-209, 1990.

Lehoczky J., Sha L., Ding Y. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", *Proceedings of the IEEE Real Time Systems Symposium*, pp. 166-171, 1989.

Liu C.L., Layland J. "Scheduling Algorithms for Multiprogramming in Hard Real-Time Environments", *Journal of the ACM* 20(1):46-61, January 1973.

Malek M. "A Consensus-Based Model for Responsive Computing", *IEICE Transactions on Information and Systems.*, Vol. E76-D, No.11, Nov. 1993.

Punnekkat S. "Schedulability Analysis for Fault Tolerant Real Time Systems", *PhD. Thesis, Dept. Computer Science, University of York*, June 1997.

Ramos-Thuel S., Strosnider J. "The Transient Server Approach to Scheduling Time-Critical Recovery Operations", *Proceedings of the IEEE Real Time Systems Symposium*, 1991.

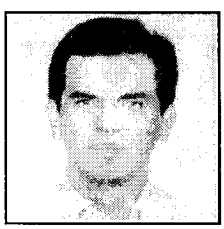
Ramos-Thuel S., Lehoczky J.P. "Algorithms for Scheduling Hard Aperiodic Tasks in Fixed Priority Systems using Slack Stealing", *Proceedings of the IEEE Real Time Systems Symposium*, 1994.

Ramos-Thuel S. "Enhancing Fault Tolerance of Real-Time Systems Through Time Redundancy", *PhD. Thesis. Department of Electrical and Computer Engineering, Carnegie Mellon, University*, May 1993.

Sprunt B., Sha L., Lehoczky J.P. "Aperiodic Task Scheduling for Hard Real-Time Systems", *Journal of Real-Time Systems*, 1989.

Siewiorek D.P., Kini V., Mashburn H., Mcconnel S., Tsao M. "A Case Study of C.mmp, Cm*, and C.vmp: Part 1 - Experiences with Fault Tolerance in Multiprocessor Systems", *Proceedings of the IEEE*, 66(10):1178–1199, Oct. 1978.

Tia T.S., Liu J., Shankar M. "Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed-priority Preemptive Systems", *Journal of Real-Time Systems*, January 1996.



Pedro Mejia Alvarez *was born in Morelia, México in 1963. He received his BS degree in computer systems engineering from ITESM, Queretaro, Mexico, in 1985, and a PhD degree in informatics from the Universidad Politécnica de Madrid, Spain, in 1995. He has been an assistant profesor in Seccion de Computación CINVESTAV-IPN Mexico since 1997. In 1999, he held a research faculty position it the Department of Computer Science at the University of Pittsburgh and, in 2000, a visiting assistant professor position in the Department of Information Sciences and Telecommunications at the University of Pittsburgh. Previously, he held a researcher position at the Electrical Research Institute in Cuernavaca, Mexico, where he was involved in the design and implementation of a multiprocessor real-time operating system for a SCADA system for electrical substations. He is a fellow of the National System of Researchers of Mexico (SNI). His main research interests are real-time systems scheduling adaptive fault tolerance, and software engineering.*