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

1

ADDING ROBUSTNESS IN DYNAMIC PREEMPTIVE SCHEDULING

Giorgio C. Buttazzo* and John A. Stankovic**

** Scuola Superiore S. Anna, Pisa, Italy*
*** University of Massachusetts at Amherst, Massachusetts, USA*

ABSTRACT

In this paper we introduce a robust earliest deadline scheduling algorithm for dealing with hard aperiodic tasks under overloads in a dynamic real-time environment. The algorithm synergistically combines many features including dynamic guarantees, graceful degradation in overloads, deadline tolerance, resource reclaiming, and dynamic re-guarantees. A necessary and sufficient schedulability test is presented, and an efficient $O(n)$ guarantee algorithm is proposed. The new algorithm is evaluated via simulation and compared to several baseline algorithms. The experimental results show excellent performance of the new algorithm in normal and overload conditions.

1 INTRODUCTION

Static real-time systems are designed for worst case situations. Assuming that all the assumptions made in the design and analysis are correct, we can say that the level of guarantee for these systems is absolute, and all tasks will make their deadlines. Unfortunately, static systems are not always possible because for many applications the environment and system itself, being imperfect, violate their assumptions fairly often, or because to develop a static design with absolute guarantees is too costly. When either or both of these conditions exist, we find dynamic real-time systems. In these systems, absolute guarantees are not attained. A property of a dynamic real-time system should be a *minimum level of guarantee* together with best effort beyond this minimum. To build a system with a reasonable cost, we may perform on–line guarantee, where tasks

are accepted based on the current load. In this way, a good processor utilization can be achieved, as well as a predictable behavior.

Many dynamic real-time systems rely on the earliest deadline first (EDF) scheduling algorithm. This algorithm has been shown to be optimal under many different conditions. In spite of this, EDF has one major negative aspect. If overload occurs, tasks may miss deadlines in an unpredictable manner, and in the worst case, the performance of the system can approach zero effective throughput [11]. By using EDF in a planning mode that performs dynamic guarantees, an effective and robust version of EDF scheduling can be developed. In this paper we develop such an algorithm, called RED (robust earliest deadline).

To increase flexibility in expressing time constraints and to enhance the performance of the system in overload conditions, our RED algorithm separates deadline and importance by introducing two additional parameters into the task model: a task *value*, which reflects the importance of the task in the set, and a deadline *tolerance*, which is the amount of time by which a specific task is permitted to be late. Moreover, to introduce a minimum level of guarantee during overloads, we consider two classes of tasks, called *HARD* and *CRITICAL*:

- *HARD* tasks are those tasks that, once accepted, are guaranteed to complete within their deadline in underload conditions;
- *CRITICAL* tasks are those tasks that, once accepted, are guaranteed to complete within their deadline in underload conditions and in overload conditions.

In summary, our main contribution is the development and the evaluation of a robust real-time scheduling algorithm which is largely a synthesis and extension of many ideas found in the literature (see section 6). The result is an algorithm with the following characteristics:

- it operates in normal and overload conditions with excellent dynamic performance and avoids the major negative aspect of EDF scheduling;
- it is used in planning mode so as to predict deadline misses; one extension to previous work is how predictions are made in that we now can depict the size of the overload, its duration, and its overall impact on the system;

- it includes extended timing semantics based on a deadline tolerance per task that is suitable to many control applications, such as robotics;
- it is easily and cost effectively implementable ($O(n)$ complexity),
- it separates guarantee, dispatching and rejection policies so that it can be easily modified for different situations,
- it reclaims resources to improve performance, and dynamically attempts re-guarantees when resources are reclaimed, and
- the major properties of the algorithm are formally proven.

A performance study is accomplished via simulation. Our new algorithm is compared against the standard earliest deadline algorithm and a guarantee based earliest deadline algorithm [13]. We compare the algorithms under widely varying conditions with respect to load, arrival rates, value distributions, allowed tolerances, and actual versus worst case execution times. The new algorithm significantly outperforms these baselines in all tested situations.

2 TERMINOLOGY AND ASSUMPTIONS

Before we describe the guarantee algorithm, we first state our definitions, notations, and assumptions:

J denotes a set of active aperiodic tasks J_i ordered by increasing deadline, J_1 being the task with the shortest absolute deadline.

a_i denotes the arrival time of task J_i, i.e., the time at which the task is activated and becomes ready to execute.

C_i denotes the maximum computation time of task J_i, i.e., the worst case execution time ($wcet$) needed for the processor to execute task J_i without interruption.

c_i denotes the dynamic computation time of task J_i, i.e., the remaining worst case execution time needed for the processor, at the current time, to complete task J_i without interruption.

d_i denotes the absolute deadline of task J_i, i.e., the time before which the task should complete its execution, without causing any damage to the system.

D_i denotes the relative deadline of task J_i, i.e., the time interval between the arrival time and the absolute deadline.

m_i denotes the deadline tolerance of task J_i, i.e., the maximum time that task J_i may execute after its deadline, and still produce a valid result.

v_i denotes the task value, i.e, the relative importance of task J_i with respect to the other tasks in the set.

f_i denotes the estimated finishing time of task J_i, i.e., the time according to the current schedule at which task J_i should complete its execution and leave the system.

L_i denotes the laxity of task J_i, i.e., the maximum time task J_i can be delayed before its execution begins.

R_i denotes the residual time of task J_i, i.e., the length of time between the finishing time of J_i and its absolute deadline.

It is easy to verify the following relationships among the parameters defined above:

$$d_i = a_i + D_i \tag{1.1}$$

$$L_i = d_i - a_i - C_i \tag{1.2}$$

$$R_i = d_i - f_i \tag{1.3}$$

$$f_1 = t + c_1; \qquad f_i = f_{i-1} + c_i \quad \forall i > 1 \tag{1.4}$$

In our more robust model, an aperiodic task J_i is completely characterized by specifying its worst case execution time C_i, its class, its relative deadline D_i, its deadline tolerance m_i, and its value v_i. In the following, we assume that the task class can be derived from the task value. In particular, tasks with maximum value $v_i = V_{max}$ will be considered as $CRITICAL$.

In summary, an aperiodic task set will be denoted as follows:

$$J = \{J_i(C_i, D_i, m_i, v_i),\ i = 1\ to\ n\}$$

Within this framework, different policies are used for handling aperiodic tasks in a robust fashion. In particular, tasks are scheduled based on their deadline, guaranteed based on C_i, D_i, m_i, v_i, and rejected based on v_i.

Throughout our discussion, we assume that a set of aperiodic tasks is scheduled on a uniprocessor system by the Earliest Deadline First (EDF) scheduling algorithm, according to a preemptive scheduling discipline, so that the processor is always assigned to the task whose deadline is the earliest. Moreover, we assume that arrival times are not known a priori. Groups of tasks with precedence constraints can also be handled by EDF by modifying their deadlines and release times so that both deadlines and precedence relations are meet [3] [5].

3 SCHEDULABILITY ANALYSIS

We formulate the dynamic, on-line, guarantee test in terms of residual time, which is a convenient parameter to deal with both normal and overload conditions. We first present the main results without the notion of deadline tolerance, and then we will extend the algorithm by including tolerance levels and task rejection policy. The basic properties stated by the following lemmas and theorems are used to derive an efficient $O(n)$ algorithm for analyzing the schedulability of the aperiodic task set whenever a new task arrives in the system. Due to space limitation, all proofs are omitted. See [4] for complete proofs.

Lemma 1 *Given a set $J = \{J_1, J_2, ..., J_n\}$ of active aperiodic tasks ordered by increasing deadline, the residual time R_i of each task J_i at time t can be computed by the following recursive formula:*

$$R_1 = d_1 - t - c_1 \tag{1.5}$$

$$R_i = R_{i-1} + (d_i - d_{i-1}) - c_i. \tag{1.6}$$

Lemma 2 *A task J_i is guaranteed to complete within its deadline if and only if $R_i \geq 0$.*

Theorem 3 *A set $J = \{J_i,\ i = 1$ to $n\}$ of n active aperiodic tasks ordered by increasing deadline is feasibly schedulable if and only if $R_i \geq 0$ for all $J_i \in J$.*

Notice that if we have a feasibly schedulable set J of n active aperiodic tasks, and a new task J_a arrives at time t, to guarantee the new ordered task set $J' = J \cup \{J_a\}$ we only need to compute the residual time of task J_a and the residual times of tasks J_i such that $d_i > d_a$. This is because the execution of J_a

does not influence those tasks having deadline less than or equal to d_a, which are scheduled before J_a.

Now we introduce a new framework for handling real-time aperiodic tasks under overload conditions, and we propose a robust version of the Earliest Deadline algorithm. Before we describe such a robust algorithm, we define few more basic concepts.

3.1 Load Calculation

In a real-time environment with aperiodic tasks, a commonly accepted definition of workload refers to the standard queueing theory, according to which a load ρ, also called *traffic intensity*, represents the expected number of task arrivals per mean service time [19]. This definition, however, does not say anything about task deadlines, hence it is not as useful in a hard real-time environment.

A more formal definition has been proposed in [1], in which is said that a sporadic real-time environment has a loading factor b if and only if it is guaranteed that there will be no interval of time $[t_x, t_y)$ such that the sum of the execution times of all tasks making requests and having deadlines within this interval is greater than $b(t_y - t_x)$. Although such a definition is more precise than the first one, it is still of little practical use, since no on-line methods for calculating the load are provided, nor proposed.

We propose an efficient procedure to compute the processor workload, which allows to create a complete load profile, and predict the magnitude and the time (intervals) at which overloads might occur. Before we explain our method of computing system load, we introduce the following notation:

$\rho_i(t_a)$ indicates the processor load in the interval $[t_a, d_i)$, where t_a is the arrival time of the latest arrived task in the aperiodic set,

ρ_{max} indicates the maximum processor load among all intervals $[t_a, d_i)$, $i = 1$ to n, where t_a is the arrival time of the latest arrived task in the aperiodic set.

In practice, the load is computed only when a new task arrives, and it is of significant importance only within those time intervals $[t_a, d_i)$ from the latest

arrival time t_a, which is the current time, and a deadline d_i. Thus the load computation can be simplified as:

$$\rho_i(t_a) \;=\; \frac{\sum_{d_k \le d_i} c_k}{d_i - t_a}.$$

Theorem 4 *The load $\rho_i(t_a)$ in the interval $[t_a, d_i)$ can be directly related to the residual time R_i of task J_i, according to the following relation:*

$$\rho_i = 1 - \frac{R_i}{d_i - t_a}. \tag{1.7}$$

It is important to point out that, within the interval $[t_a, d_n]$ between the latest arrival time t_a and the latest deadline of task J_n, the processor work load is not constant, but it varies in each interval $[t_a, d_i)$. To express this fact, we define the following *load function*:

$$\rho(t_a, t) = \begin{cases} \rho_1 & \text{for } t_a \le t < d_1 \\ \rho_i & \text{for } t \in [d_{i-1}, d_i) \\ 0 & \text{for } t \ge d_n \end{cases}$$

Definition 1 *Let ρ_{max} be the maximum of the load function $\rho(t_a, t)$ in the interval $[t_a, d_n]$. We say that the system is* **underloaded** *if $\rho_{max} \le 1$, and* **overloaded** *if $\rho_{max} > 1$.*

Definition 2 *We define Exceeding Time E_i of a task J_i as the time that task J_i will execute after its deadline, that is: $E_i = max_i(0, -R_i)$. We then define Maximum Exceeding Time E_{max} as the maximum among all E_i in the tasks set, that is: $E_{max} = max_i(E_i)$.*

Notice that, in underloaded conditions ($\rho_{max} \le 1$), $E_{max} = 0$, whereas in overload conditions ($\rho_{max} > 1$), $E_{max} > 0$.

Observation 1 *Once we have computed the load factor ρ_i for task J_i, the next load factor ρ_{i+1} can be computed as follows:*

$$\rho_{i+1} = \frac{\rho_i(d_i - t_a) + c_{i+1}}{d_{i+1} - t_a}.$$

3.2 Localization of exceeding time

By computing the load function, we can have a global picture of the system load, and we can see in advance the effects of an overload in the system. For instance, we can see whether the overload will cause a "*domino effect*", in which all tasks will miss their deadlines, or whether it is transient and it will extinguish after a while. In other words, we are able to locate the time or times at which the system will experience the overload, identify the exact tasks that will miss their deadlines, and we can easily compute the amount of computation time required above the capacity of the system – the exceeding time.

This global view of the system allows us to plan an action to recover from the overload condition. Our approach is general enough that many recovering strategies can be used to solve this problem. The recovery strategy we propose in this paper is described in Section 4.

3.3 Deadline Tolerance

In many real applications, such as robotics, the deadline timing semantics is more flexible than scheduling theory generally permits. For example, most scheduling algorithms and accompanying theory treat the deadline as an absolute quantity. However, it is often acceptable for a task to continue to execute and produce an output even if it is late – but not too late. Another real application issue is that once some task has to miss a deadline, it should be the least valuable task. In order to more closely model this real world situation, we permit each task to be characterized by two additional parameters: a deadline tolerance m_i, and a value v_i. The deadline tolerance is then the amount of time by which a specific task is permitted to be late, and the task value denotes the relative importance of the task in the set.

Notice that, when using a dynamic guarantee paradigm, a deadline tolerance provides a sort of compensation for the pessimistic evaluation of using the worst case execution time. For example, without tolerance, we could find that a task set is not feasibly schedulable, and hence decide to reject a task. But, in reality, the system could have been scheduled because, with the tolerance and full assessment of the load, we might determine that overload is simply for this task and it is within its tolerance level. Another positive effect of the tolerance is that various tasks could actually finish before their worst case times so the resource reclaiming part of our algorithm could then compensate and

the guaranteed task with tolerance could actually finish on time. Basically, our approach minimizes the pessimism found in a basic guarantee algorithm.

We recall that, to offer a minimum level of guarantee in overload conditions, we split real-time tasks into two classes:

- $HARD$ tasks are those tasks that, once accepted, are guaranteed to complete within their deadline in underload conditions;
- $CRITICAL$ tasks are those tasks that, once accepted, are guaranteed to complete within their deadline in underload and in overload conditions.

4 THE RED SCHEDULING STRATEGY

When dealing with the deadline tolerance factor m_i, each Exceeding Time has to be computed with respect to the tolerance factor m_i, so we have: $E_i = max(0, -(R_i + m_i))$.

The execution time of $CRITICAL$ tasks in overload conditions is then guaranteed by using a rejection strategy that removes non critical tasks based on their values. Several rejection strategies can be used for this purpose. As discussed in the next section on performance evaluation, two rejection strategies have been implemented and compared. The first policy rejects a single task (the least value one), while the second strategy tries to reject more tasks, but only if the newly arrived task is a $CRITICAL$ task.

To be general, we will describe the RED algorithm by assuming that, in overload conditions, some rejection policy will search for a subset J^* of least value (non critical) tasks to reject in order to make the current set schedulable. If J^* is returned empty, then the overload cannot be recovered, and the newly arrived task cannot be accepted. Clearly, $CRITICAL$ tasks previously guaranteed cannot be rejected. The RED algorithm is outlined in figure 1.

Note that if J_w is the task causing the maximum exceeding time overflow, the rejectable tasks that can remove the overload condition are only those tasks whose deadline is earlier than or equal to d_w. This means that the algorithm has to search only for tasks J_i, with $i \leq w$.

```
Algorithm RED_guarantee(J, J_a)

begin
    t = current_time();
    E = 0;              /* Maximum Exceeding Time */
    R_0 = 0;
    d_0 = t;
    J' = J ∪ {J_a};      /* Insert J_a in the ordered task list */
    k = position of J_a in the task set J';

    for each task J'_i such that i ≥ k  do {
            R_i = R_{i-1} + (d_i - d_{i-1}) - c_i;
            if (R_i + m_i < -E)  then
                    E = -(R_i + m_i);
    }

    if (E = 0)  then return ("Guaranteed");
    else {
             J* = set of least value tasks selected
                         by the rejection policy;
             if (J* is not empty)  then {
                     reject all task in J*;
                     return ("Guaranteed");
             }
             else return ("Not Guaranteed");
    }
end
```

Figure 1 RED Guarantee Algorithm.

4.1 Resource Reclaiming

One of the advantages of dynamic scheduling is that, whenever a task completes before its estimated worst case finishing time, the processor time saved is automatically[1] used for the execution of the other tasks. Such a dynamic allocation of processor time to the task set lowers the loading factor of the system. In order to take advantage of this fact in the guarantee algorithm, the loading function has to be computed not only at each task activation, but also at each task completion.

If a task cannot be guaranteed by the RED algorithm at its arrival time, there are chances that it could be guaranteed at later time, by using the execution time saved by other tasks. Scheduling tasks at an "opportune" time, rather than at arrival time has been proposed in [19] as a technique called *Well-Timed Scheduling*. However, this technique has been mainly used to reduce the scheduling overhead in highly loaded systems, rather than focusing on increasing the probability of a successful guarantee by utilizing reclaimed time. Also it did not treat holding a rejected task for possible re-guarantee at a later time.

In a more general framework, a task J_r rejected in an overload condition can still be guaranteed if the sum of the execution time saved by all tasks completing within the laxity of J_r is greater than or equal to the Maximum Exceeding Time found when J_r was rejected.

To take advantage of reclaimed time, we propose a more general framework for scheduling aperiodic hard tasks, as illustrated in figure 2.

Within this framework, if a task cannot be guaranteed by the system at its arrival time, it is not removed forever, but it is temporarly rejected in a queue of non guaranteed tasks, called *Reject Queue*, ordered by decreasing values, to give priority to the most important tasks. As soon as the running task completes its execution δ units of time before its worst case finishing time, the highest value task in the Reject Queue having positive laxity and causing a $Maximum\_Exceeded\_Time < \delta$ will be reinserted in the Ready Queue and scheduled by earliest deadline. All rejected tasks with negative laxity are removed from the system, and inserted in another queue, called *Miss Queue*, containing all late tasks; whereas all tasks that complete within their timing constraints are inserted in a queue of regularly terminated jobs, called *Term Queue*. The purpose of the Miss and Term Queues is to record the history

[1] If resources can be locked or multiprocessing is being used then resource reclaiming is not automatic. See [16] for a full discussion and solutions.

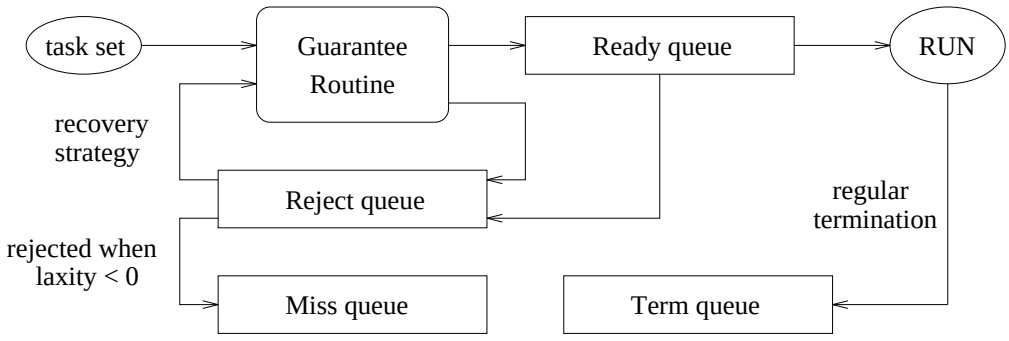


Figure 2 RED Scheduling Block Diagram.

of the system, which aids in debugging and understanding the operation of system.

5 PERFORMANCE EVALUATION

Simulations[2] were conducted to evaluate the performance of the RED algorithm with respect to several other baseline algorithms including EDF which is commonly used in dynamic hard real-time environments.

In all the experiments, the following scheduling algorithms have been compared:

- **EDF** - Earliest Deadline First algorithm, without any form of guarantee. As soon as a new task arrives in the system, it is inserted in the ready queue by its deadline and scheduled according to the EDF policy.
- **GED** - Guaranteed Earliest Deadline algorithm. When a new task arrives, a guarantee routine verifies whether the new task set is schedulable: if yes, the newly arrived task is inserted in the ready queue and scheduled by its deadline; if no, the newly arrived task is definitively rejected.
- **RED** - Robust Earliest Deadline algorithm with single task rejection. When a new task arrives, a guarantee routine verifies whether the new task set is feasibly schedulable: if yes, the newly arrived task is accepted;

[2]Due to space limitations, only the main results are shown here. See [4] for a full set of performance results.

if no, the system will reject the least value task, if any, such that the remaining set is schedulable, else the newly arriving task is rejected. Everytime a task completes its execution, a recovery routine tries to reaccept the greatest value task, among those rejected tasks whose laxity is positive.

- **MED** - Robust Earliest Deadline algorithm with multiple task rejection. The same as the RED algorithm, with the following difference: if the new task set is found unschedulable and the newly arrived task is critical, then the system may reject more than one lower value tasks to make the remaining task set schedulable.

The main performance metrics we used are:

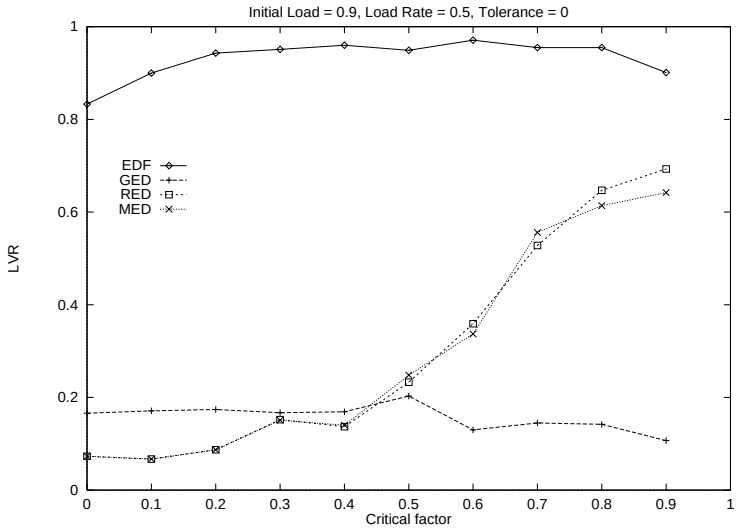
Loss Value Ratio (LVR) : ratio of the sum of the values of late HARD tasks to the total set value. Note that the value of CRITICAL tasks is not considered in this parameter, since CRITICAL tasks belong to another class.

Loss Critical Ratio (LCR) : ratio of the number of critical tasks that missed their deadline to the total number of critical tasks. This is used to show how the system operates in a region beyond which the dynamic guarantee had accounted for.

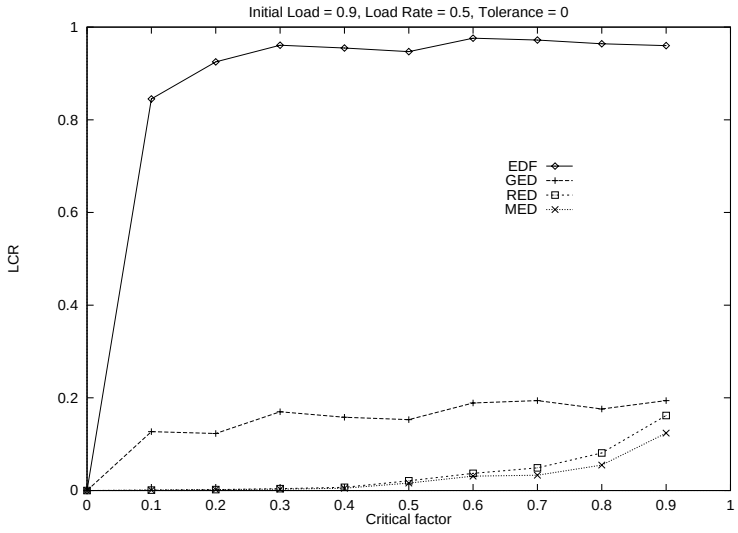
In all the graphs, average values are obtained over 50 runs. Standard deviations for these averages were computed and they were never greater than 3%. The value of non critical tasks is defined as a random variable uniformly distributed in the interval $[1, N]$. The value of critical tasks is defined as CRIT_VALUE, which is a value greater than N. The number of critical tasks in the set is controlled by a parameter called *critical factor*, which is the ratio of the number of critical tasks to the total number of tasks in the set. Since tasks can be rejected, the overload condition is maintained by generating an increasing load, computed as $\rho = \rho_0 + \alpha\rho_0 t$, where ρ_0 is the initial load, and α is a parameter called *load rate*, which controls the load growth at each task activation.

5.1 Experiment 1: Critical Factor

In the first experiment, we tested the capability of the algorithms of handling critical tasks in overload conditions. Figure 3a and 3b plot the Loss Value Ratio (LVR) and the Loss Critical Ratio (LCR) obtained for the four algorithms as a



(a)



(b)

Figure 3 LVR vs critical factor (a); LCR vs critical factor (b).

function of the critical factor. In this experiment, the initial workload was 0.9, with a growth factor $\alpha = 0.5$. For each task, the deadline tolerance was set to zero, and the computation time was set equal to its worst case estimation.

As shown in Figure 3, both LVR and LCR for EDF go over 0.9 as soon as the critical factor become greater than 0.2. This is clearly due to the *domino effect* caused by the heavy load. Although the guarantee routine used in the GED algorithm avoids such a domino effect typical of the EDF policy, it does not work as well as RED nor MED, since critical tasks are rejected as normal hard tasks, if they cause an overload. For example, in Figure 3b, when the percentage of critical tasks is 50% (critical factor = 0.5) we see a gain of about 15% for RED and MED over GED.

Another important implication from figure 3b is that RED and MED are able to provide almost no loss for critical tasks in overload conditions, until the number of critical tasks in the set is above 50% of the total number of tasks; the LCR is practically zero for both algorithms. Above this percentage, however, some loss is experienced and, by around 80% of the load being critical tasks, we start to see the multiple task rejection policy used in MED begin to be slightly more effective than RED.

To understand the behavior of RED and MED depicted in figure 3a, remember that the LVR is computed from the value of HARD tasks only, since CRITICAL tasks belong to a different class. Therefore, as the critical factor increases, RED and MED have to reject more HARD tasks to keep the LCR value low, whereas GED does not make any dinstiction between HARD and CRITICAL tasks.

An important result shown in this experiment is that, when the number of critical tasks is not high, it is not worthwhile to use complicated rejection strategies. In this cases, the simple ($O(n)$) strategy used in RED, in which the least value task is rejected, performs as well as more sophisticated and time consuming policies.

Notice that in all experiments presented in this paper, no assumption has been made on the minimum interarrival time of critical tasks. Therefore, even when the percentage of critical tasks is low, there is always a (low) probability that a critical task can be rejected with the MED algorithm, if it arrives just after another critical task and the deadlines of both are close. Note that this condition is an overload.

5.2 Experiment 2: Load Rate

In this experiment, we tested the performance of RED as a function of the load. Since the rejecting policy always maintains the load below the limit of one, we tested the system by increasing the load rate α. In this experiment, the number of critical tasks was 20 percent of the total number of tasks, the initial workload was 0.9, and the load rate α was varied from 0 to 0.75, with a step of 0.05. For each task, the deadline tolerance was set to zero, and the computation time was set equal to its worst case estimation.

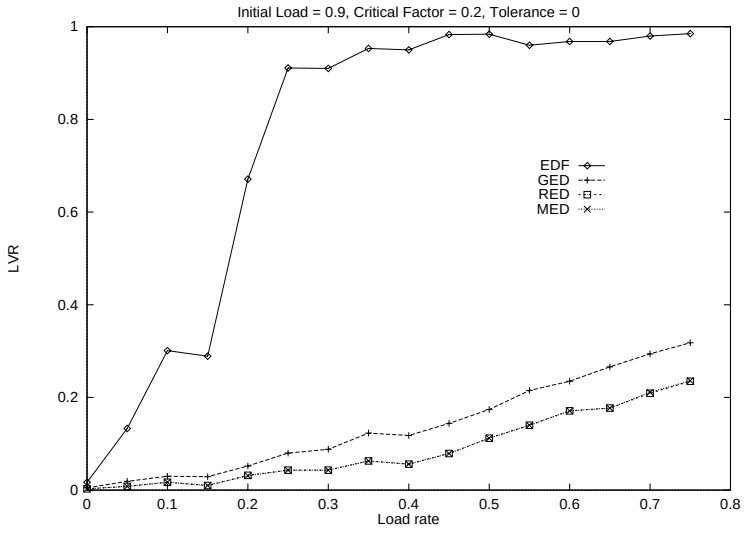
Figure 4a plots the loss value ratio (LVR) obtained with the four algorithms as a function of the load rate α, and figure 4b plots the loss critical ratio (LCR). When $\alpha = 0$ the system workload is maintained on the average around its initial value $\rho = 0.9$, therefore the loss value is negligible for all algorithms. By increasing α, the load increases as new tasks arrive in the system.

As shown in figure 4, the EDF algorithm without guarantee was not capable of handling overloads, so that the loss in value increased rapidly towards its maximum (equal to the total set value). At this level, only the first tasks were able to finish in time, while all other tasks missed their deadlines. Again, RED and MED did not show any significant difference between themselves during this test, but a very significant improvement was achieved over EDF and GED. For example, using a load rate $\alpha = 0.5$, which causes a heavy load, the LVR is 0.98 with EDF, 0.17 with GED, and only 0.11 for both RED and MED. Notice that the loss value obtained running RED and MED is entirely due to hard tasks, since from figure 4b we see that the number of critical tasks missing their deadlines is practically zero for RED and MED.

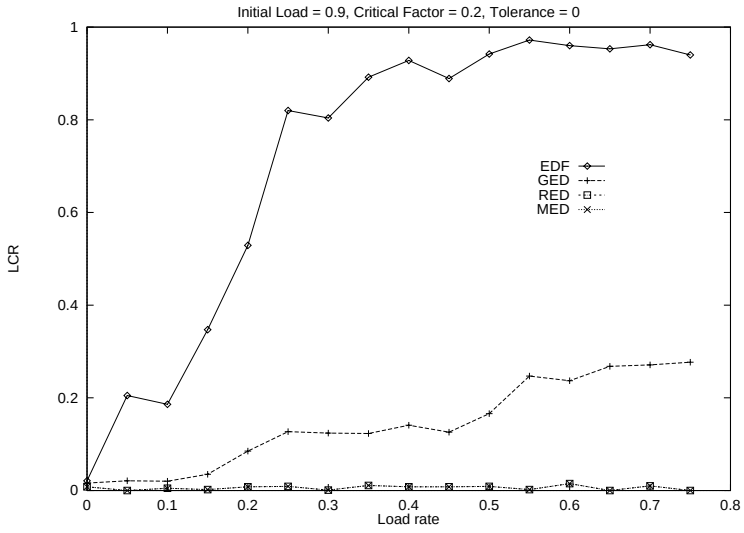
5.3 Other experiments

Other experiments have been conducted to show the effectiveness of the recovery strategy used in the RED (and MED) algorithm. When a task completes its execution before its estimated finishing time, the recovery strategy tries to reaccept rejected tasks (based on their values) until they have positive laxity.

Although EDF does not use any recovery strategy, EDF performs better than GED for high values of *dw*. This is due to the fact that for high computation time errors, the actual workload of the system is much less than the one estimated at the arrival time, so EDF is able to execute more tasks. On the other hand, GED cannot take advantage of saved time, since it rejects tasks at arrival



(a)



(b)

Figure 4 LVR vs load rate (a); LCR vs load rate (b).

time based on the estimated load. RED and MED also reject tasks based on the current estimated workload, but the recovery strategy makes use of saved execution time for reaccepting rejected tasks. Since tasks are reaccepted based on their value, RED and MED perform as well as EDF for large computation time errors.

Another experiment tested the effect of having a deadline tolerance. Remember that a task with deadline d_i and tolerance m_i is not treated as a task with deadline $(d_i + m_i)$: deadline is used for scheduling, and tolerance is used for guarantee. This means that the algorithm always tries to schedule all tasks to meet their deadlines; only in overload conditions there is a chance that some task may exceed its deadline. In order to compare the four algorithms in a consistent fashion, the concept of tolerance has been used also for EDF and GED.

LCR values obtained with EDF were more than an order of magnitude bigger than those obtained with the other algorithms. For tolerance values greater than 10, we observed that RED and MED caused no late critical tasks. We also noted that GED performs better than the other three algorithms in terms of LVR. This is due to the fact that RED and MED reject more hard tasks than GED, to keep the LCR as low as possible.

The performance results clearly show the poor performance of EDF scheduling in overload. This fact has been often stated, but rarely shown with performance data. The results also show the clear advantage of on-line planning (dynamic guarantee) algorithms under such conditions. More importantly, the results also show that the new RED algorithm is significantly better than even the basic guarantee approach because it uses the entire load profile, deadline tolerance, and resource reclaiming. The better performance occurs across different loads, deadline distributions, arrival rates, tolerance levels, and errors on the estimates of worst case execution time. RED also is significantly better than EDF and better than GED in handling unexpected overloads of critical tasks; an important property for safety critical systems. This implies that RED keeps the system safe, longer, when unanticipated overloads occur. We also see that the very simple rejection policy for RED suffices in almost all conditions, and that MED only improves RED in a small part of the parameter space.

6 RELATED WORK

Earliest deadline first (EDF) scheduling has received much attention. It has been formally analyzed proving that it is an optimal algorithm for preemptive, independent tasks when there is no overload [6]. It is also known that EDF can perform poorly in overload conditions.

In 1984, Ramamritham and Stankovic [13] used EDF to dynamically guarantee incoming work via on-line planning and if a newly arriving task could not be guaranteed the task was either dropped or distributed scheduling was attempted. All tasks had the same value. The dynamic guarantee performed in this paper had the effect of avoiding the catastrophic effects of overload on EDF.

In 1986, Locke [11] developed an algorithm which makes a best effort at scheduling tasks based on earliest deadline with a rejection policy based on removing tasks with the minimum value density. He also suggests that removed tasks remain in the system until their deadline has passed. The algorithm computes the variance of the total slack time in order to find the probability that the available slack time is less than zero. The calculated probability is used to detect a system overload. If it is less than the user prespecified threshold, the algorithm removes the tasks in increasing value density order. Consequently, detection of overload is performed in a statistical manner rather than based on an exact profile as in our work. This gives us the ability to perform specific analysis on the load rather than a probabilistic analysis. While many features of our algorithm are similar to Locke's algorithm, we extend his work, in a significant manner, by the careful and exact analysis of overload, support n classes, provide a minimum level of guarantee even in overload, allow deadline tolerance, more formally address resource reclaiming, and provide performance data for the impact of re-guarantees. We also formally prove all our main results.

In Biyabani et. al. [2] the previous work of Ramamritham and Stankovic was extended to tasks with different values, and various policies were studied to decide which tasks should be dropped when a newly arriving task could not be guaranteed. This work used values of tasks such as in Locke's work, but used an exact characterization of the first overload point rather than a probabilistic estimate that overload might occur. However, Biyabani's work did not fully analyze the overload characteristics, build a profile, allow deadline tolerance, nor integrate with resource reclaiming and re-guarantee, as we do in this paper.

Haritsa, Livny and Carey [8] present the use of a feedback controlled EDF algorithm for use in real-time database systems. The purpose of their work is to get good average performance for transactions even in overload. Since they are working in a database environment their assumptions are quite different than ours, e.g., they assume no knowledge of transaction characteristics, they consider firm deadlines not hard deadlines, there is no guarantee, and no detailed analysis of overload. On the other hand, for their environment, they produce a very nice result, a robust EDF algorithm. The robust EDF algorithm we present is very different because of the different application areas studied, and because we also include additional features not found in [8].

In real-time Mach [18] tasks were ordered by EDF and overload was predicted using a statistical guess. If overload was predicted, tasks with least value were dropped. Jeffay, Stanat and Martel [9] studied EDF scheduling for non-preemptive tasks, rather than the preemptive model used here, but did not address overload.

Other general work on overload in real-time systems has also been done. For example, Sha [15] shows that the rate monotonic algorithm has poor properties in overload. Thambidurai and Trivedi [17] study transient overloads in fault tolerant real-time systems, building and analyzing a stochastic model for such a system. However, they provide no details on the scheduling algorithm itself. Schwan and Zhou [14] do on-line guarantees based on keeping a slot list and searching for free time intervals between slots. Once schedulability is determined in this fashion, tasks are actually dispatched using EDF. If a new task cannot be guaranteed, it is discarded. Zlokapa, Stankovic and Ramamritham [19] propose an approach called well-time scheduling which focuses on reducing the guarantee overhead in heavily loaded systems by delaying the guarantee. Various properties of the approach are developed via queueing theoretic arguments, and the results are a multi-level queue (based on an analytical derivation), similar to that found in [8] (based on simulation).

Finally, Gehani and Ramamritham [7] propose programming language features to allow specification of deadline and a deadline slop factor (similar to our deadline tolerance), but propose no algorithms for supporting this feature.

7 CONCLUSIONS

We have developed a robust earliest deadline scheduling algorithm for hard real-time environments with preemptive tasks, multiple classes of tasks, and tasks with deadline tolerances. We have formally proven several properties of the approach, developed an efficient on-line mechanism to detect overloads, and provided a complete load profile which can be usefully exploited in various ways. A performance study shows the excellent performance of the algorithm in both normal and overload conditions. Precedence constraints can be handled by *a priori* converting precedence constraints to deadlines. A future extension we are working on is to include resource sharing among tasks.

REFERENCES

[1] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the Competitiveness of On-Line Real-Time Task Scheduling," *Proceedings of Real-Time Systems Symposium*, December 1991.

[2] S. Biyabani, J. Stankovic, and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," *Proceedings of the Real-Time Systems Symposium*, December 1988.

[3] J. Blazewicz, "Scheduling dependent tasks with different arrival times to meet deadlines" In E. Gelenbe, H. Beilner (eds), Modelling and Performance Evaluation of Computer Systems, Amsterdam, North-Holland, pp. 57-65, 1976.

[4] G. C. Buttazzo and J. A. Stankovic, "RED: Robust Earliest Deadline Scheduling," Technical Report, Dept. of Computer Science, University of Massachusetts, Amherst, TR-93-25, 1993.

[5] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic Scheduling of Real-Time Tasks under Precedence Constraints," *The Journal of Real-Time Systems*, 2, pp. 181-194, 1990.

[6] M. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Proceedings of the IFIP Congress*, 1974.

[7] N. Gehani and K. Ramamritham, "Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems," *Real-Time Systems*, 3, pp. 377-405, 1991.

[8] J. R. Haritsa, M. Livny, and M. J. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proceedings of Real-Time Systems Symposium*, December 1991.

[9] K. Jeffay, D. Stanat, and C. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proceedings of Real-Time Systems Symposium*, December 1991.

[10] C. L. Liu and J. Leyland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, 20, pp. 46-61, 1973.

[11] C. D. Locke, "Best Effort Decision Making For Real-Time Scheduling," PhD Thesis, Computer Science Dept., CMU, 1986.

[12] A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment," PhD Dissertation, MIT, May 1983.

[13] K. Ramamritham and J. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software*, pp. 65-75, July 1984.

[14] K. Schwan and H. Zhou, "Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads," *IEEE Transactions on Software Engineering*, Vol. 18, No. 8, pp. 736-748, August 1992.

[15] L. Sha, J. Lehoczky, and R. Rajkumar, "Solutions for Some Practical problems in Prioritized Preemptive Scheduling," *Proceedings of Real-Time Systems Symposium*, December 1986.

[16] C. Shen, K. Ramamritham, and J. Stankovic, "Resource Reclaiming in Real-Time," *Proceedings of Real-Time Systems Symposium*, December 1990.

[17] P. Thambidurai, and K. S. Trivedi, "Transient Overloads in Fault-Tolerant Real-Time Systems," *Proceedings of Real-Time Systems Symposium*, December 1989.

[18] H. Tokuda, J. Wendorf, and H. Wang, "Implementation of a Time-Driven Scheduler for Real-Time Operating Systems," *Proceedings of the Real-Time Systems Symposium*, December 1987.

[19] G. Zlokapa, J. A. Stankovic, and K. Ramamritham, "Well-Timed Scheduling: A framework for Dynamic Real-Time Scheduling," Submitted to *IEEE Transactions on Parallel and Distributed Systems*, 1991.