An Incremental Approach to Scheduling during Overloads in Real-Time Systems*

Pedro Mejía-Alvarez†
CINVESTAV-IPN. Sección de Computación
Av. I.P.N. 2508, Zacatenco.
México, DF. 07300
pmejia@computacion.cs.cinvestav.mx

Rami Melhem, Daniel Mossé
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260
melhem,mosse@cs.pitt.edu

Abstract

In this paper we propose a novel scheduling framework for a real-time environment that experiences dynamic changes. This framework is capable of adjusting the system workload in incremental steps under overloaded conditions such that the most critical tasks in the system are always scheduled and the total value of the system is maximized. Each task has an assigned criticality value and consists of two parts, a mandatory part and an optional part. A timely answer is available after the mandatory part completes execution and its value may be improved by executing the entire optional part. Optional parts can be discarded in overloaded conditions.

The process of selecting optional parts to discard while maximizing the value of the system requires the exploration of a potentially large number of combinations. Since this process is too time consuming to be computed on-line, an approximate algorithm is executed incrementally whenever the processor would otherwise be idle, progressively refining the quality of the solution. This criteria allows the scheduler to handle overloads with low cost while maximizing the use of the available resources and without jeopardizing the temporal constraints of the most critical tasks in the system. Simulation results show that few stages of the algorithm need to be executed for achieving a performance with near-optimal results.

1 Introduction

The use of complex and dynamic real-time systems is nowadays becoming common for the management and control of a variety of applications such as manufacturing, industrial automation systems, space or avionics and telecommunications systems. In a critical real-time system, each task must complete and produce correct results by the specified deadline. Failure to conform with any timing constraint is considered a catastrophic failure. In order to guarantee that the timing constraints will be satisfied, it is necessary that the resource requirements for all tasks in the system be known and that resources be available in a timely manner. Therefore, the resources must be reserved for worst-case execution time of tasks to provide absolute guarantees.

Traditionally, the resource scheduling problem for real-time tasks is to generate a feasible schedule or to verify if a given scheduling policy can meet the timing requirements of a specific set of tasks. In practice, however, real-time environments experience frequent changes in workloads, caused by new task arrivals or tasks that leave the system after finishing their execution. The problem with accepting new tasks in the system is that they may result in an overload and cause some of the tasks already in the system to miss their deadlines. Under such overload conditions, we may augment the available resources or reject some tasks (or both).

In this paper we study the problem of scheduling dynamic tasks in an overloaded single processor environment, where new tasks arrive or leave the system at arbitrary instances of time. A framework is proposed for adjusting the system workload incrementally by relating the criticality value[2, 3] of the tasks to the resource allocation problem. The identification of feasible options that maximize an optimality criteria (expressed as the total value of the system) requires the exploration of a potentially large combinatorial space of solutions. Our approach to solve this problem is based on an on-line Incremental Server (INCA), which searches feasible solutions by executing a sequence of approximate algorithms. At each approximate algorithm execution, the load is adjusted and the quality of the solution is refined. The minimum number of approximate algorithms executed produces a feasible but suboptimal solution that can be incrementally improved if more approximate algorithms can be executed. Functions with this property are called *incrementally* precise functions or incremental processes.

We consider real-time tasks that consist of mandatory parts and optional parts for refining the result of the mandatory parts. Systems exhibiting this behavior include (1) multimedia systems that receive, enhance or transmit audio, video or images

*This work has been supported by the Defense Advanced Research Projects Agency through the FORTS project (Contract DABT63-96-C-0044)

†Work done while this author was visiting the Information Sciences and Telecommunications Department, University of Pittsburgh

0-7695-0900-2/00 $10.00 © 2000 IEEE

and process this information for specific intervals of time, (2) process control systems with sensors and actuators that are activated by changing environmental conditions and (3) real-time database query processing systems. For systems such as these, our approach is to produce approximate solutions that can be progressively refined, when the exact solutions cannot be produced in time, due to overloaded conditions.

The remainder of this paper is organized as follows. In Section 2 related models and previous work are reviewed. In Section 3, the task model used in this paper is defined. In Section 4, the overload scheduling problem is formulated. In Section 5, the INCA server is described and in Section 6 we analyze the merit of the incremental execution of the INCA server, and compare its performance against a non incremental server. In Section 7, simulation results are presented to show the performance of the INCA Server and to give insight into its effectiveness in handling overload conditions. Finally, Section 8 presents concluding remarks.

2 Related Work

In a dynamic real-time environment, even when the system is properly designed and sized, a transient overload can occur for different reasons, such as changes in the environment, simultaneous arrivals of asynchronous events, faults of peripheral devices, or system exceptions[5]. The worst consequence that may happen is that some critical tasks in the system miss their deadlines, jeopardizing the correct/safe behavior of the system. As all systems have finite resources, their ability to execute a set of periodic and aperiodic tasks while meeting the temporal requirements is limited. Clearly, overload conditions arise if the system has to process more new tasks than the available set of resources can handle.

In the real-time literature several scheduling algorithms have been proposed to deal with overloaded systems. The development of the Best-Effort algorithm [14] introduced a rejection policy for overloaded systems based on removing tasks with the minimum value density. The Best-Effort approach basically behaves as the Earliest Deadline First [12] when the system is underloaded and chooses the subset of tasks that maximize the value of the computation per unit of time (value density) when the system is overloaded. The Alpha effort [9] introduced the concept of time-valued functions, which associate a value according to the task finishing time. The function presents a drop in value after the deadline has passed, and beyond a certain time the value drops to zero.

The problem of selecting tasks for rejection in an overloaded system is also considered in [8], where random criticality values are assigned to tasks. An approximate algorithm incorporates *simulated annealing* to deal with the problem of selecting a feasible solution within the large combinatorial space of permutations. The RED (Robust Earliest Deadline) algorithm [4] deals with aperiodic tasks in overloaded environments, combining criticality-based scheduling, deadline tolerance (the amount of time by which a task is permitted to be late) and resource reclaiming. It is able to predict not only deadline misses but also the size of the overload, its duration, and its impact on the system. However, the strategy for handling the overload is to reject the least-valued task. Other approaches for handling overload focuses on providing a less stringent guarantees for temporal constraints. In [10] some instances of a task are allowed to be skipped entirely. The *skip factor* determines how often instances of a given task may be left unexecuted. A best effort approach is introduced in [7], aiming at meeting k deadlines out of n instances of a given task. However, it is assumed that the value of the tasks in the system is proportional to their computation time, provided that they complete by their deadlines.

Many of these techniques (e.g., [10, 7]) assume that a task's output is of no value if it is not executed completely. In contrast, in the Imprecise Computation(IC) model the task's output has some value even if a partial or approximate result is produced [1, 13]. In the IC model, every real-time task is composed of a mandatory part and an optional part. A timely answer is available after the mandatory part completes execution; moreover, the longer the optional part executes, the higher the value of the task (i.e., the higher the quality of the result). However, the IC model uses an error function as a metric to evaluate the performance of the system. In [13] an error function is defined to be inversely proportional to the total amount of time that the optional parts execute. An optimal schedule corresponds to the one where the total error of the system is minimized. In the IC model the shape of the error functions and policies for scheduling optional parts are crucial in maximizing the performance of the system.

From previous work we have learned that: (a) many scheduling algorithms have been developed for overloaded conditions, but few research work studied in practice how far from optimal is the performance of their algorithms and their complexity (reference [11] provided a measure for the performance of their D-over algorithm using a metric called *competitive factor*; this metric denotes the cumulative value of the system compared with the optimal value obtained by a *clairvoyant scheduler* that knows the entire task set *a priori*);(b) In most developed algorithms, the criteria for rejection in overloaded conditions is to select the lesser-valued tasks, a strategy that clearly yields low cost solutions but may lead to a situation with underutilized resources and a resulting system with poor performance; and (c) the time-value function of [9] or the error functions of the IC model [13] are difficult to obtain, and performance may be degraded if the system designers are not familiar with the functions that represent the applications at hand. Although our model is similar to the IC model, we do not use error functions but performance metrics that are largely available, such as utilization and criticality[2, 3].

3 Task Model

In our framework we consider periodic preemptive tasks running on one processor. Tasks are independent and have no

precedence constraints. Each task τ_i arrives in the system at time a_i. The *life-time* of each task τ_i consists of a fixed number of instances r_i. After the execution of r_i instances, the task leaves the system[1]. The time interval between the arrival of the first instances of two consecutive tasks τ_x and τ_y is defined as $l_{xy} = a_y - a_x$. We assume that the tasks characteristics (e.g., computation time, period, deadline and criticality) are known at arrival time. Each task τ_i is decomposed into a mandatory part M_i followed by an optional part P_i. The use of the imprecise computation model is not restrictive in the sense that each task may have only an optional part and no mandatory part. In this model, T_i is the period and C_i is the worst case computation time of task τ_i. Each execution time C_i consists of a mandatory part of length m_i and an optional part of length p_i (i.e., $C_i = m_i + p_i$). The mandatory part M_i must execute to completion in order to produce an acceptable and usable result. The optional part P_i can execute only after the completion of the mandatory part M_i. However, a partially executed optional part or an optional part that misses its deadline is of no value to the system (0/1 constraint). The task τ_i meets its deadline if its mandatory part completes by its deadline. It is assumed that the set of mandatory parts can never cause an overload in the system. Each task has an associated criticality value v_i, which denotes its importance within the system[2]. The Earliest Deadline First[12] priority assignment scheme will be considered.

4 Formulation of the Problem

In overloaded conditions, the scheduler should be able to guarantee the timing constraints of all mandatory parts at every periodic task invocation and to select optional parts for exclusion from the schedule while maximizing the performance of the system. If the criticality of each task is proportional to its computation time, the decision of excluding optional parts must be based only on maximizing the usage of the resources (e.g., CPU time). In the more general case, where criticality and computation time are not directly related, we would like to exclude the less critical optional parts and maximize the total value of the system. Therefore, the problem can be formulated as follows. If a new task τ_i that arrives in the system at time a_i causes an overload, the problem is (a) to determine whether or not τ_i can be accepted in the system without interfering with the deadlines of the mandatory parts of any task already in the system, (b) if accepted, at what time τ_i should be dispatched?, (c) what optional parts (if any) should be excluded such that an optimality criteria is satisfied and (d) while searching for a solution, how can we maximize the usage of the resources and the performance of the system with a reasonable low cost?

Each task in the system accrues an accumulated value upon executing a number of optional parts during its life-time. Our objective is to maximize the accumulated value obtained after scheduling the set of optional parts for the complete duration of the schedule. The accumulated value will be evaluated in terms of utilization or criticality as follows. Let us define $CU(I)$ and $CV(I)$ as the cumulative utilization and cumulative criticality potentially achieved by the set of optional parts that execute during the interval of time I.

The cumulative utilization achieved is computed by,

$$CU(I) = \sum_{i=1}^{n} I * \frac{p_i}{T_i} \quad (1)$$

The cumulative criticality achieved is computed by,

$$CV(I) = \sum_{i=1}^{n} I * \frac{v_i}{T_i} \quad (2)$$

For the interval of time between two consecutive arrivals a_x and a_y the accumulated value can be formulated as $CU(l_{xy})$ and $CV(l_{xy})$ for utilization and criticality respectively. At a_x, the goal is to select the optional parts that maximize $CU(l_{xy})$ or $CV(l_{xy})$. This selection requires the searching of a usually large search space.

4.1 Definition of the Search Space

Consider n tasks, $\tau_1, ..., \tau_n$ such that $\sum_{i=1}^{n} \frac{m_i}{T_i} \leq 1$ and $\sum_{i=1}^{n} \frac{m_i+p_i}{T_i} > 1$. That is, all mandatory parts can be accepted for execution but not all optional parts can be accepted for execution.

Let S be the search space containing all combinations (both feasible and non-feasible) of optional tasks. More specifically, a search space is defined as $S = \cup_{k=0}^{n} S_k$ where $S_k = \{(x_1, \ldots, x_n); \sum_{i=1}^{n} x_i = k\}$ where $x_i = 0$ means that the optional part of τ_i is discarded and $x_i = 1$ means that the optional part of τ_i is chosen for execution. Note that for any $k \in \{0, .., n\}$, S_k is a set of elements containing all (feasible and non-feasible) combinations resulting from including k optional parts for execution and that $|S_k| = \frac{n!}{k!(n-k)!}$. Any element in S_k includes for execution exactly k optional parts.

Set	Search Space
S_4	$\{1,1,1,1\}$
S_3	$\{1,1,1,0\}$ $\{1,1,0,1\}$ $\{1,0,1,1\}$ $\{0,1,1,1\}$
S_2	$\{1,1,0,0\}$ $\{1,0,1,0\}$ $\{1,0,0,1\}$ $\{0,1,1,0\}$ $\{0,1,0,1\}$ $\{0,0,1,1\}$
S_1	$\{1,0,0,0\}$ $\{0,1,0,0\}$ $\{0,0,1,0\}$ $\{0,0,0,1\}$
S_0	$\{0,0,0,0\}$

Table 1. Search space for four tasks.

The structure of the search space S is shown in Table 1 through an example with 4 tasks. For example, $\{1,1,0,0\}$ is the element in which p_1 and p_2 are included for execution and p_3 and p_4 are discarded.

4.2 Definition of the Objective Functions

Each element in the search space will be evaluated in terms of utilization or criticality, using the objective functions $\mu(s)$ and $\gamma(s)$ respectively, where $s = \{x_1, ..., x_n\} \in S$. The objective functions are defined as follows.

[1] We assume that some tasks that leave the system may return at a later time.

[2] Methods to derive this criticality values are proposed in [2, 3, 14].

- $\mu(s)$: In this function we add the utilization of the set of optional parts in an element $s \in S$ to the total utilization of all mandatory parts.

$$\mu(s) = \sum_{i=1}^{n} \frac{m_i}{T_i} + \sum_{j=1}^{n} x_j \frac{p_j}{T_j} \quad (3)$$

  $\mu(s)$ denotes the utilization of the system, after choosing some optional parts for execution. For example, for $s = \{0,1,1,0\}, \mu(s) = \sum_{i=1}^{4}(\frac{m_i}{T_i}) + \frac{p_2}{T_2} + \frac{p_3}{T_3}$.

- $\gamma(s)$: In this function we compute the criticality per period achieved after including for execution a set of optional parts in an element $s \in S$, recalling that v_i is the criticality of task τ_i.

$$\gamma(s) = \sum_{i=1}^{n} x_i(\frac{v_i}{T_i}) \quad (4)$$

  For example, for $s = \{0,1,1,0\}, \gamma(s) = \frac{v_2}{T_2} + \frac{v_3}{T_3}$

Note that if the tasks $\{\tau_1, ..., \tau_n\}$ are to execute during an interval I, the choice of s that maximizes $\mu(s)$ and $\gamma(s)$, also maximizes CU(I) and CV(I) respectively.

4.3 Feasibility Test

To evaluate the feasibility of each element in the search space we apply an utilization-based test (UBT). The utilization-based test has been chosen because of its simplicity and because it can be used for scheduling policies such as EDF.

For each element $s = \{x_1, ..., x_n\}$ of the search space S the utilization-based test is defined by,

$$UBT(s) = \begin{cases} true & if\ \mu(s) \leq 1 \\ false & \text{otherwise} \end{cases}$$

Note that, when choosing a feasible solution, the utilization of the optional parts ($U_p = \sum_{i=1}^{n} x_i \frac{p_i}{T_i}$) must satisfy: $U_p \leq 1.0 - U_m$, where $U_m = \sum_{i=1}^{n} \frac{m_i}{T_i}$. Also, any single optional part with utilization $\frac{p_i}{T_i}$ greater than $1.0 - U_m$ can be discarded without any further test.

4.4 The Optimization Problems

Our first optimization problem is related to shedding a number of optional parts that maximizes the utilization of the system. This objective favors a solution in which the utilization of the workload is maximized without considering the number of optional parts to be shed. Our second optimization problem assumes that different criticality values are associated with optional parts, therefore we are interested in maximizing the total value obtained after a number of optional parts are shed.

The optimization problems are formally described as follows,

- **Maximize the utilization.** The aim of this objective is to find a feasible element $s \in S$ such that the utilization in the system is maximized. That is

$$\begin{array}{ll} \text{maximize} & \mu(s) \\ \text{subject to} & \text{UBT(s)} \end{array}$$

  Let U^{max} be the value of $\mu(s)$ obtained by solving this optimization problem.

- **Maximize the value.** Maximizing the value requires to find a feasible element $s \in S$ such that $\gamma(s)$ is maximized. That is

$$\begin{array}{ll} \text{maximize} & \gamma(s) \\ \text{subject to} & \text{UBT(s)} \end{array}$$

  Let V^{max} be the value of $\gamma(s)$ obtained by solving this optimization problem.

The optimization problems consist of maximizing the value of the system at the instant of time at which a new arrival causes an overload in the system. By achieving the optimality criteria, whenever a new task arrives or departs from the system, we intend to maximize the accumulated value (CU(I) or CV(I)) obtained after scheduling the entire set of tasks for the complete duration of the schedule.

5 The Incremental Scheduling Server: INCA

The incremental scheduling server is an extension of the earliest deadline first scheduling algorithm (EDF[12]). In response to transient overload requests, the INCA Server adjusts the load of the system by executing a sequence of approximate algorithms, $AP(0), ..., AP(n)$ to determine which optional parts to shed in order to satisfy our optimality criteria. The algorithms are such that $AP(i)$ may obtain a solution closer to optimal than $AP(i-1)$ but with longer execution time. The INCA Server is activated whenever the feasibility test (UBT) detects an overload caused by the arrival of a new task in the system. The INCA Server first executes the approximate algorithm AP(0) to eliminate the overload. The solution provided by AP(0) allows the scheduler to disable temporarily the execution of some optional parts[3], while providing a low cost non-optimal solution. The slack-time[4] introduced in the system by the removal of the overload is used by the scheduler to execute approximate algorithms AP(k) (for $k = 1, ..., n$), progressively refining the quality of the solution.

If during the execution of the INCA Server a new task arrives or a task leaves the system, it will re-start its execution, taking into account the modified load of the system.

[3] Note that after the execution of each approximate algorithm some optional parts may be disabled temporarily, but not discarded. The reason for this is that at each approximate algorithm we may find different solutions involving different optional parts to execute.

[4] Slack-time is defined as the time at which the processor is not executing any task.

```
1: INCA Server:
2: input: a set of tasks τ1, ..., τn, including the newly arrived task τm
3: If Σ mi/Ti > 1 then reject the new task τm; exit;
4: Execute AP(0); (remove the overload)
5: Compute the start time of the new task τm;
6: Schedule the new task at its start time;
7: k = 1;
8: while (there is slack in the schedule) do
9: begin
10:     Execute AP(k); (during slack time)
11:     If the result from AP(k) is better
            than the result from AP(k-1) then
12:         Enable the optional parts selected by AP(k);
13:     else exit;
14:     k = k + 1;
15: end;
```

Figure 1. Incremental Server (INCA)

The INCA Server stops its execution when (a) there is no more slack in the schedule to execute additional AP(k) algorithms, or (b) the result of AP(k) is not better than the result of AP(k-1). The INCA Server is described in Figure 1.

5.1 Methodology for Handling Overload

In this section, a methodology for handling overload conditions is introduced.

1. Activating the Incremental Server. The INCA server is activated if a new task, τ_m, arrives in the system and causes an overload. Our feasibility test (UBT) detects this condition. After detecting the overload, AP(0) is executed[5] and some optional parts are chosen to be discarded for removing the overload. The INCA server is also activated when a task leaves the system. In this case, the approximate algorithms AP(k) (for k=0,...,n) are executed incrementally to satisfy the optimality criteria for the new set of optional parts in the system.

2. Scheduling the new task. After removing the overload from the system, the newly arrived task can be scheduled. However, regardless of the priority of the newly arrived task, if the task is accepted it may not be scheduled at its arrival time because it may cause some missing deadlines, even after executing AP(0). This is because, at the instant of the new arrival, we may choose optional parts that (a) already finished their current execution, or (b) have been preempted while executing their optional parts. The resulting utilization cannot be immediately subtracted from the total processor load because the discarded optional parts could already have delayed the execution of other tasks. As a consequence, to keep the feasibility test consistent, the utilization of a discarded optional part can be subtracted from the total load only at the end of its current period. Thus, the new task should wait until the end of the longest period of all preempted tasks [6]. Clearly, the tasks that have not already started the execution of their optional parts allow the utilization to be subtracted immediately.

3. Execution of AP(1), ..., AP(n). After removing the overload through $AP(0)$, an increase in the slack available in the system is expected. $AP(1)$ is then executed on the slack existing in the system after the execution of AP(0). Analogously, $AP(k)$ (for $k = 2, ..., n$) is executed on the slack existing in the system after the execution of the previous $AP(k-1)$. The INCA Server executes the approximate algorithms AP(1), ..., AP(n) incrementally. After executing each $AP(k)$, the workload of the system is adjusted by enabling and disabling some optional parts. Since the INCA Server executes on the slack available in the system, it will execute as many approximate algorithms as possible.

Algorithm AP(k) may yield better solutions than AP(k-1) but at the cost of higher execution times. However, the execution of each AP(k) may increase the utilization and thus decrease the amount of available slack. This can eventually exhaust all the available slack in the system. If this condition occurs, the execution of AP(k+1) will not be possible, therefore, the INCA Server will stop its execution.

During the execution of the some AP(k) algorithm, a new task may arrive in the system or a task may leave the system. If this occurs, the INCA Server will re-start its execution, taking into account the modified load of the system. If the incremental server is invoked when a task leaves the system, the instructions in lines 3, 4, 5, 6 (from Figure 1) are not executed.

4. Stopping the execution of the server. The conditions for finishing the execution of the server are a). there is no more slack in the schedule to execute some AP(k) algorithm, or b). the result of AP(k) is not better than the result of AP(k-1). Also, after AP(n) is executed the server finish its execution.

5.2 The Approximate Algorithm: $AP(k)$

In this section, we describe the approximate algorithm used by the INCA Server. The approximate algorithm makes use of a greedy-type procedure[16] which finds a heuristic solution by selecting for execution optional parts in order of decreasing utilization $\frac{p_i}{T_i}$ if the objective function is $\mu(s)$, or $\frac{v_i}{p_i/T_i}$ if the objective function is $\gamma(s)$[16]. The algorithm AP(k) considers all possible subsets in the search space with at least k optional parts chosen for execution. It first chooses for inclusion in the schedule a subset of k optional parts, and if this subset does not satisfy our feasibility condition (UBT) it is discarded and a new subset with k optional parts is selected. If the subset passes the UBT, the remaining optional parts are considered for selection in decreasing order of $\frac{p_i}{T_i}$ or $\frac{v_i}{p_i/T_i}$, while the UBT is satisfied. The best solution obtained by examining all subsets of k optional parts is the solution generated by this algorithm.

[5] We assume that the execution times of AP(0) and UBT are negligible.

[6] Computing analytically the best time at which it is possible to accept the new task may involve some additional run-time overhead[6]. Therefore we have decided to schedule the new task at the end of the last period of the instances running in the system when the new task arrives.

```
1: Algorithm AP(k):
2: input: F: Objective Function (See section 4.2)
3:        τ1, ..., τn ordered according to pi/Ti or vi/(pi/Ti)
4: output: X^k: set of optional parts chosen for execution.
5:        Z^k: the optimized value computed for F.
6: Z^k = 0;
7: for each M ⊂ {1, ..., n}
        such that |M| = k and UBT(M) = true
8:    call SEQ; (to compute z and X)
9:    if z + F(M) > Z^k then
10:   begin
11:       Z^k = z + F(M);
12:       X^k = X ∪ M;
13:   end
14: procedure SEQ:
15: input: M
16: output: X: set of optional parts chosen for execution.
17:   z: the value of F for the subset X.
18: z = 0;      X = ∅;
19: for i=1 to n do
20:   if i ∉ M and UBT(M ∪ X ∪ {i}) = true then
21:   begin
22:     z = z + F({i});
23:     X = X ∪ {i};
24:   end
```

Figure 2. Approximate Algorithm (AP)

The algorithm is described in Figure 2. The output of algorithm AP(k) is X^k and Z^k. X^k denotes the set of optional parts chosen for execution and Z^k denotes the optimal solution found by AP(k). Note that Z^k approximates U^{max} or V^{max} depending on the objective function used.

The time complexity of procedure SEQ is $O(n)$: there is a loop for each task, and the UBT can be computed incrementally in $O(1)$ for each task. Since the number of times SEQ is executed is $O(n^k)$, the time complexity of AP(k) is $O(n^{k+1})$. Even for a small number of tasks (e.g., $n = 10$ tasks) this complexity seems rather high. However, we will demonstrate with simulations that for $k \leq 2$, the value of the system is very close to optimal. The worst-case performance ratio of AP(k) is $\frac{k}{k+1}$[16], which directs us to a solution with large k. However, since the complexity of the AP(k) algorithm is high for large k, we are interested in finding the smallest value of k such that AP(k) reaches a near-optimal solution. In the following example we will measure the real performance of the AP(k) algorithm in terms of complexity and run-time.

Example 1: Consider the set of tasks with its associated timing constraints and criticality values described in Table 2. Our goal is to apply the approximate algorithm for our objective functions and to evaluate its performance. The total utilization of the set of tasks in Table 2 denotes an overload (load = 120%). The utilization of the mandatory parts is 54% and the utilization of the optional parts is 66%. The problem to be solved is to handle the overload for this workload using the AP(k) algorithms, selecting the number of task to be included for execution such that our optimality criteria is satisfied. Tables 3 and 4 show the results from algorithm AP(k), for $k = 0, ..., 5$. The results shown in the Tables are: (a) the result from algorithm AP(k), which is X^k and Z^k (see Figure 2); (b) the number of combinations (N.C) necessary to obtain a solution; (c) the run-time of AP(k), which denotes the physical time in microseconds, using a PC Intel 233 MHz running Linux with 48MB of RAM; and (d) the set of optional parts chosen for execution (X^k).

Task	C_i	T_i	m_i	p_i	u_i	$Value$
τ_1	39.0	116.0	18.0	21.0	0.336	37.0
τ_2	49.0	154.0	23.0	26.0	0.318	30.0
τ_3	44.0	174.0	18.0	26.0	0.253	27.0
τ_4	47.0	195.0	20.0	27.0	0.241	29.0
τ_5	47.0	903.0	27.0	20.0	0.052	2.0

Table 2. Example Real-Time Workload: Mandatory and Optional parts and Criticality Values

k	$Z^k \approx U^{max}$	(N.C)	Run-Time (μs)	Result Set
0	89.030136	4	16	11000
1	91.244980	16	33	11001
2	91.244980	24	44	11001
3	99.715370	17	38	01110
4	99.715370	5	14	01110
5	99.715370	1	6	01110

Table 3. Results for maximizing utilization.

k	$Z^k \approx V^{max}$	(N.C)	Run-Time (μs)	Result Set
0	0.49	4	13	10010
1	0.492	16	36	10011
2	0.559	25	58	01110
3	0.559	17	46	01110
4	0.559	5	18	01110
5	0.559	1	7	01110

Table 4. Results for maximizing criticality.

For the goal of maximizing utilization, it is possible to observe that AP(k) with $k = 3$ yields optimal results, while for maximizing criticality AP(k) for $k = 2$ yields the optimal solution. For the case of utilization, tasks are ordered in terms of decreasing $\frac{p_i}{T_i}$ (i.e., τ_1, τ_2, τ_3, τ_4, τ_5), while for the case of the criticality the order is by decreasing $\frac{v_i}{p_i/T_i}$ (i.e., τ_4, τ_1, τ_3, τ_2, τ_5).

5.2.1 Measuring the Performance and Complexity of the Solutions

To extend the previous results, an experiment with 1000 randomly generated task sets has been conducted for measuring the performance and the complexity of the algorithm. For each experiment, a workload of 10 tasks has been generated with an overload (120% utilization for each task set). Results shown in Table 5 indicate the number of solutions within a certain percent close to optimal. For the two optimality criteria a near optimal solution (more than 91%) is obtained using $AP(2)$. For example, for maximizing utilization, results for $AP(2)$ indicate that 951 experiments yield a near optimal solution (0-0.1%) and the remaining 49 yield a $1 - 5\%$ near optimal solution.

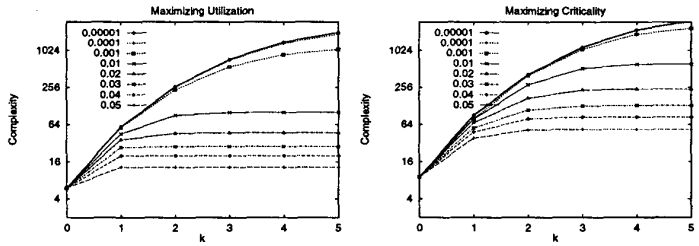

Figure 3. Complexity of AP(k) for different values of ϵ. Y axis is in log scale.

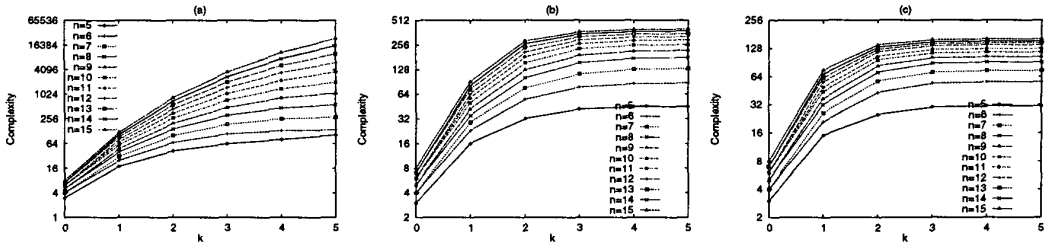

Figure 4. Complexity of AP(k) for different number of tasks, (a) ϵ= 0.001 (b) ϵ= 0.01 (c) ϵ= 0.02. Y axis is in log scale.

For maximizing criticality, results show that for AP(2) 911 experiments yield a near optimal solution (0-0.1%). This surprising result shows the excellent performance of the approximate algorithm AP(k).

	Maximizing Utilization				
k	$0-0.1\%$	$1-5\%$	$5-10\%$	$10-15\%$	$15-20\%$
0	617	251	119	13	0
1	662	336	2	0	0
2	951	49	0	0	0
3	999	1	0	0	0
4	1000	0	0	0	0
	Maximizing Criticality				
0	631	131	62	50	83
1	829	35	54	51	28
2	911	15	25	25	24
3	1000	0	0	0	0
4	1000	0	0	0	0

Table 5. Number of Solutions within x percent near optimal for 1000 Tasks Sets.

Further reductions in complexity could be obtained by relaxing the feasibility bound (see UBT). According to our feasibility tests, an element $s \in S$ is feasible if its feasibility condition is met. However a result less than 100% (e.g., 95%) could be *sufficient* for some applications which would cause an earlier end to the search for feasible solutions. Let us define ϵ, $0 < \epsilon < 1.0$, as the *feasibility error* which indicates a relaxation on the feasibility condition. The feasibility test shown in Equation (5) indicates a *sufficient feasibility condition.*

$$\sum_{i=1}^{n} \frac{m_i}{T_i} + \frac{x_i p_i}{T_i} \leq 1.0 - \epsilon \qquad (5)$$

We are interested in measuring the complexity (the number of elements searched) of the algorithm using the *sufficient feasibility condition* for different values of ϵ. We have conducted 1000 experiments comprising 10 tasks in each experiment whose total utilization (mandatory + optional) is 1.2. The average complexity of the algorithm is shown in Figure 3 for a varying value of ϵ. Note that the complexity of our algorithm is much lower than the worst-case complexity (exemplified by $\epsilon = 10^{-5}$) and that big reductions in complexity can be achieved by increasing the value of ϵ. For example for $\epsilon = 0.02$ and $k = 5$ the complexity achieved is 47 and 240 for maximizing utilization and criticality, respectively. However, it is worth noting that for all values of k the algorithm for maximizing utilization performs slightly better than for maximizing criticality. A possible explanation for this surprising result is the fact that the number of tasks is relatively low ($n = 10$). According to the worst-case complexity of the algorithm, having a higher number of tasks may increase considerably the complexity of the algorithm.

To measure the effect of the number of tasks on the complexity of the algorithm, an experiment has been conducted using 1000 randomly generated tasks sets for three values of $\epsilon = 0.001, \epsilon = 0.01, \epsilon = 0.02$ and for a varying number of tasks (from 5 to 15 tasks). Figure 4 shows that the complexity of the algorithm is relatively low even with a high degree of quality ($\epsilon = 0.02$).

From the results shown in Table 5 it can be concluded that for values of $k \leq 2$, 92.5% and 100% of the solutions are 95%-close to optimal when the criterion is to maximize criticality and utilization, respectively. Note that when $k = 1$, only 0.2% are less than 95% of optimal for the utilization criterion. From Figures 3 and 4, it can be seen that keeping ϵ between the values of 0.0001 and 0.02 is reasonable for achieving low complexity while maintaining high quality results.

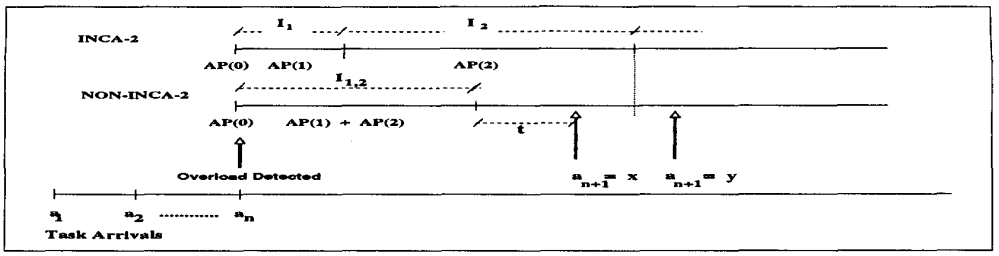

Figure 5. INCA-2 and NON-INCA-2 Servers: Execution Sequences.

6 Analysis of the INCA Server

As explained above, the INCA server is based on the incremental execution of several stages (Approximate Algorithms). At the end of each stage, information is available regarding the optional parts chosen for execution, and the resulting value of the objective function for the set of optional parts chosen. The process of scheduling the chosen optional parts at every stage will be called *commitment*. The INCA server executes the approximate algorithm AP(k) only after $AP(k-1)$ commits. In contrast, a non-incremental server would execute a number of stages $AP(0), ..., AP(k)$, before committing to the system.

In what follows we analyze the merit of the incremental execution by assuming $k = 2$ and comparing the following two servers,

INCA-2: This is the incremental execution sequence used by the INCA Server considering only two stages. That is, $AP(1)$, **commit**, $AP(2)$, **commit**.

NON-INCA-2: In this case, there is no incremental execution, and two stages are executed continuously before committing. That is, $AP(1) + AP(2)$, **commit**.

In Figure 5 we illustrate a sequence of n tasks arriving in the system, where the arrival of task τ_n causes an overload. $a_{n+1} = x$ and $a_{n+1} = y$ denote two instants of time at which a new tasks τ_{n+1} may arrive. Although the following analysis considers only two stages of the execution of the incremental server, the same analysis can be easily extended to include more than 2 stages.

After executing AP(0) at time a_n, X^0 and Z^0 are obtained. The set of optional parts, X^0, selected for execution will execute until AP(1) commits. Then a new set of optional parts X^1 will be chosen for execution and the process repeats. The resulting utilization or criticality value, Z^0, will depend on the objective function used. Let the utilization after committing at AP(k) be: $Z^k = (1 - \alpha_k)$, where α_k denotes the resulting slack time expressed in a percentage of resource usage. This slack time will be used for the execution of AP(k+1). Note that $\alpha_{k+1} \leq \alpha_k$ if the objective of the AP(k) algorithm is to maximize utilization.

Definition 1. Let ϕ_k be the worst-case execution time of algorithm AP(k) measured continuously (i.e., without interference from other tasks). We assume that the execution time of AP(0), ϕ_0, is negligible.

Definition 2. The interval of time at which AP(k) executes is defined as $I_k = \frac{\phi_k}{\alpha_{k-1}}$. If the execution of AP(1) and AP(2) is non incremental, the interval of time at which both AP(1)+AP(2) execute is defined as $I_{1,2} = \frac{\phi_1+\phi_2}{\alpha_0}$ (see Figure 5).

Utilization Metric We will compare the cumulative utilization achieved by INCA-2 and NON-INCA-2 during the period from a_n to a_{n+1}. If during a period I, the slack in the system is a constant, α, then the cumulative utilization given by Equation 1 can be alternatively computed from $CU(I) = I(1 - \alpha)$. Given that both INCA-2 and NON-INCA-2 will produce a utilization of $(1 - \alpha_0)$, resulting from AP(0), during the period $[a_n, a_n + I_1]$, and that both will produce a utilization of $(1-\alpha_2)$, resulting from AP(2) during the period $[a_n + I_1 + I_2, a_{n+1}]$, if $a_{n+1} > (a_n + I_1 + I_2)$, we will only compare the utilizations during the period $[a_n + I_1, a_{n+1}]$, where $a_{n+1} \leq a_n + I_1 + I_2$.

We will denote the cumulative utilization resulting from INCA-2 by CU while denoting the cumulative utilization resulting from NON-INCA-2 by CU_N. The following lemma proves that the incremental server always outperforms the non incremental server when the goal is to increase the cumulative utilization of the system.

Lemma 1. $CU([a_n, a_{n+1}]) \geq CU_N([a_n, a_{n+1}])$, if the objective of the AP(k) algorithm is to maximize utilization.

Proof: If $a_{n+1} \leq a_n + I_1$ then both servers produce the same utilization, while if $a_n + I_1 \leq a_{n+1} \leq a_n + I_{1,2}$ then the INCA server is at least as good as the NON-INCA server, since $CU([a_n + I_1, a_n + I_{1,2}]) = (I_{1,2} - I_1)(1 - \alpha_1)$ and $CU_N([a_n + I_1, a_n + I_{1,2}]) = (I_{1,2} - I_1)(1 - \alpha_0)$.

Now assume that $a_{n+1} = a_n + I_{1,2} + t$, where $0 \leq t \leq I_1 + I_2 - I_{1,2}$. In this case,

$$CU([a_n, a_{n+1}]) = I_1(1 - \alpha_0) + I_2(1 - \alpha_1)$$
$$CU_N([a_n, a_{n+1}]) = I_{1,2}(1 - \alpha_0) + t(1 - \alpha_2)$$

using the values of I_1, I_2 and $I_{1,2}$ from Definition 2 we get

$$CU([a_n, a_{n+1}]) = CU_N([a_n, a_{n+1}]) + \frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0} - t(1 - \alpha_2)$$

Given that $t \leq I_1 + I_2 - I_{1,2} = \frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0}$, and that $(1-\alpha_2) \leq 1$, we conclude that $CU([a_n, a_{n+1}])$ is equal to or larger than $CU_N([a_n, a_{n+1}])$. Finally, if $a_{n+1} > a_n + I_1 + I_2$, then INCA-2 and NON-INCA-2 produce the same utilization, $(1-\alpha_2)$, for any period latter than $a_n + I_1 + I_2$ and thus the result that $CU([a_n, a_{n+1}]) \geq CU_N([a_n, a_{n+1}])$ holds. **QED**

Criticality Metric The above lemma assumes that the objective of the servers is to maximize the system utilization. If however, the goal is to maximize the cumulative criticality (see Equation 2), then the relative performance of INCA-2 and NON-INCA-2 depends on the performance of the incremental algorithms AP(0), AP(1) and AP(2). Let Z^k be the criticality value achieved by AP(k), when the goal of AP(k) is to maximize criticality, and let α_k be the slack of the system after AP(k) commits. As before, we will use $CV(I)$ and $CV_N(I)$ to denote the cumulative criticality obtained by INCA-2 and NON-INCA-2 respectively during the period I.

Lemma 2. *If $a_{n+1} \leq I_{1,2}$, then $CV([a_n, a_{n+1}]) \geq CV_N([a_n, a_{n+1}])$, when the objective of AP(k) is to maximize criticality.*

Proof: $CV([a_n, a_n + I_1]) = CV_N([a_n, a_n + I_1]) = I_1 * Z^0$, while for $a_n + I_1 \leq t \leq a_{n+1}$, we have

$$CV([a_n + I_1, a_n + I_1 + t]) = \quad t * Z^1$$
$$CV_N([a_n + I_1, a_n + I_1 + t]) = \quad t * Z^0$$

The result follows since $Z^1 \geq Z^0$. **QED**

Lemma 3. *If $a_{n+1} > I_1 + I_2$ and the objective of AP(k) is to maximize criticality, then $CV([a_n, a_{n+1}]) \geq CV_N([a_n, a_{n+1}])$ if and only if $\alpha_1(Z^2 - Z^1) \leq \alpha_2(Z^2 - Z^0)$.*

Proof: Both INCA-2 and NON-INCA-2 produce the same criticality, namely Z^2 after the time $I_1 + I_2$. $Z^2 = CV([I_1 + I_2, a_{n+1}]) = CV_N([I_1 + I_2, a_{n+1}])$. Hence, we will only compare the cumulative criticality in the period from $a_n + I_1$ to $a_n + I_1 + I_2$.

$$CV([a_n + I_1, a_n + I_1 + I_2]) = I_2 * Z^1$$
$$CV_N([a_n + I_1, a_n + I_1 + I_2]) =$$
$$(I_{1,2} - I_1) * Z^0 + (I_1 + I_2 - I_{1,2}) * Z^2$$

Substituting for the values of I_1, I_2 and $I_{1,2}$ we get

$$CV([a_n + I_1, a_n + I_1 + I_2]) = \frac{\phi_2}{\alpha_1} * Z^1 =$$
$$CV_N([a_n + I_1, a_n + I_1 + I_2]) = \frac{\phi_2}{\alpha_0} * Z^0 + (\frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0}) * Z^2$$

Hence, $CV([a_n + I_1, a_n + I_1 + I_2]) = CV_N([a_n + I_1, a_n + I_1 + I_2]) + \frac{\phi_2}{\alpha_0}(Z^2 - Z^0) - \frac{\phi_2}{\alpha_1}(Z^2 - Z^1)$.

The lemma follows directly from the last Equation. **QED**

Given that AP(k) does not decrease the value of Z^k bellow Z^{k-1}, then $(Z^2 - Z^1) \leq (Z^2 - Z^0)$. However, nothing can be said about the relative values of α_1 and α_2 if AP(k) is used to improve criticality rather than utilization. The relative performance of the INCA and NON-INCA servers will be studied using simulations in the next section.

7 Simulation Experiments

The following simulation experiments have been designed to test the performance of the incremental server and its ability to achieve our optimality criteria using synthetic workloads. We are interested in measuring the performance of the algorithm using up to five stages of execution. According to the results obtained in Section 5.2 we are aware that statically we need to execute no more that 3 stages to achieve near-optimal results. Our goals are the following:

- to measure the quality of the results over a large set of dynamic tasks that arrive and leave the system at arbitrary instants of time.
- to measure and compare the performance among several stages for our different optimality criteria.

Each plot on the graphs represents the average of a set of 100 independent simulations. Up to the first 5 stages of the INCA server are executed in each simulation. Each curve INCA-k in the graphs denotes the execution of the INCA server in which only the first k stages are executed. That is, only the incremental execution of $AP(j)$, for $j = 0, ..., k$ is considered.

On each simulation 5,000 tasks are generated dynamically. Each task has a life-time (lt_i) that follows a uniform distribution between 400 and 600 instances (periods). At the end of its life-time, the task leaves the system. The utilization of task τ_i, U_i, is chosen as a random variable with uniform distribution between 5% and 20%. The period T_i of each task is chosen as a random variable with uniform distribution between 30 and 100 time units. The computation time of task τ_i is $C_i = T_i * U_i$. The experiments were conducted with a total utilization $U_T = \sum_i \frac{C_i}{T_i}$ varying between 80% and 180%. The number of tasks in the system (nt) executing at any time is computed by $nt = \frac{U_T}{U_i}$. The task inter-arrival time is computed by $IT_i = \frac{T_i * lt_i}{nt}$. The computation time of the optional part p_i is a random variable that follows a uniform distribution between 40% and 60% of the total computation time of task τ_i.

The execution time of AP(k) used was obtained from the experiments described by Figure 5, using a value of $\epsilon = 0.001$. Throughout this simulation experiments we will consider randomly generated *correlated tasks sets* [15], which means that the criticality is a linear function of the utilization[7]. The value v_i of each task is randomly distributed in $[U_i - 0.10, U_i + 0.10]$ such that $v_i > 0$ (i.e., plus or minus 10% from the utilization of the task).

The performance of our algorithms was measured according to the following metrics:

[7] It is hard to maximize the criticality value ratio for correlated tasks sets because many task combinations give similar results, therefore a larger number of combinations must be computed in order to find an optimal solution. We do not consider *uncorrelated task sets* because it is relatively easier to maximize their criticality value ratio (there is a large variation between the utilization of the tasks, making it easier to obtain a feasible and optimal solution).

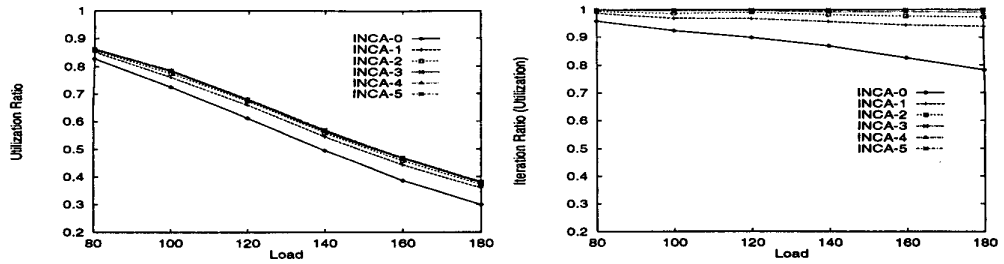

Figure 6. Utilization Ratio for up to 5 Stages of the Incremental Server.

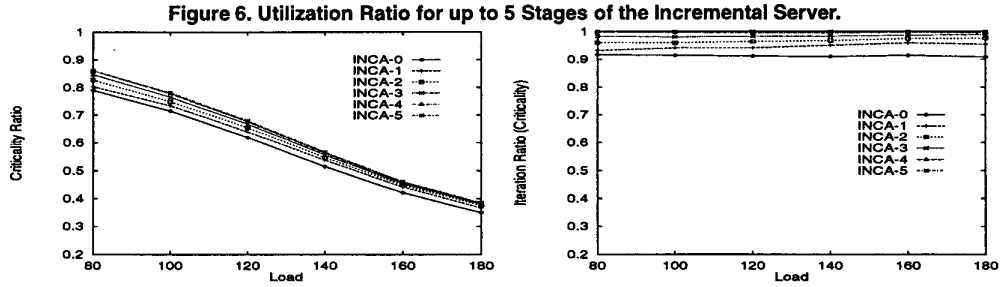

Figure 7. Criticality Ratio for up to 5 Stages of the Incremental Server.

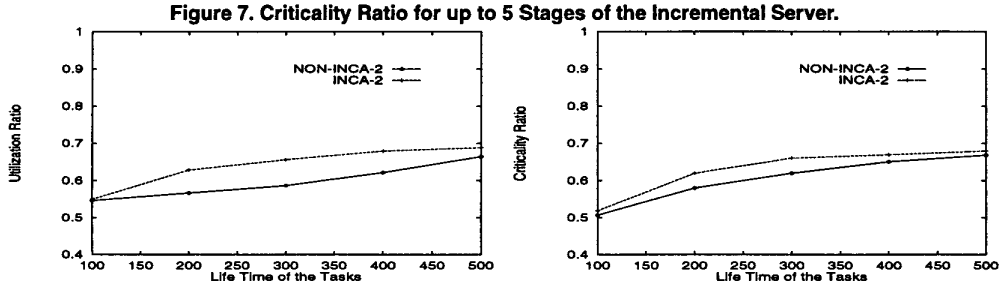

Figure 8. Utilization and Criticality Ratio for the INCA and NON-INCA Servers.

- **Utilization Ratio**: This metric is computed as follows,

$$\text{Utilization Ratio} = \frac{CU(I)}{\text{Total Utilization}} \quad (6)$$

where $CU(I)$ is the cumulative utilization of the system (see Section 4), for the interval of time I that denote the total duration of the schedule. The total utilization that can be achieved is computed by: $\sum_i r_i * p_i$, where the sum is over all tasks that arrive to the system in the interval of time I. Recall that r_i denotes the total number of instances of τ_i.

- **Criticality Ratio**: This metric is computed as follows,

$$\text{Criticality Ratio} = \frac{CV(I)}{\text{Total Criticality}} \quad (7)$$

where $CV(I)$ is the cumulative criticality (see Section 4), for the interval of time I that denote the total duration of the schedule. The total criticality that can be achieved is computed by, $\sum_i r_i * v_i$.

Two sets of experiments were conducted for our simulations. The first experiment, shown in Figures 6 and 7 was designed to compare the performance of INCA-k for different values of k, $0 \leq k \leq 5$. The second experiment, shown in Figure 8 was designed to compare the performance of the INCA-2 algorithm against the NON-INCA-2 algorithm. In the graphs shown in Figures 6 and 7, the utilization and the criticality ratio were measured. The left graph shows the value of the utilization metric, while the graph on the right shows the ratio of the value obtained by INCA-k and INCA-5, called *iteration ratio*.

The results shown in Figures 6 and 7 indicate that for values of $k > 2$ there is no significant improvement on the performance of the INCA server. Therefore, we will consider that INCA-5 achieves the maximum value possible in the system. For the utilization ratio, it is observed that INCA-2 achieves results close to those obtained by INCA-5 for all load values. Notice that for the iteration ratio, Figure 6 shows that INCA-0 achieves a performance that varies from 96% of INCA-5 for a load of 0.80, to 78% of INCA-5 for a load of 1.80. The performance of the algorithm for INCA-2 varies from 99% to 98.5% of INCA-5. It is important to note that even INCA-1 achieves a utilization performance higher or equal to 95% of INCA-5. The performance results for the criticality ratio indicate that INCA-k ($k = 0, ..., 4$) yield a performance higher than 90% of INCA-5 for all values of the load.

For our second experiment, Figure 8 shows the utilization and criticality ratio for the INCA-2 and the NON-INCA-2 servers. Our main interest in this experiment is to validate the results obtained previously in the analysis of the INCA server (see Section 6). In this experiment, the load of the system has a fixed value of 120%, and the life-time of each task lt_i varies between 100 and 500.

The behavior of the INCA-2 and NON-INCA-2 servers can be explained as follows: For low values of lt_i (e.g., 100 instances) both servers yield similar values because both servers are only able to execute AP(0) (which removes the overload). For $lt_i = 200, ..., 300$ the INCA-2 server yields much better results, because the INCA-2 server is capable of committing more frequently than NON-INCA-2. In this situation, the NON-INCA-2 server is able to execute a few times AP(1) and AP(2) but is mostly only able to execute AP(0). Finally, for the last values of lt_i (e.g. 400 and 500 instances) the performance of INCA-2 and NON-INCA-2 servers get closer because both servers are now able to commit both AP(1) and AP(2). In any case, the performance of the INCA-2 server is better that that of the NON-INCA server for all values of lt_i.

The results obtained in our simulations confirm the results obtained Section 6 and indicate that the INCA Server is a low cost and effective mechanism for scheduling real-time tasks under overloaded conditions.

8 Conclusion

In this paper, the problem of scheduling an overloaded real-time system has been studied. As observed by different research studies[8, 4, 9, 11], a significant performance degradation may occur in the system if the overload is not addressed efficiently. The set of tasks selected for execution is crucial for the proper operation of an overloaded real-time system. In our framework, each task has an assigned criticality value, and an objective function is evaluated in overloaded conditions such that an optimality criteria is met. The process of selecting tasks to discard while meeting the optimality criteria requires the exploration of a potentially large number of combinations. Since this process is too time consuming to be computed on-line, we have developed an Incremental Server (INCA) scheduling paradigm, which is based in a sequence of approximate algorithms. The execution of the approximate algorithms is conducted in an incremental manner, during the time at which the processor would otherwise be idle (slack-time), progressively refining the quality of the solution. The computational complexity of the INCA Server is high. However, we have shown that in practice only few stages need to be executed for achieving near-optimal solutions. An important feature of the incremental algorithm is that its run-time overhead and the quality of the solutions are parameters that can be controlled on-line. Our simulation results show that our approximate algorithm is efficient, has low overhead, and most importantly generates near-optimal solutions for overloaded real-time systems.

References

[1] H. Aydın, R. Melhem, D. Mossé, P. Mejía-Alvarez. "Optimal Reward-Based Scheduling of Periodic Real-Time Tasks", *Proc. of the IEEE Real Time Systems Symposium*, Dec. 1999.

[2] A. Burns, D. Prasad, A. Bondavalli, F.Di. Giandomenico, K. Ramamritham, J. Stankovic, L. Strigini "The Meaning and Role of Value in Scheduling Flexible Real-Time Systems", *J. of Systems Architecture, Jan. 2000*

[3] A. Burns and D. Prasad, "Value-Based Scheduling of Flexible Real-Time Systems for Intelligent Autonomous Vehicle Control", *Proc. of the 3rd. IFAC Symposium on Intelligent Autonomous Vehicles* March 1998.

[4] G.C. Butazzo, "Red: A Robust Earliest Deadline Scheduling Algorithm", *Proc. of Third Int. Workshop on Responsive Computing Systems*, Spain, Dec. 1998.

[5] G.C. Butazzo, "Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications", *Kluwer Academic Publisher*, 1997.

[6] H. Chetto and M. Chetto. "Some Results of the Earliest Deadline Scheduling Algorithm", *IEEE Transactions on Software Engineering*, Oct. 1989.

[7] M. Hamdaoui, P. Ramanathan. "A Dynamic Priority Assignment Technique for Streams with (m,k)-firm Deadlines", *IEEE Transactions on Computers*, Dec. 1995.

[8] S. Hwang, C.M. Chen and A.K. Agrawala. "Scheduling an Overloaded Real-Time System", *Proc. of the 1996 IEEE Conference on Computers and Communications.*

[9] E.D. Jensen, J.D. Northcutt, R.K.Clark, S.E. Shipman, F.D. Reynolds, D.P. Maynard, K.P. Loepfere. "Alpha: An Operating System for the Mission-Critical Integration and Operation of Large, Complex, Distributed Real-Time Systems – An Overview", *OSMCC*, Sept. 1989.

[10] G. Koren and D. Shasha. "Skip-over: Algorithms and Complexity for Overloaded Real-Time Systems", *Proc. of the IEEE Real Time Systems Symposium*, Dec. 1995.

[11] G. Koren and D. Shasha. "D-over: An Optimal Scheduling Algorithm for Overloaded Real-Time Systems", *Proc. of the IEEE Real Time Systems Symposium*, 1992.

[12] C.L. Liu, J. Layland. "Scheduling Algorithms for Multiprogramming in Hard Real-Time Environments", *J. ACM* 20(1). pp. 46-61, Jan. 1973.

[13] J.W. Liu and W.K. Shih. "Imprecise Computations", *Proceedings of the IEEE*, Jan. 1994.

[14] C.D. Locke. "Best-effort Decision Making for Real-Time Scheduling", *PhD. Thesis, CS-CMU. 1986*

[15] S. Martello, P. Toth. "Knapsack Problems", *Wiley*, 1990

[16] S. Sahni. "Approximate Algorithms for the 0/1 Knapsack Problem", *J. of the ACM*, Jan. 1975.