An Incremental Server for Scheduling Overloaded Real-Time Systems

Pedro Mejía-Alvarez, *Member*, *IEEE Computer Society*, Rami Melhem, *Fellow*, *IEEE*, Daniel Mossé, *Member*, *IEEE Computer Society*, and Hakan Aydin

Abstract—The need for supporting dynamic real-time environments where changes in workloads occur frequently requires a scheduling framework that: 1) explicitly addresses overload conditions, 2) allows the system to achieve graceful degradation while guaranteeing the deadlines of the most critical tasks in the system, and 3) supports an efficient runtime selection mechanism capable of determining the load to be shed from the system to handle the overload. In this paper, we propose a novel scheduling framework for a real-time environment that experiences dynamic workload 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. The process of selecting tasks (mandatory or optional parts) to discard while maximizing the value of the system requires the exploration of a potentially large number of combinations. Since an optimal solution is too time-consuming to be computed online, an approximate algorithm is executed incrementally whenever the processor would otherwise be idle, progressively refining the quality of the solution. This scheme 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.

Index Terms—Real-time systems scheduling, incremental processing, approximate algorithms.

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, avionics, and telecommunications systems. In such real-time systems, each task must complete and produce correct results by the specified deadline. In order to guarantee that deadlines 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.

- *P. Mejía-Alvarez is with CINVESTAV-IPN, Sección de Computación, Av. IPN. 2508, México D.F. E-mail: pmejia@cs.cinvestav.mx.*
- *R. Melhem and D. Mossé are with the Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260. E-mail: {melhem, mosse}@cs.pitt.edu.*
- *H. Aydin is with the Computer Science Department, George Mason University, Fairfax, VA 22030. E-mail: aydin@cs.gmu.edu.*

Manuscript received 9 Jan. 2002; revised 25 Oct. 2002; accepted 8 Jan. 2003. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115670.

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 instants of time. A framework is proposed for adjusting the system workload incrementally by relating the criticality value [5], [6] of the tasks to the resource allocation problem. The selection of a set of tasks that maximize an optimality criteria (expressed as the total value of the system) requires the exploration of a potentially large combinatorial space of solutions. In general, heuristics must be used, but, in particular, the overload mandates that heuristics with minimum overhead be used. Our approach to solve this problem is based on an online 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 *incremental processes* [4] or *progressive processing tasks* [24].

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 still images, and process this information during 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 optimal 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 related 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 and, in Section 5, the INCA server is described. In Section 6, we analyze the merit of the incremental execution of the INCA server by performing simulation experiments. Also, analytical results are presented to show the performance of the INCA Server when compared to a nonincremental server and to give insight into its effectiveness in handling overload conditions. In Section 7, we discuss how the INCA Server can be extended to enhance its applicability. 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 [9]. 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 [20] 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 [18] 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 [13] introduced the concept of time-valued functions which associate a value according to the task finishing time. Typically, 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 [12], 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 [7] 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; the strategy for handling the overload is to reject the least-valued task.

Other approaches for handling overload focus on providing less stringent guarantees for temporal constraints. In [15], 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 [11], 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. The elastic task model (ETM) is introduced by [8] aiming at increasing task periods to handle overloads in adaptive real-time control systems. Under ETM, periodic tasks are able to change their execution rate to provide different *quality of service* as a function of the current workload to keep the system underloaded.

Many of these techniques (e.g., [15], [11]) 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 [2], [19]. 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 [19], 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: 1) Many scheduling algorithms have been developed to handle overloaded conditions, but few research works studied, in practice, how far from optimal the performance of their algorithms and their complexity is ([16] provided a measure for the performance of their D-over algorithm using a metric called *competitive factor*); 2) 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 3) the time-value function of [13] or the error functions of the IC model [19] 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; instead, we use performance metrics that are largely available, such as utilization and criticality [5], [6].

3 Model

In our framework, we consider periodic preemptive tasks running on one processor. Tasks are independent (do not share resources) and have no precedence constraints. We assume a dynamic system in which each task τ_i arrives in the system at time a_i. Clearly, if one needs to ensure that all

tasks that may be submitted to the system are guaranteed to execute within their deadlines, one needs to provision the system to allow for the worst-case combination of tasks. This is NOT our goal: We consider dynamic systems in which an admission control procedure is executed to admit or reject tasks that are submitted. We assume that there is a contingency plan to take care of rejected tasks.

The *lifetime* 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. The time interval between the arrival of the first instances of two consecutive tasks τ_x and τ_y is defined as $l_{xy} = a_y - a_x$. Each task τ_i is composed of a mandatory part M_i followed by an optional part P_i. Note that this model is general in the sense that each task may have only an optional part and no mandatory part, or vice versa. The deadline of task τ_i is d_i, which is equal to the period, T_i, and C_i is its worst case computation time. 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 (so-called 0/1 constraint). We say the task τ_i *meets its deadline if its mandatory part completes by its deadline*. Further, in our model, once a task passes the admission control, it is guaranteed to meet the deadlines of all r_i instances until it leaves the system.

In our framework, we first will assume that the set of mandatory parts can never cause an overload in the system and only optional parts are subject to being rejected for execution. In Section 7, we relax this assumption and extend the framework to consider the more general case where mandatory parts can also cause overloads in the system and some tasks in the system will be denied execution. This is an extension that is appropriate for soft real-time systems.

Each task has an associated criticality value, v_i, which denotes its importance within the system. Methods to derive this criticality values are proposed in [5], [6], [20].

We assume that the tasks' characteristics (e.g., computation time, period, deadline, and criticality) are known at arrival time. Although the work in this paper focuses on Earliest Deadline First [18] and dynamic-priority schedulers, a more conservative fixed-priority scheduler can also be used [14], [18]. This is the subject of future work.

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 to determine:

1. 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;
2. the time that τ_i should be dispatched, if it is accepted;
3. the optional parts (if any) that should be excluded such that an optimality criteria is satisfied;
4. how to maximize the usage of the resources and the performance of the system with a reasonably low computation time while searching for a solution; and
5. how to take into account CPU time taken to compute the solution itself.

Each task in the system accrues an accumulated value upon executing a number of optional parts during its lifetime. 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 $I = [t_a, t_b]$.

The cumulative utilization achieved is computed by:

$$CU(t_a, t_b) = \sum_{i=1}^{n} e_i(t_a, t_b) \cdot p_i. \quad (1)$$

The cumulative criticality achieved is computed by:

$$CV(t_a, t_b) = \sum_{i=1}^{n} e_i(t_a, t_b) \cdot v_i, \quad (2)$$

where $e_i(t_a, t_b) = \lfloor \frac{t_b - t_a}{T_i} \rfloor$ denotes the number of instances that P_i is scheduled for execution during the interval of time $[t_a, t_b]$ and $0 \leq t_a < t_b$. For the interval of time between two consecutive arrivals, a_x and a_y, the accumulated value can be formulated as $CU(a_x, a_y)$ and $CV(a_x, a_y)$ for utilization and criticality, respectively. At a_x, the goal is to select the optional parts that maximize $CU(a_x, a_y)$ or $CV(a_x, a_y)$. This selection requires the searching of a usually large search space, as shown below. Note that the next arrival, a_y, may not be known and, therefore, the goal is to maximize CU or CV for the current set of tasks. The interval $[a_x, a_y]$ is defined to facilitate reasoning.

4.1 Definition of the Search Space

Consider n tasks, $\tau_1, \ldots, \tau_n$, 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 nonfeasible) of optional parts. More specifically, a search space is defined as $S = \cup_{k=0}^{n} S_k$, where $S_k = \{(x_1, \ldots, x_n); \sum_{i=1}^{n} x_i = k\}$, $x_i = 0$ 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

TABLE 1
Search Space for Three Tasks

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

(feasible and nonfeasible) 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.

The structure of the search space S is shown in Table 1 through an example with three tasks. For example, $\{1,1,0\}$ is the element in which p_1 and p_2 are executed and p_3 is discarded. Note that, since the search space is the power set of the tasks in the system, it is clearly exponential in size.

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 $s = \{x_1, \ldots, x_n\} \in S$.

The objective functions are defined as follows:

- $\mu(s)$: 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.

$$\mu(s) = \sum_{i=1}^{n} \frac{m_i + (x_i \cdot p_i)}{T_i.} \tag{3}$$

  $\mu(s)$ denotes the utilization of the system after choosing some optional parts for execution. For example, for $s = \{0,1,1\}$, $\mu(s) = \sum_{i=1}^{3}(\frac{m_i}{T_i}) + \frac{p_2}{T_2} + \frac{p_3}{T_3}$.
- $\gamma(s)$: 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.

$$\gamma(s) = \sum_{i=1}^{n} x_i \left(\frac{v_i}{T_i}\right). \tag{4}$$

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

Note that if the tasks $\{\tau_1, \ldots, \tau_n\}$ 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 a 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 $s = \{x_1, \ldots, x_n\}$ of the search space S, the utilization-based test is defined by

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

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 $U_m = \sum_{i=1}^{n} \frac{m_i}{T_i}$. Clearly, any single optional part with utilization $\frac{p_i}{T_i}$ greater than $1.0 - U_m$ can be immediately discarded.

4.4 The Optimization Problems

Our first optimization problem is related to shedding a number of optional parts that maximize 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 finding 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 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 [18]). In response to transient overload requests, the INCA Server adjusts the load of the system by executing a sequence of approximate algorithms, $AP(0), \ldots, AP(n)$, 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. Before the new task is scheduled, the INCA Server executes the approximate algorithm AP(0) to eliminate the overload. The solution provided by AP(0) allows the scheduler to *temporarily disable* the execution of some optional parts,

```
1:    INCA Server:
2:    input: a set of tasks τ_1,...,τ_n, and the newly arrived task τ_new
3:    If Σ_i m_i/T_i > 1 then reject the new task τ_new and exit;
4:    Execute AP(0);                                   (*remove the overload*)
5:    Compute the start time of the new task τ_new;
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);            (* the algorithm runs 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) at the appropriate time;
13:         else exit;
14:         k = k + 1;
15:   end;
```

Fig. 1. Incremental Server (INCA).

while providing a low cost nonoptimal solution. The slack-time[1] introduced in the system by the removal of the overload is used by the scheduler to execute approximate algorithms AP(k) (for $k = 1, \ldots, n$), 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, the INCA server will restart, taking into account the modified load of the system. The INCA Server stops its execution when 1) there is no more slack in the schedule to execute additional AP(k) algorithms or 2) the result of AP(k) is not better than the result of AP(k-1). The INCA Server is described in Fig. 1. We detail the INCA server operation in the next section, followed by details on the algorithm $AP(k)$.

5.1 Operation of the INCA Server

In this section, we detail the operation of the INCA Server.

1. **Activating the Incremental Server**. The conditions for activating the INCA server are:
   - **Task arrival**: The INCA server is activated if a new task, τ_{new}, arrives in the system and causes an overload. Our feasibility test (UBT) detects this condition. After detecting the overload (and before τ_{new} is scheduled), AP(0) is executed[2] and a set of optional parts are chosen to be discarded for removing the overload. For now, it is assumed that mandatory parts cannot be discarded, therefore, if the sum of mandatory parts, including that of the newly arrived task τ_{new}, cause an overload ($\sum_i \frac{m_i}{T_i} > 1$), then the new task is rejected (see line 3 in Fig. 1). In Section 7, we will discuss the more general case in which the task with least value in the system (rather than the newest task) may be rejected to solve the overload situation.
   - **Task departure**: In this case, clearly there is no overload of mandatory parts (that is, UBT is vacuously true). Then, we execute the approximate algorithms AP(k) (for k = 0, ..., n) incrementally to satisfy the optimality criteria for the set of optional parts remaining in the system.
2. **Scheduling the new task.** After removing the overload from the system (i.e., after AP(0)), the newly arrived task can be scheduled. However, regardless of its priority, the newly accepted task cannot be safely immediately scheduled because it may cause some other tasks to miss their deadlines, even after executing AP(0). This is because, unlike systems with only mandatory tasks, at the instant of the new arrival, we may choose to disable optional parts that a) already finished their current execution or b) have been preempted during the current execution. 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. To see how this happens, consider the following example.

Example 1. Let there be three tasks, τ_1, τ_2, and τ_3. Let $a_1 = 0$, $m_1 = 0$, $p_1 = 2.5$, $T_1 = 5$, $v_1 = 10$, $a_2 = 0$, $m_2 = 2.5$, $p_2 = 0$, $T_2 = 5.01$, $v_2 = 1$, and $a_3 = 3$, $m_3 = 3$, $p_3 = 0$, $T_3 = 6$, $v_3 = 10$. Consequently, $U_1 = 50\%$, $U_2 \approx 50\%$, and $U_3 = 50\%$. Note that, at time a_1, $U_1 + U_2 < 1$ and, at time a_3, $U_1 + U_2 + U_3 > 1$ and, thus, a task must be removed. Because there is only one optional part, τ_1 becomes the victim. However, τ_1 executed in interval $[0, 2.5]$ (due to the EDF dispatching discipline) and τ_2 executes in interval $[2.5, 5]$. At $a_3 = 3$, τ_1 is removed and τ_2 resumes execution, completing at time 5 (on time). τ_3 executes in interval $[5, 8]$ (because it has the smallest deadline), consequently making τ_2 miss its second deadline.

1. Slack-time is defined as the time at which the processor is not executing any task.
2. We assume that the execution times of UBT and AP(0) are negligible.

Clearly, the tasks that have not already started the execution of their optional parts allow the utilization to be subtracted immediately. But, there is no guarantee when a new task will arrive in the system and there is no guarantee that all optional parts will be unexecuted at that time. 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 deadline of all active tasks.[3]

Other criteria for scheduling tasks and enhancing their response time will be discussed in Section 7.

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. If there is such slack, $AP(1)$ is then executed during this slack. Analogously, $AP(k)$ (for $k = 2, \ldots, n$) 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 $AP(1), \ldots, AP(n)$ incrementally. That is, after $AP(k)$ ends, the workload of the system is adjusted by enabling and disabling some optional parts. After this, $AP(k+1)$ is executed and the process repeats.

The following rules are applied by the INCA server for the execution of optional parts:

Rule 1. All optional parts of tasks not selected by AP(k) (for k=0, ..., n) will finish the execution of the current instance and will be discarded after that. Therefore, there will be no partial execution of any optional part.

Rule 2. A new task will only start after the largest deadline of currently active tasks.

Note that an optional part that is discarded by AP(k) can be selected for execution by AP(k + 1). The reason for this is that, at each approximate algorithm, we may find different solutions involving a different set of optional parts to execute.

Algorithm AP(k) yields better (or at least equal) solutions, in terms of $\mu(s)$ or $\gamma(s)$, than AP(k - 1), but at the cost of higher execution times. However, the optional parts selected by 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 not be invoked further (see line 8 of Fig. 1). If, during the execution of AP(k), a new task arrives in or departs from the system, the INCA Server will restart its execution, taking into account the modified set of tasks in the system. If the incremental server is invoked when a task leaves the system, lines 3, 4, 5, 6 (from Fig. 1) are not executed.

3. Computing analytically the best time at which it is possible to accept the new task may involve some additional runtime overhead [10]. Therefore, we have decided to use a low-cost solution and to schedule the new task at the end of the last period of the instances running in the system when the new task arrives.

Since the INCA Server executes on the slack available in the system, it will execute as many approximate algorithms as possible.

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

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-like procedure [22] 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 $\gamma(s)$ [22]. This heuristic attempts to get the most value as a function of processor utilization; moreover, this heuristic has been shown to produce better results than considering only utilization or value [21], [22].

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 (i.e., UBT = false), it is discarded and a new subset with k optional parts is selected. If the subset passes the UBT (i.e., UBT = true), the remaining optional parts in this subset are also included in decreasing order of $\frac{p_i}{T_i}$ or $\frac{v_i}{p_i/T_i}$, while the UBT is satisfied. The solutions obtained from all feasible subsets are compared and the solution with higher value is chosen to be the solution generated by the AP(k) algorithm; if no subset satisfies UBT, AP(k) will return the value returned by AP(k-1) and exit.

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

The time complexity of procedure SEQ (in Fig. 2) 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. In fact, the quality of the solution (or performance) and the complexity are two conflicting forces in the determination of k. It has been shown in [22] that the worst-case performance ratio of AP(k) is $\frac{k+1}{k+2}$, which directs us to a solution with large k. On the other hand, the complexity of the AP(k) algorithm is high for large k, which leads us to attempt to find the smallest value of k such that AP(k) reaches a near-optimal solution. However, we will show that, for $k = 2$, the value of the system is very close to optimal. In the following example, we will measure the real performance of the AP(k) algorithm in terms of complexity and runtime overhead.

```
1: Algorithm AP(k):
2: input: F: Objective Function
3:        τ1, ..., τn ordered according to pi/Ti or vi/(pi/Ti)
4: output: X^k: set of optional parts chosen for execution.
5:         value_k: the optimized value computed for F.
6: value_k = 0;
7: for each M ⊂ {1, ..., n} such that |M| = k and UBT(M) = true
8:     call SEQ(M) to compute z and X
9:     if z + F(M) > value_k then
10:    begin
11:        value_k = z + F(M);
12:        X^k = X ∪ M;
13:    end
14: procedure SEQ(M):
15: input: M: initial set of optional parts
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
```

Fig. 2. Approximate Algorithm: AP(k).

Example 2. Consider the set of tasks with its associated timing constraints and criticality values described in Table 2. The total utilization of the set of tasks in Table 2 creates an overload of 20 percent (that is, total utilization of 120 percent). The utilization of the mandatory and optional parts is 54 percent and 66 percent, respectively. The problem to be solved is to handle the overload using the AP(k) algorithms such that our optimality criteria is satisfied.

Tables 3 and 4 show the results from algorithm AP(k), for $k = 0, \ldots, 5$ for maximizing utilization (CU) and for maximizing critically (CV), respectively.

The results shown in the tables are:

1. $value_k$: the result from algorithm AP(k);
2. C_k: the complexity of algorithm AP(k), that is, the number of subsets in the search space explored before a solution is found;
3. the runtime of AP(k), which denotes the physical time in microseconds, using a PC Intel 233 MHz running Linux with 48MB of RAM; and
4. the result set X^k: the set of optional parts chosen for execution.

TABLE 2
Example Real-Time Workload: Mandatory and Optional Parts and Criticality Values

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 3
Results for Maximizing Utilization, for $k > 3$, UBT = False

k	$value_k$	(C_k)	Run-Time (μs)	Result Set
0	89.030136	4	16	1 1 0 0 0
1	91.244980	16	33	1 1 0 0 1
2	91.244980	24	44	1 1 0 0 1
3	99.715370	17	38	0 1 1 1 0
4	99.715370	5	14	0 1 1 1 0
5	99.715370	1	6	0 1 1 1 0

Note that, for $k > 3$, UBT = false and, therefore, AP(k) does not update $value_k$ or X^k (it keeps the values for AP(3), in this example).

It is possible to observe that, for the goals of maximizing utilization and criticality, $AP(k)$ with $k = 3$ yields optimal results.

5.3 The Performance and Complexity Trade Off

To measure the performance of our proposed algorithm, an experiment with 1,000 randomly generated task sets has been conducted. For each experiment, a workload of 10 tasks has been generated with an overload (120 percent 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 percent) is obtained using $AP(2)$. For maximizing utilization, results for $AP(2)$ indicate that 951 experiments yield a near optimal solution (within 0-0.1 percent of optimal) and the remaining 49 yield a 1-5 percent near optimal solution. For maximizing criticality, results show that, for $AP(2)$, 911 experiments yield a solution within 0-0.1 percent of optimal. This shows the excellent performance of the approximate algorithm $AP(k)$.

Further reductions in *complexity* (the number of elements to be searched before a solution is found) could be obtained by relaxing the feasibility bound. As presented above, an element $s \in S$ is feasible if UBT is met (i.e., UBT = true). However, a result less than 100 percent (e.g., 95 percent) could be *sufficient* for some applications, which would cause

TABLE 4
Results for Maximizing Criticality, for $k > 3$, UBT = False

k	$value_k$	(C_k)	Run-Time (μs)	Result Set
0	0.4676	4	13	1 0 0 1 0
1	0.4698	16	36	1 0 0 1 1
2	0.5137	25	58	1 1 0 0 0
3	0.5159	17	46	1 1 0 0 1
4	0.5159	5	18	1 1 0 0 1
5	0.5159	1	7	1 1 0 0 1

TABLE 5
Number of Solutions with x Percent Near Optimal for 1,000 Task Sets

	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

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 (5) indicates a *sufficient feasibility condition*.

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

Note that increasing the value of ϵ may reduce the number of elements to be searched in S, but at the cost of reduced CPU usage.

We are interested in measuring the complexity of the AP(k) algorithm using the *sufficient feasibility condition* for different values of ϵ. We have conducted 1,000 experiments comprising 10 tasks in each experiment whose total utilization (mandatory + optional) is 120 percent. The average number of subsets examined during the AP(5) algorithm is shown in Fig. 3 as a curve for each value of ϵ $(10^{-5} \leq \epsilon \leq 0.05)$. The complexity of the algorithm (shown in the Y axis of Fig. 3) denotes the number of subsets in the search space explored before a solution is found. Note 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.

From Fig. 3, it can be noted that keeping ϵ between the values of 0.01 and 0.03 is reasonable for achieving low complexity; clearly, these values maintain high quality results.

6 Evaluation of INCA Server

In this section, we start by presenting an empirical evaluation of the INCA server, achieved through extensive simulations. We then provide an intuitive analytical comparison of incremental and nonincremental servers, which is the basis for why the incremental server is preferable in overloaded situations.

6.1 Simulation Experiments

The following simulation experiments have been designed to test the performance of the INCA Server and its ability to achieve our optimality criteria using synthetic workloads. From the results obtained in Section 5.2, we need to execute no more that three stages to achieve near-optimal results. For this reason, we measure the performance of the INCA Server using up to five stages of execution.

Our goals in this simulation study 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 five 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 $j = 0, \ldots, k$ is considered.

For each simulation, 5,000 tasks are generated dynamically. Each task has a lifetime (r_i) that follows a uniform distribution between 400 and 600 instances (periods). At the end of its lifetime, the task leaves the system. The utilization of task τ_i, U_i, is chosen as a random variable with uniform distribution between 5 percent and 20 percent. An average utilization U_{av} is computed for all tasks currently executing in the system. 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 $C_i = T_i \cdot U_i$. The experiments were conducted considering an average number of tasks in the system (nt) executing at any time, $nt = \frac{U_T}{U_{av}}$, where U_T denotes the total utilization of the tasks executing at any time, $U_T = \sum_i \frac{C_i}{T_i}$, which varies between 80 percent and 180 percent.

The arrival time of task τ_{i+1} is computed by $a_{i+1} = \frac{T_{i+1} \cdot r_{i+1}}{nt}$ and $a_1 = 0$. The computation time of the optional part p_i is a random variable that follows a uniform distribution[4] between 40 percent and 60 percent of the total computation time of task τ_i.

The execution time of AP(k), ϕ_k, used in the experiments was obtained from actual experiments described by Fig. 3, using a value of $\epsilon = 0.001$. Throughout these simulation experiments, we considered randomly generated *correlated tasks sets*, which means that the criticality is a linear function of the utilization.[5] The value v_i of each task is randomly distributed in $[U_i - 0.10, U_i + 0.10]$ such that $v_i > 0$.

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

4. Additional experiments were conducted with p_i following a uniform distribution between 20-40 percent and 60-80 percent, obtaining similar results.

5. 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) [21].

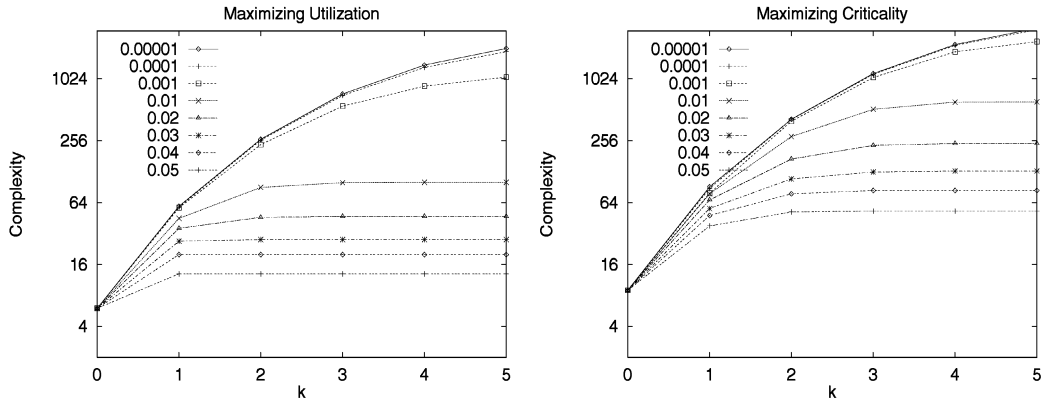

Fig. 3. Complexity of AP(5) for different values of ϵ. Y axis is in log scale, denoting the number of subsets examined.

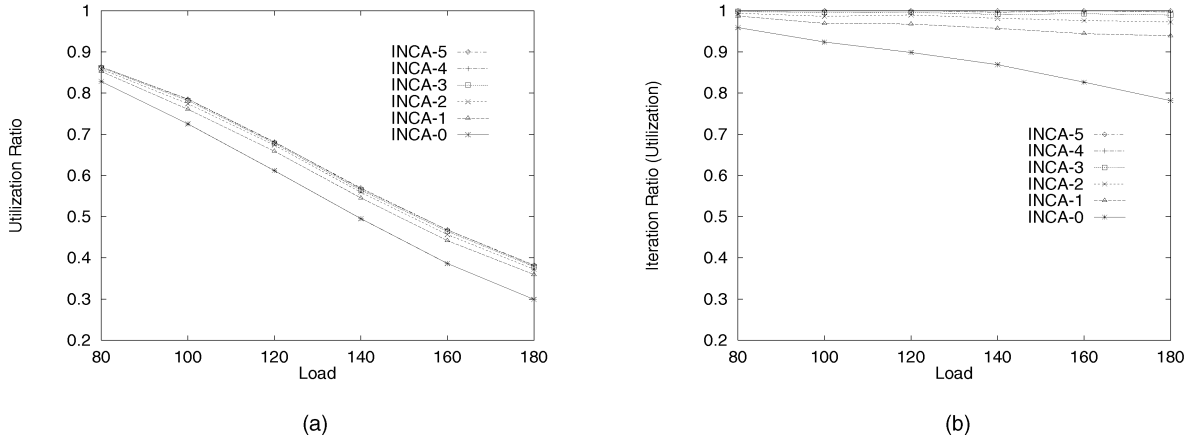

Fig. 4. Absolute value of utilization (a) and utilization normalized to INCA-5 performance (b).

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

$$\text{Utilization Ratio} = \frac{\sum_i e_i \cdot \frac{p_i}{T_i}}{\text{Total Utilization}}, \qquad (6)$$

where e_i denotes the number of instances where optional part p_i is executed.

The *total utilization* is computed by: $\sum_i r_i \cdot \frac{p_i}{T_i}$, where r_i denotes the total number of instances of τ_i and the sum is over all tasks that arrive to the system for the complete duration of the schedule.

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

$$\text{Criticality Ratio} = \frac{\sum_i e_i \cdot v_i}{\text{Total Criticality}}. \qquad (7)$$

The total criticality is computed by: $\sum_i r_i \cdot v_i$.

Two sets of experiments were conducted for our simulations. The first experiment, whose results are shown in Figs. 4 and 5, was designed to compare the performance of INCA-k for different values of k, $0 \leq k \leq 5$. In the graphs shown in Figs. 4 and 5, 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 second experiment, shown in Fig. 6, was designed to compare the performance of the INCA-2 algorithm against the NON-INCA-2 algorithm.

The results shown in Figs. 4 and 5 indicate that, for values of $k > 2$, there is no significant improvement on the performance of the INCA server for all load values. Similar results were obtained on experiments considering $k = 6, \ldots, 10$. Such results allow us to conclude that INCA-5 achieves near-optimal results. The performance results for the utilization ratio (shown in Fig. 4) indicate that INCA-0 achieves a performance that varies from 96 percent of INCA-5 for a load of 80 percent to 78 percent of INCA-5 for a load of 180 percent. The performance of the algorithm for INCA-2 varies from 99 percent to 98.5 percent of INCA-5. It is important to note that even INCA-1 achieves a utilization performance higher than or equal to 95 percent of INCA-5 for all load values. The performance results for the criticality ratio (shown in Fig. 5) indicate that INCA-k ($k = 0, \ldots, 5$) yields a performance higher than 90 percent of INCA-5 for all values of the load.

For our second experiment, Fig. 6 shows the utilization and criticality ratios for the INCA-2 and the Non-INCA-2 servers. In this experiment, the load of the system has a fixed value of 120 percent and the lifetime r_i of each task varies between 100 and 500.

The comparative behavior of the INCA-2 and Non-INCA-2 servers can be explained as follows: For low values of r_i (e.g., 100 instances and below), both servers yield similar values because they are only able to execute AP(0) (which removes the overload). We observed that, on average, it takes from 90 to 110 instances for INCA-2 to

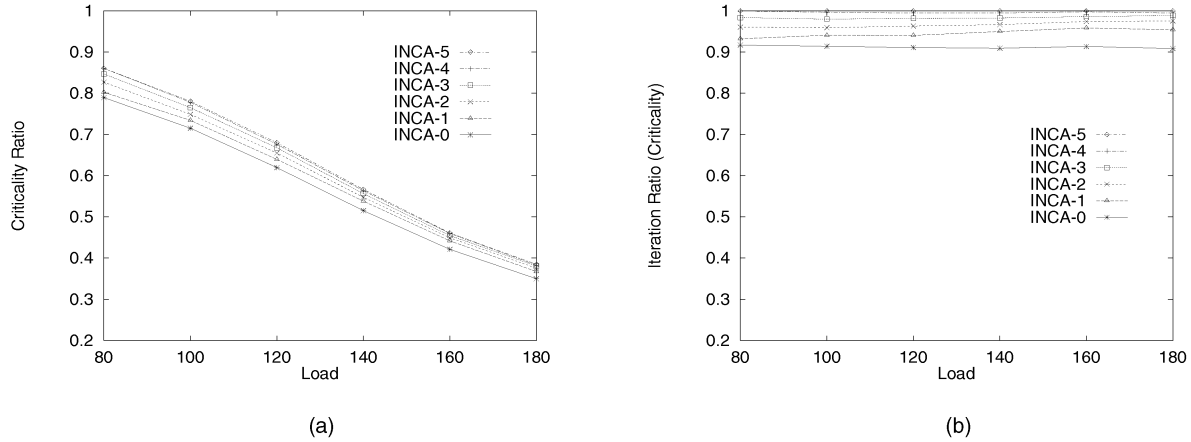

Fig. 5. Absolute value of criticality (a) and criticality normalized to INCA-5 performance (b).

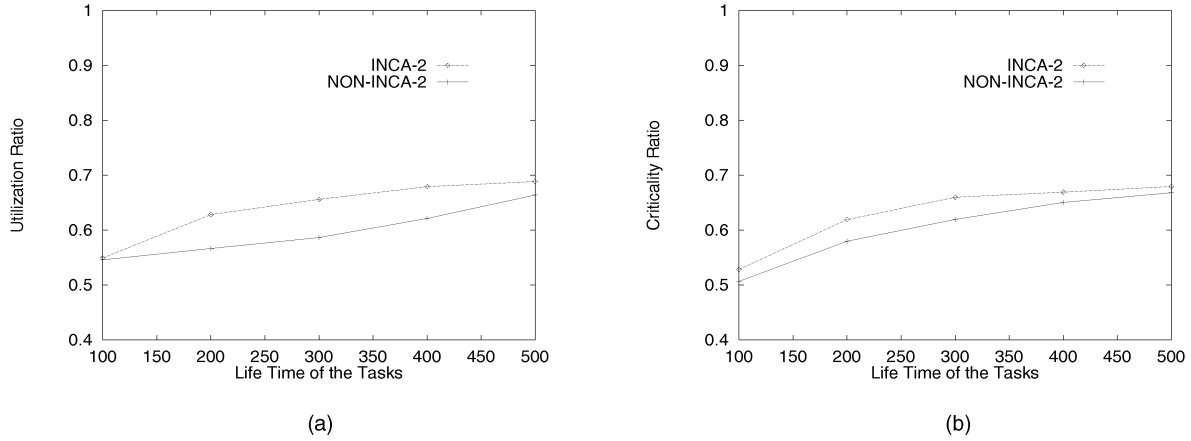

Fig. 6. Utilization (a) and criticality (b) ratios for the INCA-2 and Non-INCA-2 Servers.

commit to AP(1) and from 350 to 380 instances to commit to AP(2). These results are explained by the fact that the execution time of AP(k), ϕ_k, used was too high. However, additional experiments were conducted using values of $\epsilon = 0.01$ and $\epsilon = 0.02$, with consequent reductions on ϕ_k, as indicated in Fig. 3. For both values of ϵ, graphs were obtained with similar shapes and similar results (i.e., INCA-2 was always better than Non-INCA-2) than those produced using a value of $\epsilon = 0.001$.

For higher values of r_i (200 to 300 instances), the INCA-2 server yields 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 AP(1) and AP(2) a few times, but is mostly only able to execute AP(0). Finally, for the highest tested values of r_i (400 to 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 than that of the Non-INCA server for all tested values of r_i.

The results obtained in our simulations confirm the results obtained in Section 5.3 and indicate that the INCA Server is a low cost and effective mechanism for scheduling real-time tasks under overloaded conditions.

6.2 Analytical Comparison of the INCA and Non-INCA Servers

In this section, we carry out an approximate analysis of the performance of an INCA server when compared against that of a Non-INCA Server. We are interested in validating analytically the results obtained in previous simulation experiments. As explained above, the INCA server is based on the incremental execution of several stages of *approximate algorithms* AP(k). At the end of each stage, information is available regarding the set of optional parts chosen for execution, X^k, and the resulting value of the objective function, $value_k$, for the set of optional parts chosen. The approximate algorithms are such that the resulting value of AP(k+1), $value_{k+1}$, is as least as good as the resulting value of AP(k), $value_k$.

The INCA server executes AP(k - 1) which chooses a set of optional parts to remove and a set to execute (X^k). This set is passed on to the scheduler (we say AP(k - 1) *commits*), which schedule the chosen tasks. After AP(k - 1) commits, AP(k) can be executed. In contrast, a *non-INCA server* would execute a number of stages $AP(0), \ldots, AP(k)$ before committing to the system. In what follows, we analyze the merit of the incremental execution by assuming $k = 2$ (see Fig. 7). We denote by I_k the amount of time it takes to execute algorithm AP(k) while executing the tasks chosen

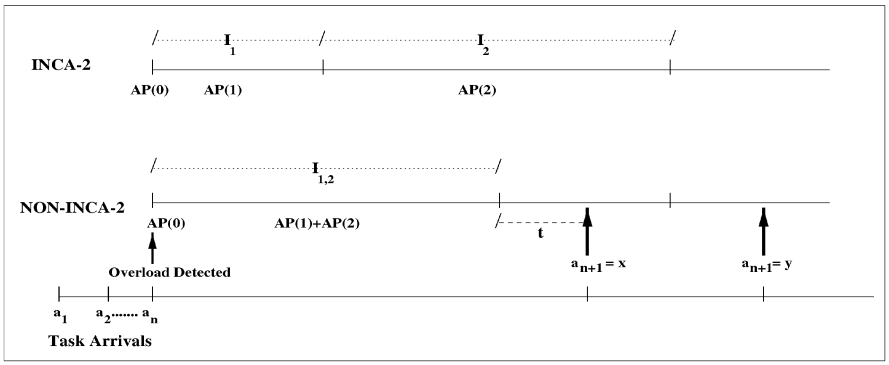

Fig. 7. INCA-2 and Non-INCA-2 Servers: execution sequences.

by AP(k - 1). Although we present the analysis through a $k = 2$ example, the same analysis can be extended to $k > 2$.

INCA-2: This is the incremental execution sequence used by the INCA Server considering only the following stages: $AP(0)$, **commit**, $AP(1)$, **commit**, $AP(2)$, **commit**.

Non-INCA-2: In this case, there is no incremental execution and three stages are executed back to back before committing. That is, $AP(0) + AP(1) + AP(2)$, **commit**.

In Fig. 7, we illustrate a sequence of n tasks arriving in the system such that the arrival of task τ_n (arriving at time a_n) causes an overload. After executing AP(0) at time a_n, X^0 and $value_0$ are obtained. The set of optional parts, X^0, selected for execution will execute (given the restrictions about task start times, as mentioned in Section 5.1, item 2) until AP(1) commits at time $a_n + I_1$. Then, a new set of optional parts X^1 will be chosen for execution and the process repeats. Recall that the resulting utilization or criticality values will depend on the objective function used.

After committing at AP(k), let α_k ($0 \leq \alpha_k \leq 1$) denote the resulting slack time expressed in a percentage of resource usage (that is, system utilization is expressed by $1 - \alpha_k$). 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. If ϕ_k denotes the worst-case execution time of algorithm AP(k) measured continuously (i.e., without interference from other tasks), the interval of time during which AP(k) executes, while also executing the tasks committed by AP(k-1), is approximately $I_k = \frac{\phi_k}{\alpha_{k-1}}$. This is because AP(k) is executed on the slack time resulting from AP(k - 1); note that computing the time that AP(k) executes through the slack α_{k-1} is an approximate measure since the slack is not necessarily uniformly distributed. For a non-INCA server, the interval of time during which AP(k) and AP(k + 1) execute is approximately $I_{k,k+1} = \frac{\phi_k + \phi_{k+1}}{\alpha_{k-1}}$ (see Fig. 7). Note that, during $I_{k,k+1}$, the only tasks being executed are those committed by AP(k - 1). We also assume that $I_{k,k+1} < I_k + I_{k+1}$.

To exemplify the concepts above, consider a new arrival, τ_n, causing an overload. Let us consider the associated timing constraints of the approximate algorithms AP(1) and AP(2) as follows: $\phi_1 = 10$, $\phi_2 = 15$, $\alpha_0 = 0.2$, and $\alpha_1 = 0.1$. Without loss of generality, assume that AP(0) committed at time $t_0 = 0$. The intervals of time at which $AP(1)$, $AP(2)$, and $AP(1) + AP(2)$ execute are $I_1 = \frac{10}{0.2} = 50$, $I_2 = \frac{15}{0.1} = 150$, and $I_{1,2} = \frac{10+15}{0.2} = 125$.

6.2.1 Utilization Metric

We will compare the cumulative utilization achieved by INCA-2 and Non-INCA-2 during the interval of time from a_n to the next arrival a_{n+1}. If, during an interval I, the slack in the system is a constant, α, then the cumulative utilization given by (1) can be alternatively computed from $CU(I) = I(1 - \alpha)$; strictly speaking, I is the interval, but we will abuse the notation and let I denote both the interval and its length when used in a computation (as above).

To see why the utilization achieved by an incremental server is higher than that achieved by a nonincremental server, consider the following. Both INCA-2 and Non-INCA-2 have the same utilization $(1 - \alpha_0)$ after AP(0) commits and before AP(1) commits, that is, during the interval $[a_n, a_n + I_1]$. Also, both will have the same utilization $(1 - \alpha_2)$ after AP(2) commits, that is, during the interval $[a_n + I_1 + I_2, a_{n+1}]$. Therefore, we only need to compare the utilization achieved in the interval $[a_n + I_1, a_n + I_1 + I_2]$.

We denote the cumulative utilization resulting from INCA-2 and Non-INCA-2 during interval I by $CU(I)$ and $CU_N(I)$, respectively.

Result 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. In the Appendix. □

The proof in the Appendix shows that, minus the inaccuracies of the intervals I_k, the incremental server always outperforms the nonincremental server when the goal is to increase the cumulative utilization of the system. To see that, first note that, 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 the INCA server had the opportunity to commit at least one stage. Also, if $a_{n+1} = (a_n + I_{1,2} + t)$ (arrival $a_{n+1} = x$ in Fig. 7), where $0 \leq t \leq (I_1 + I_2 - I_{1,2})$,

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

where the first term of CU shows the value accrued after committing AP(0) for I_1 and committing AP(1) for the rest of the interval. Likewise, the first term of CU_N shows the value committed after committing for AP(0) and AP(1 + 2).

Further algebraic manipulations shown in the Appendix demonstrate that, for all values of $t, 0 \leq (t \leq I_{1,2} - I_1 + t)$, the INCA server has better performance with utilization as the metric.

6.2.2 Criticality Metric

The above result assumes that the objective of the servers is to maximize the system utilization. If, however, the goal is to maximize the cumulative criticality, as in (2), then the relative performance of INCA-2 and Non-INCA-2 depend on the performance of the incremental algorithms AP(0), AP(1), and AP(2). Let $value_k$ 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.

Result 2. *If $(a_{n+1} \leq I_{1,2})$ or $(a_{n+1} > I_1 + I_2)$, 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. We know that

$$CV([a_n, a_n + I_1]) = CV_N([a_n, a_n + I_1]) = I_1 \cdot value_0.$$

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

$$CV([a_n, a_{n+1}]) = (I_1 \cdot value_0) + (t \cdot value_1)$$
$$CV_N([a_n, a_{n+1}]) = (I_1 + t) \cdot value_0.$$

The result follows since $value_1 \geq value_0$.

Now, assume that $a_{n+1} > I_1 + I_2$. In this case, both INCA-2 and Non-INCA-2 produce the same criticality, namely, $value_2$, after the time $I_1 + I_2$ and then $CV([a_n, a_{n+1}]) = CV_N([a_n, a_{n+1}])$. □

Finally, if $a_{n+1} = (a_n + I_{1,2} + t)$, where $0 \leq t \leq I_1 + I_2 - I_{1+2}$ we will show next that it is not possible to say that the condition holds when AP(k) is used to maximize criticality. In this case,

$$\begin{aligned} CV([a_n, a_{n+1}]) &= I_1 value_0 + (I_{1,2} - I_1 + t) value_1 \\ &= \frac{\phi_1}{\alpha_0} value_0 + \left(\frac{\phi_1 + \phi_2}{\alpha_0} - \frac{\phi_1}{\alpha_0} + t\right) value_1 \\ &= \frac{\phi_1}{\alpha_0} value_0 + \left(\frac{\phi_2}{\alpha_0} + t\right) value_1 \\ CV_N([a_n, a_{n+1}]) &= I_{1,2} value_0 + t \cdot value_2 \\ &= \left(\frac{\phi_1 + \phi_2}{\alpha_0}\right) value_0 + t \cdot value_2. \end{aligned}$$

If $CV([a_n, a_{n+1}]) \geq CV_N([a_n, a_{n+1}])$, we get

$$\begin{aligned} \frac{\phi_1}{\alpha_0} value_0 + \left(\frac{\phi_2}{\alpha_0} + t\right) value_1 &\geq \left(\frac{\phi_1 + \phi_2}{\alpha_0}\right) value_0 \\ &\quad + (t * value_2) \Rightarrow \\ \left(\frac{\phi_2}{\alpha_0} +\right) value_1 &\geq \left(\frac{\phi_2}{\alpha_0}\right) value_0 + (t * value_2) \Rightarrow \\ \frac{\phi_2}{\alpha_0}(value_1 - value_0) &\geq t(value_2 - value_1). \end{aligned}$$

If $t = 0$, the condition holds. If, however, the value of t is equal to its maximum possible value, $t_{max} = I_1 + I_2 - I_{1,2} = (\frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0})$, we get

$$\begin{aligned} \frac{\phi_2}{\alpha_0}(value_1 - value_0) &\geq \left(\frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0}\right)(value_2 - value_1) \Rightarrow \\ \frac{\phi_2}{\alpha_0} value_1 - \frac{\phi_2}{\alpha_0} value_0 &\geq \frac{\phi_2}{\alpha_1} value_2 - \frac{\phi_2}{\alpha_1} value_1 \\ &\quad - \frac{\phi_2}{\alpha_0} value_2 + \frac{\phi_2}{\alpha_0} value_1 \Rightarrow \\ \frac{\phi_2}{\alpha_0}(value_2 - value_0) &\geq \frac{\phi_2}{\alpha_1}(value_2 - value_1). \end{aligned}$$

Using the *worst-case performance ratio of* $value_k = \frac{k+1}{k+2}$ (see Section 5.2 and reference [22]), we have that $(value_2 - value_0) > (value_2 - value_1)$ since $(\frac{3}{4} - \frac{1}{2}) > (\frac{3}{4} - \frac{2}{3})$. However, since $\frac{\phi_2}{\alpha_0} \leq \frac{\phi_2}{\alpha_1}$, it is not possible to say analytically that, when AP(k) is used to maximize criticality rather than utilization, the INCA server has better performance than the Non-INCA server. However, as shown in Fig. 6, the performance of the INCA is typically better than Non-INCA servers.

7 Extensions

In this section, we discuss four extensions that may be used to enhance the applicability of the INCA Server. The first is an extension which extends the Feasibility Test to consider that mandatory parts may also cause an overload. The second extension uses the Total Bandwidth Server [23] to improve the scheduling time of the new tasks. In the third and fourth extensions, aperiodic tasks and tasks with resource sharing constraints can be introduced easily in our framework using the Total Bandwidth Server [23] and the Stack Resource Policy [3], respectively.

- **Feasibility Tests: A more general case**. The criteria for activating the INCA server assumes that the mandatory parts can never cause an overload. Therefore, only optional parts must be included into the search space S.

  In general, we should also consider that the mandatory parts can cause an overload. The new feasibility test used on the INCA Server for this general case is as follows:

$$UBT(s) = \begin{cases} false & \text{if } \sum_i \frac{m_i}{T_i} > 1 \quad \text{(C1)} \\ false & \text{if } \sum_i \frac{m_i + (x_i p_i)}{T_i} > 1 \quad \text{(C2)} \\ true & otherwise. \end{cases}$$

  In (C1) only mandatory parts must be included into S, while, in (C2), only optional parts must be included into S.

  The new condition is used when *the utilization of the mandatory parts exceeds the total capacity of the system*. This condition can only be true when dealing with soft real-time systems since, if the condition $\sum_i \frac{m_i}{T_i} > 1$ is met, some mandatory task(s) must be discarded to avoid the overload.

  The INCA Server will consider only mandatory parts to be included into the search space S, will discard all optional parts from the system, and will proceed as described above for AP(k) algorithms

(that is, AP(0), commit, AP(1), etc.). The optional parts discarded will be removed from execution until the next activation of the INCA Server (where a new case for the feasibility test may be found), while a mandatory part that is discarded by AP(k) will possibly be selected for execution by AP(k + 1). The INCA Server will not reject the newly arrived task, as in the INCA Algorithm (see line 3 of Fig. 1), because the approximate algorithms $AP(k)$ will select for execution only those mandatory parts with the most value in the system. However, note that the selection by different stages of AP(k) may lead to different mixes of mandatory parts at different instants of time (tasks complete, arrive, and AP(k) execute).

Note that Rules 1 and 2 discussed in Section 5.1 will also be applied in this case, with the difference that only mandatory parts could be discarded.

- **Scheduling new tasks**. The criteria used by the INCA server for scheduling newly arriving tasks is that they cannot be scheduled to execute until the end of the longest period of all tasks that have been preempted when the new task arrived. Although this criteria has low overhead, it may cause new tasks to wait for a long time before they are scheduled. To mitigate this effect, one can extend this criteria to enhance the schedulability of the new tasks by including the Total Bandwidth Server (TB) [23]. The TB Server is a low-cost scheduling mechanism designed to improve the response time of soft aperiodic requests in dynamic environments where tasks are scheduling according to the EDF [18] policy. This mechanism simply assigns a suitable deadline to each request such that no task in the system misses its deadline. When the kth task arrives at time $t = a_k$, it receives a deadline, $d_k = max(a_k, d_{k-1}) + \frac{C_k^a}{U_s}$, where C_k^a is the execution time of the new task, $U_s = (1 - U_m)$ is the server utilization factor, and U_m is the utilization of the mandatory parts in the system. By definition, $d_0 = 0$. The new task is then inserted into the ready queue of the system and then scheduled by the INCA Server. The response time of the new tasks can be further improved by using the TB* algorithm. The key idea of the TB* algorithm is to assign to the new task a deadline shorter than that obtained by the TB algorithm. The TB* algorithm uses an iterative procedure to shorten the deadline assigned to the new tasks. At each iteration, it verifies if the set of tasks is still schedulable. The TB* algorithm stops when the deadline assigned at some iteration makes some of the tasks in the system to miss their deadlines.

  Important features of this extension are that it can be easily adapted to the INCA Server to schedule new tasks and that the overhead and the response time of the new tasks can be controlled by limiting the number of iterations of the TB* algorithm.
- **Scheduling aperiodic tasks.** As discussed above, aperiodic tasks (characterized as tasks with only one period) can be scheduled in the INCA Server by using the TB Server. Note that the TB algorithm is used only for *soft aperiodic tasks*. In the case of *hard aperiodic tasks* we can resort to the *Aperiodic Server* [25] and change its capacity (according to the load of the system) at every new arrival and departure of a task. This way, we may respond faster when running the INCA server, while still maintaining the schedulability of the set of tasks in the system.
- **Scheduling tasks with resource requirements.** Tasks with resource sharing constraints can be introduced easily into our framework using the Stack Resource Policy [3]. This protocol has several interesting properties that make it suitable to be included into our INCA framework. For example, it applies to a dynamic scheduling environment, prevents deadlocks, bounds the maximum blocking times of tasks, reduces the number of context switches, and, most importantly, its implementation is straightforward and its overhead is low. This protocol has been enhanced using the TB and the TB* server to handle resource requirements for soft and hard tasks [17]. The analysis described in [17] can be used to perform online guarantees for reserving a bandwidth to serve new tasks in the INCA Server. However, there are several yet unresolved issues, for example, the SRP will influence the temporal behavior of the applications and, therefore, this will influence the UBT. One of our avenues for future work is precisely how to design algorithms that take advantage of the interaction between the SRP and UBT.

8 Conclusion

In this paper, the problem of scheduling an overloaded real-time system has been studied. As observed by different research studies [12], [7], [13], [16], 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 online, 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 in 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 if an optimal solution is sought. However, we have shown that, in practice, only a few stages need to be executed for achieving near-optimal solutions. An important feature of the incremental algorithm is that its runtime overhead and the quality of the solutions are parameters that can be controlled online. 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. To extend the applicability of the INCA server, we proposed a more general feasibility test, where mandatory parts could also cause overloads. Also, additional criteria was discussed for improving the response time of new tasks. Finally, aperiodic tasks and tasks with resource sharing constraints were introduced into our framework using the Total Bandwidth Server [23] and the Stack Resource Policy [3], respectively.

APPENDIX

PROOF OF RESULT 1

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 $CU_N([a_n + I_1, a_n + I_{1,2}]) = (I_{1,2} - I_1)(1 - \alpha_0)$ and $\alpha_{k+1} \leq \alpha_k$. Further, if $a_{n+1} > a_n + I_1 + I_2$ (arrival $a_{n+1} = y$ in Fig. 7), then INCA-2 and Non-INCA-2 produce the same utilization, $(1 - \alpha_2)$, for any time later than $a_n + I_1 + I_2$; thus, $CU([a_n, a_{n+1}]) \geq CU_N([a_n, a_{n+1}])$ holds.

Now, assume that $a_{n+1} = (a_n + I_{1,2} + t)$ (arrival $a_{n+1} = x$ in Fig. 7), where $0 \leq t \leq (I_1 + I_2 - I_{1,2})$. In this case,

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

using the values of I_1, I_2, and $I_{1,2}$ from Section 6.2 and assuming $t_0 = 0$, we get,

$$CU([a_n, a_{n+1}]) = R + \left(\frac{\phi_1 + \phi_2}{\alpha_0} - \frac{\phi_1}{\alpha_0} + t\right)(1 - \alpha_1) \Rightarrow$$
$$CU([a_n, a_{n+1}]) = R + \left(\frac{\phi_2}{\alpha_0} + t\right)(1 - \alpha_1),$$

where $R = \frac{\phi_1}{\alpha_0}(1 - \alpha_0)$ and

$$CU_N([a_n, a_{n+1}]) = \left(\frac{\phi_1 + \phi_2}{\alpha_0}\right)(1 - \alpha_0) + t(1 - \alpha_2) \Rightarrow$$
$$CU_N([a_n, a_{n+1}]) = R + \frac{\phi_2}{\alpha_0}(1 - \alpha_0) + t(1 - \alpha_2).$$

Assuming that $CU([a_n, a_{n+1}]) \geq CU_N([a_n, a_{n+1}])$, we get

$$R + \left(\frac{\phi_2}{\alpha_0} + t\right)(1 - \alpha_1) \geq R + \frac{\phi_2}{\alpha_0}(1 - \alpha_0) + t(1 - \alpha_2) \Rightarrow$$
$$\left(\frac{\phi_2}{\alpha_0} + t\right)(1 - \alpha_1) \geq \frac{\phi_2}{\alpha_0}(1 - \alpha_0) + t(1 - \alpha_2).$$

We are interested in verifying the above condition for all possible values of t, $0 \leq t \leq t_{max}$, where t_{max} is the maximum possible value of t and $t_{max} = (I_1 + I_2 - I_{1,2}) = (\frac{\phi_1}{\alpha_0} + \frac{\phi_2}{\alpha_1} - \frac{\phi_1 + \phi_2}{\alpha_0}) = (\frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0})$. Substituting t in the above equation, we get

$$\left[\frac{\phi_2}{\alpha_0} + \lambda\left(\frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0}\right)\right](1 - \alpha_1) \geq$$
$$\frac{\phi_2}{\alpha_0}(1 - \alpha_0) + \lambda\left(\frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0}\right)(1 - \alpha_2),$$

where $0 \leq \lambda \leq 1$. Therefore,

$$\left(\frac{\phi_2}{\alpha_0} + \frac{\lambda\phi_2}{\alpha_1} - \frac{\lambda\phi_2}{\alpha_0} - \frac{\phi_2\alpha_1}{\alpha_0} + \frac{\lambda\phi_2\alpha_1}{\alpha_0} - \lambda\phi_2\right) \geq$$
$$\frac{\phi_2}{\alpha_0} - \phi_2 + \lambda\left(\frac{\phi_2}{\alpha_1} - \frac{\phi_2}{\alpha_0} - \frac{\phi_2\alpha_2}{\alpha_1} + \frac{\phi_2\alpha_2}{\alpha_0}\right) \Rightarrow$$
$$\left(\frac{\lambda\phi_2\alpha_1}{\alpha_0} - \frac{\phi_2\alpha_1}{\alpha_0} - (\lambda\phi_2)\right) \geq$$
$$-\phi_2 + \lambda\left(\frac{\phi_2\alpha_2}{\alpha_0} - \frac{\phi_2\alpha_2}{\alpha_1}\right) \Rightarrow$$
$$\lambda\phi_2\left(\frac{\alpha_1 - \alpha_0}{\alpha_0}\right) + \phi_2\left(\frac{\alpha_0 - \alpha_1}{\alpha_0}\right) \geq \lambda\phi_2\left(\frac{\alpha_2}{\alpha_0} - \frac{\alpha_2}{\alpha_1}\right).$$

Note that both the first term on the left side and the term on the right side of the condition may be negative numbers since $\alpha_0 \geq \alpha_1$.

Rearranging terms on the condition, we get

$$\lambda\phi_2\left(\frac{\alpha_2}{\alpha_1} - \frac{\alpha_2}{\alpha_0}\right) + \phi_2\left(\frac{\alpha_0 - \alpha_1}{\alpha_0}\right) \geq \lambda\phi_2\left(\frac{\alpha_0 - \alpha_1}{\alpha_0}\right).$$

Given that $\alpha_k \geq \alpha_{k+1}$, $\lambda\phi_2(\frac{\alpha_2}{\alpha_1} - \frac{\alpha_2}{\alpha_0}) \geq 0$, and $\phi_2(\frac{\alpha_0 - \alpha_1}{\alpha_0}) \geq \lambda\phi_2(\frac{\alpha_0 - \alpha_1}{\alpha_0})$, we conclude that

$$CU([a_n, a_{n+1}]) \geq CU_N([a_n, a_{n+1}]).$$

□

ACKNOWLEDGMENTS

A shorter version of this paper was presented at the 21st Real-Time Systems Symposium (RTSS '00). This work has been supported by the US Defense Advanced Research Projects Agency through the FORTS project (Contract DABT63-96-C-0044).

REFERENCES

[1] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," *Proc. IEEE Real Time Systems Symp.*, Dec. 1998.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Optimal Reward-Based Scheduling for Periodic Real-Time Tasks," *IEEE Trans. Computers*, vol. 50, no. 2, pp. 111-130, Feb. 2001.

[3] T. Baker, "Stack Based Scheduling of Real-Time Processes," *J. Real-Time Systems*, vol. 3, no. 1, pp. 284-292, 1993.

[4] P. Binns, "Incremental Rate Monotonic Scheduling for Incremental Control System Performance," *Proc. IEEE Real Time Technology and Applications Symp.*, June 1997.

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

[6] A. Burns and D. Prasad, "Value-Based Scheduling of Flexible Real-Time Systems for Intelligent Autonomous Vehicle Control," *Proc. Third IFAC Symp. Intelligent Autonomous Vehicles*, Mar. 1998.

[7] G.C. Buttazzo, "Red: A Robust Earliest Deadline Scheduling Algorithm," *Proc. Third Int'l Workshop Responsive Computing Systems*, Dec. 1998.

[8] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic Task Model for Adaptive Rate Contro,l" *Proc. IEEE Real Time Systems Symp.* Dec. 1998.
[9] G.C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic, 1997.
[10] H. Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Trans. Software Eng.,* Oct. 1989.
[11] M. Hamdaoui and P. Ramanathan, "A Dynamic Priority Assignment Technique for Streams with (m,k)-Firm Deadlines," *IEEE Trans. Computers,* vol. 44, no. 12, pp. 1443-1451, Dec. 1995.
[12] S. Hwang, C.M. Chen, and A.K. Agrawala, "Scheduling an Overloaded Real-Time System," *Proc. IEEE Conf. Computers and Comm.,* 1996.
[13] R.K. Clark, E.D. Jensen, and F.D. Reynolds, "An Architectural Overview of the Alpha Real-Time Distributed Kernel," *Proc. USENIX Workshop Microkernels and Other Kernel Architectures,* pp. 200-208, 1993.
[14] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *Computer J.,* pp. 390-395, Oct. 1986.
[15] G. Koren and D. Shasha, "Skip-Over: Algorithms and Complexity for Overloaded Real-Time Systems," *Proc. IEEE Real Time Systems Symp.,* Dec. 1995.
[16] G. Koren and D. Shasha, "D-Over: An Optimal Scheduling Algorithm for Overloaded Real-Time Systems," *Proc. IEEE Real Time Systems Symp.,* 1992.
[17] G. Lipari and G. Buttazzo, "Schedulability Analysis of Periodic and Aperiodic Tasks with Resource Constraints," *J. Systems Architecture,* vol. 46, pp. 327-338, Jan. 2000.
[18] C.L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real-Time Environments," *J. ACM,* vol. 20, no. 1, pp. 46-61, Jan. 1973.
[19] J.W. Liu, W.-K. Shih, K.-Y. Lin, and R. Bettati, "Imprecise Computations," *Proc. IEEE,* vol. 82, no. 1, Jan. 1994.
[20] C.D. Locke, "Best-Effort Decision Making for Real-Time Scheduling," PhD thesis, Computer Science Dept., Carnegie Mellon Univ., 1986.
[21] S. Martello and P. Toth, *Knapsack Problems.* Wiley, 1990.
[22] S. Sahni, "Approximate Algorithms for the 0/1 Knapsack Problem," *J. ACM,* Jan. 1975.
[23] M. Spuri and G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Proc. IEEE Real Time Systems Symp.,* Dec. 1996.
[24] S. Zilberstein and A. Mouaddib, "Optimal Scheduling of Progressive Processing Tasks," *Int'l J. Approximate Processing,* vol. 25, pp. 169-186, 2000.
[25] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *J. Real-Time Systems,* vol. 1, pp. 27-60, 1989.

Pedro Mejía-Alvarez received the BS degree in computer systems engineering from ITESM, Queretaro, Mexico, in 1985 and the PhD degree in informatics from the Universidad Politecnica de Madrid, Spain, in 1995. He has been an asisistant professor in the Seccion de Computacion CINVESTAV-IPN Mexico since 1997. In 1999, he held a research faculty position in the Department of Computer Science at the University of Pittsburgh and, in 2000, a visiting assistant professor position in the Department of Information Sciences and Telecommunications at the University of Pittsburgh. He is a fellow of the National System of Researchers of Mexico (SNI). His main research interests are real-time systems scheduling, low-power computing, adaptive fault tolerance, and software engineering. He is member of the IEEE Computer Society.

Rami Melhem received the BE degree in electrical engineering from Cairo University in 1976, the MA degree in mathematics and the MS degree in computer science from the University of Pittsburgh in 1981, and the PhD degree in computer science from the University of Pittsburgh in 1983. He was an assistant professor at Purdue University prior to joining the faculty of the University of Pittsburgh in 1986, where he is currently a professor of computer science and electrical engineering and the chair of the Computer Science Department. His research interests include real-time and fault-tolerant systems, optical interconnection networks, high performance computing, and parallel computer architectures. He has served on program committees of numerous conferences and workshops and was the general chair for the Third International Conference on Massively Parallel Processing Using Optical Interconnections. He was on the editorial board of the *IEEE Transactions on Computers* and served on the advisory boards of the IEEE Technical Committees on Parallel Processing and on Computer Architecture. He is the editor for the Plenum Book Series on Computer Science and is on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*. He is a fellow of the IEEE and a member of the ACM.

Daniel Mossé received the BS degree in mathematics from the University of Brasilia in 1986 and the MS and PhD degrees in computer science from the University of Maryland in 1990 and 1993, respectively. He joined the faculty of the University of Pittsburgh in 1992, where he is currently an associate professor. His research interests include fault-tolerant and real-time systems, as well as networking. The major thrust of his research in the new millenium is power-aware computing and security. He has served on program committees for all major IEEE-sponsored real-time related conferences and as and program and general chairs for RTAS and RT Education Workshop. Typically funded by the US National Science Foundation and US Defense Advanced Research Projects Agency, his projects combine theoretical results and implementations. He on the editorial board of the *IEEE Transactions on Computers* and is a member of the IEEE Computer Society and of the ACM.

Hakan Aydin received the BSc and MSc degrees in control and computer engineering from Istanbul Technical University in 1991 and 1994, respectively. In 1996, he joined the University of Pittsburgh Computer Science Department, where he received the PhD degree in 2001. He is currently an assistant professor at George Mason University, Computer Science Department. He served on the program committee of the IEEE Real-Time Technology and Applications Symposium (RTAS) in 2001 and 2003. His research interests include real-time systems, low-power computing, and fault tolerance.