Technical Report

Capacity Sharing and Stealing in Server-based Real-Time Systems

Luís Nogueira

Luís Miguel Pinho

TR-051205

Version: 1.0

Date: December 2005

Capacity Sharing and Stealing in Server-based Real-Time Systems

Luís NOGUEIRA, Luis PINHO

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: {luis, lpinho}@dei.isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

In this paper we introduce an algorithm that supports the coexistence of guaranteed and best-effort bandwidth servers in a dynamic scheduling environment. In addition to being able to efficiently reclaim residual capacities, originated by early completions, an overloaded active server can also steal future capacities of inactive best-effort servers. The proposed dynamic budget accounting mechanism ensures that at a particular time, the currently executing server is using a residual capacity, its own capacity or is stealing some future capacity, eliminating the need of additional server states or unbounded queues. The server to which the budget accounting is going to be performed is dynamically determined at the time instant when a capacity is needed. The paper describes and evaluates the proposed scheduling algorithm, stating that it can efficiently reduce the mean tardiness of periodic jobs. The achieved results become even more significant when tasks' computation times have a large variance.

Capacity Sharing and Stealing in Dynamic Server-based Real-Time Systems

Luís Nogueira, Luís Miguel Pinho
IPP Hurray Research Group
Polythecnic Institute of Porto, Portugal
{luis,lpinho}@dei.isep.ipp.pt

Abstract

A dynamic scheduler that supports the coexistence of guaranteed and non-guaranteed bandwidth servers is proposed. Overloads are handled by an efficient reclaiming of residual capacities originated by early completions as well as by allowing reserved capacity stealing of non-guaranteed bandwidth servers. The proposed dynamic budget accounting mechanism ensures that at a particular time the currently executing server is using a residual capacity, its own capacity or is stealing some reserved capacity, eliminating the need of additional server states or unbounded queues. The server to which the budget accounting is going to be performed is dynamically determined at the time instant when a capacity is needed. This paper describes and evaluates the proposed scheduling algorithm, showing that it can efficiently reduce the mean tardiness of periodic jobs. The achieved results become even more significant when tasks' computation times have a large variance.

1 Introduction

It can be easily observed that in many real-time applications the worst-case execution time (WCET) of some tasks is rare and much longer than the average case. Reserving resources based on a worst-case feasibility analysis will drastically reduce resource utilisation, causing a severe system's performance degradation when compared to a soft guarantee based on average execution times. Furthermore, it is increasingly difficult to compute WCET bounds in modern hardware without introducing excessive pessimism [7].

Several authors have already proposed algorithms that achieve guaranteed service and inter-task isolation, using mean execution times (see for example [2, 9, 10, 11, 12, 23]). By isolating the effects of task overloads, hard tasks can be guaranteed using classical schedulability analysis [16].

Abeni and Buttazo proposed the Constant Bandwidth Server (CBS) [2] to efficiently handle soft real-time requests with a variable or unknown execution behaviour under the EDF [16] scheduling policy, achieving isolation among tasks, through a resource reservation mechanism which bounds the effects of tasks overruns. Several CBS-based algorithms were proposed

next. For example, the works reported in [15, 5, 17, 14] improve CBS's ability to reclaim residual capacities to handle jobs' overloads.

Also in fixed priority scheduling, a resource allocation mechanism based on servers is often advocated to improve the quality of soft tasks' behaviour, whilst guaranteeing the deadlines of hard tasks.

However, new highly dynamic systems introduce new requirements and opportunities to an efficient overload control, not completely handled by the existing scheduling algorithms. It may be important to loose isolation between some servers by donating reserved capacities to other servers to increase the chance of reducing the mean tardiness of periodic services.

An example of such a system was presented in [18], where previously guaranteed users' services coexist with the arrival of new service requests for coalition formation and service proposal formulation for a cooperative execution of resource intensive applications. It is desirable to be able to process the framework's management algorithms at a certain minimum rate. However, overloaded servers that deal with users' services should be able to use capacities reserved for those algorithms, giving priority to previously guaranteed services with respect to new service requests that eventually would bring more workload to the system. A significant reduction in the mean tardiness of periodic users' services can be achieved by stealing reserved capacities for the framework's management.

We consider the coexistence of *non-isolated* and *isolated* servers in the same system. For an isolated server an amount Q of a resource is ensured to be available every period T. An inactive non-isolated server, however, can have some or all of its reserved capacity stolen by active overloaded servers.

Non-isolated servers are motivated by the use of imprecise computation models, such as the anytime algorithms for the framework's management [19]. Anytime algorithms provide a useful scheme for integrating complex and unbounded computations into real-time systems.

This paper introduces and evaluates the Capacity Sharing and Stealing (CSS) algorithm that in the presence of isolated and non-isolated bandwidth servers can: (i) achieve isolation among tasks; (ii) perform efficient reclaiming of unused computation time, exploiting early completions; (iii) reduce the number of deadline postponements, assigning all excess capacity to the currently executing server; (iv) steal capacity from inactive non-isolated servers in overload situations, reducing the mean tardiness of isolated servers.

The proposed dynamic budget accounting mechanism considers residual capacities, generated by early completions, as well as inactive non-isolated capacities, as a common resource that can be shared by active servers. The server to which the budget accounting is going to be performed is dynamically determined at the time instant when a capacity is needed.

2 Related work

An efficient reclaiming of unused bandwidth has been previously addressed by several authors. Some of the most relevant works in fixed and dynamic priority scheduling are discussed in the next paragraphs.

Optimal fixed priority capacity stealing algorithms have been reported in [13, 8] to minimise soft tasks response times whilst guaranteeing that the deadlines of hard tasks are met. However, they present some drawbacks. The algorithm presented in [13] relies on a pre-computed table that define the residual capacity present on each invocation of a hard task. In contrast, [8] calculates the available residual capacity at run time, but the execution time overhead introduced by the optimal dynamic approach is infeasible in practice.

In [4] Bernat and Burns propose a capacity sharing algorithm for enhancing soft aperiodic responsiveness in fixed priority scheduling. Each server can consume capacity of other servers to advance the execution of the served task in overload situations. As such, a server can receive less bandwidth than expected, loosing isolation among served tasks.

The HisReWri fixed priority scheduler [3] identifies those tasks that did execute when a hard task frees some of its maximum allocation budget and retrospectively assigns their execution times to the hard task. If there was residual capacity available, tasks' budgets are replenished by the amount of residual capacities they consumed. As execution time is retrospectively reallocated the authors describe the protocol as history rewriting.

In dynamic scheduling, CBS's ability to manage residual capacities originated by early completions has been extended in several algorithms.

GRUB [15] uses excess capacity to reduce the number of tasks' preemptions, assigning all the excess bandwidth to the currently executing job and postponing the deadline before starting a new job, regardless of the current value of the server's budget. Although a greedy reclamation policy is used, excess capacity always tends to be distributed in a fair manner among needed servers across the time line.

A critical parameter of this approach is the time granularity used in the algorithm, since a small period reduces the scheduling error, but increases the overhead due to context switches [5]. While GRUB starts executing early arrivals with server's current budget, but postponing its deadline, we want to reduce the number of deadline shifts, executing periodic tasks with a more stable frequency. We state that those early arrived jobs should only begin their execution in the expected period of arrival.

CASH [5] uses a global queue of residual capacities, originated by early completions, ordered by deadline. Each server consumes available residual capacities before using its own budget. The main benefit of this approach is to reduce the number of deadline shifts, executing periodic tasks with more stable frequencies.

While CASH presents an efficient reclaiming of unused computation times and achieves temporal isolation on tasks' execution, it may not schedule tasks as expected, since it immediately recharges servers' budget without suspending the tasks as in CBS [17]. Also, because the number of elements in the queue of unused resources can grow beyond any bound, the CASH algorithm poses challenges to its formal specification and analysis [24]. An improvement to CASH's residual bandwidth reclaiming and the its ability to work in the presence of shared resources has been recently reported in [6]. We explicitly set a recharging time for each server and eliminate the need of a queue of residual capacities by using a dynamic

budget accounting mechanism.

IRIS [17] presents a work-conserving mechanism that guarantees a minimum budget in a fixed interval of time. The authors identify problems in CBS when scheduling acyclic tasks (tasks that are continuously active for large intervals of time) and propose to suspend each task's replenishment until a specific time, implementing a hard reservation technique [22].

IRIS fairly distributes residual capacities among needed servers only after they have consumed their own budgets and still have work to do. Our work focuses on minimising the number of deadline shifts and the mean tardiness of periodic jobs by consuming residual capacities as early, and not necessarily as fairly, as possible. We carefully considered the fairness issue. The increased computational complexity of fairly assign residual capacities to all active servers and the fact that fairly distributing residual capacities to a large number of servers can originate a situation where no enough excess capacity is provided to any one to avoid a deadline miss, lead us to assign all residual bandwidth to the currently executing overloaded server.

BACKSLASH [14] proposes to retroactively allocate residual capacities to tasks that have previously borrowed their current resource reservations to complete previous overloaded jobs, using an EDF version of the mechanism implemented in HisReWri. At every capacity exhaustion, servers' budget is immediately recharged and their deadlines extended as in CBS. However, a task that borrows from a future job remains eligible to residual capacity reclaiming with the priority of its previous deadline.

Allowing a task to use resources allocated to the next job of the same task, while not jeopardises the schedulability of other tasks, may cause future jobs to miss their deadlines by larger amounts. This is specially true in tasks whose actual execution requirements, even if only temporarily, do not vary around an average-case estimate. BACKSLASH can be outperformed by an algorithm that do not borrows from future resources, when considering the mean tardiness of a set of periodic tasks on higher system loads [14]. Rather than borrowing from future resource reservations of the same task to handle task's overload, we propose to steal reserved capacities of inactive non-isolated servers and suspend each task's replenishment until its deadline, not lowering task's priority either for execution scheduling and residual capacity reclaiming.

Our work integrates and extends some of the best principles of previous approaches to efficiently handle soft-tasks' overloads by making additional capacity available from two sources: (i) residual capacity allocated but unused when jobs complete in less than their budgeted execution time; (ii) stealing capacity from inactive non-isolated servers used to schedule best-effort jobs. None of the discussed algorithms tries to minimise the mean tardiness of periodic jobs as our do.

3 System model and notation

We consider the existence of a set $\tau = \tau_h \cup \tau_s \cup \tau_n$ of hard, soft and non-real time tasks in the system.

Each task τ_i consists of a sequence of jobs $\{J_{i,j}, \ldots, J_{i,n}\}$, such that $\forall_{i,j} a_{i,j} < a_{i,j+1}$, where $a_{i,j}$ is the arrival time of job $J_{i,j}$. Each job has an execution requirement of $e_{i,j}$ time units.

Each hard task is characterised by a tuple $\tau_i = (C_i, D_i, T_i)$, where C_i is the worst-case execution time, D_i is the deadline of the task, and T_i is the minimum inter-arrival time between successive jobs, so that $a_{i,j+1} \geq a_{i,j} + T_i$. In our model, the absolute deadline of each hard job $J_{i,j}$ is implicitly set to $d_{i,k} = a_{i,k} + T_i$.

Soft tasks are characterised by average values. The arrival time $a_{i,k}$ of a particular job $J_{i,k}$ is only revealed during execution, and the exact execution requirements $e_{i,j}$ can only be determined by actually executing job $J_{i,j}$ to completion. The soft absolute deadline is set to $d_{i,k} = a_{i,k} + T_i$. Soft deadline misses can decrease provided QoS, but do not cause critical system faults.

Since soft tasks cannot be guaranteed to complete execution before their deadlines, our goal is to minimise the mean tardiness, without jeopardising the guarantees of isolated servers. The tardiness $E_{i,j}$ of a job $J_{i,j}$ is defined as $E_{i,k} = max\{0, f_{i,j} - d_{i,j}\}$, where is the finishing time of job $J_{i,j}$.

A non-real time task is a task without time constraints. Non-real time tasks are scheduled when possible, preserving the guarantees of real-time tasks.

Each hard or soft task τ_i is associated to a server S_i that is characterised by a pair (Q_i, T_i), where Q_i is the maximum capacity and T_i is the server period. Each server S_i maintains a current capacity c_i, a server deadline d_i and a recharging time r_i. The fraction of the CPU reserved to server S_i (the utilisation factor) is given by $U_i = \frac{Q_i}{T_i}$.

Our scheduling scheme is based on the EDF algorithm. Each server receives a job for computation at time $a_{i,j}$ and serves it assigning a dynamic absolute deadline $d_{i,k} = a_{i,k} + T_i$.

We consider the coexistence of non-isolated servers S^N and isolated servers S^I in the system. Active (isolated or non-isolated) overloaded servers can steal inactive non-isolated capacities until their respective deadlines.

At time t each server in the system can be in one of the following states:

Active the served task is ready to execute, or is executing using a residual capacity, its own capacity or stealing capacity from a inactive non-isolated server, or the server is supplying its residual capacity to the currently executing server.

Inactive the server has no pending jobs and is not supplying any residual capacity to the currently executing server.

All servers begin in the Inactive state. State transitions are determined by the arrival of a new job or by the nonexistence of pending jobs at replenishment time r_i, as follows:

- **Inactive** → **Active**: a served job instance arrives at time $a_{i,k}$
- **Active** → **Inactive**: at replenishment time r_i there is no pending job to execute

Note that the transition to the Inactive state only occurs at replenishment time. On an early completion, a server S_i remains active, supplying its residual capacity until its deadline. This eliminates the need of a global queue of residual capacities,

as used in [5], or an additional server state, as used in [17, 15]. We state that if a server is supplying residual capacity, it is contributing to a better global system's performance and, as such, can be considered as being active.

At each instant in time the proposed algorithm selects for execution, from the set of servers in Active state A, the server S_i with earliest deadline d_i and with pending work to execute. Server S_i is such that $\exists S_x \in A : min(d_x), A \neq \emptyset$.

If there is not any server in Active state, then the processor is idle, or executing non-real time tasks.

4 Capacity sharing and stealing

Our approach is based upon the notion of reserving a fraction of the processor bandwidth for each server to achieve isolation among users' tasks. We also want to reclaim, as much as possible, the unused computation time originated by early completions, and give it to the currently executing server.

Since the execution time of each job is not known beforehand, it makes sense to devote as much excess capacity as possible to the currently executing server, giving it a chance to complete without deadline postponements, rather than distribute this capacity (usually in proportion of servers' bandwidths) among a large number of servers, without providing enough excess capacity to any of the servers to avoid deadline postponements.

Each server starts by using available residual capacities, according to an EDF policy, before using its own capacity, aiming at reducing the number of deadline shifts. When a job completes, any remaining capacity is immediately available to the next server to be scheduled. CSS preemptively allocates residual capacities as soon as they are available to the earliest deadline server with the priority of the donating server, maximising its likelihood of using those residual capacities to meet its deadline, as opposed to the approach of only start consuming residual capacities on a budget exhaustion.

When available residual capacities and a server's own capacity were not enough to handle the execution requirements of the current job, we propose to steal future reserved capacities of other servers. To do so, we must be very careful to not end up using any of the future capacity of currently inactive isolated servers. Remember that we do not know when those servers will become active, since we have no idea of the arrival times of new jobs, and we must guarantee the reserved capacity to an isolated server. As such, overloaded servers can only steal inactive non-isolated capacities.

In order to allow an overloaded active server to steal inactive non-isolated capacities and continue its execution after its own capacity exhaustion, its current capacity and deadline cannot be automatically updated as in CBS, CASH and BACKSLASH, for example. As such, we suspend the capacity recharging and deadline update until a specific time.

When a server consumes some capacity amount, either residual, its own, or a stolen capacity, budget accounting must be performed. The proposed dynamic budget accounting mechanism ensures that at time t, the currently executing server S_i is using a residual capacity c_r, originated by an early completion of another active server, its own capacity c_i or is stealing capacity c_s from an inactive non-isolated server.

These principles are detailed in the next sections.

4.1 Dynamic budget accounting

CSS requires three additional parameters to characterise each server when compared to the original CBS algorithm. Each server has a type (isolated or non-isolated), a pointer to a server from which the budget accounting is going to be performed and a specific recharging time. However, the proposed dynamic budget accounting mechanism eliminates the need of additional server states and extra queues to manage residual and stolen capacities, reducing the needed overhead when compared to other algorithms that improve CBS.

Intuitively, each servers' deadline is a measure of its priority under EDF scheduling. The proposed dynamic budget accounting protocol follows these rules: (i) whenever a server is selected to be the running server, if there are high priority servers with residual capacities greater than zero, the server consumes available residual capacities until their exhaustion or job completion (whatever comes first); (ii) if all residual capacities are exhausted, and there are still pending work to do, the server points to itself and consumes its own capacity; (iii) if all consumed (residual and own) capacities were not enough to complete the job, the server steals high priority available capacities of inactive non-isolated servers, until its deadline or job completion (whatever comes first); (iv) if the currently executing server is connected to another server and it is preempted, the former is immediately disconnected from the later and points to itself; (v) on job's completion the server points to itself.

The used capacity is decremented from the reserved capacity of the pointed server. Note that at a particular time t there is only one server pointing to another server.

4.2 Residual capacity reclaiming

Constant bandwidth schedulers reserve a given bandwidth to each server, achieving isolation among tasks, even in the presence of jobs' overloads. However, we also want to reclaim, as much as possible, the unused computation time left by other servers, maximising the probability of an overloaded server complete its overrun without introducing long delays, executing periodic tasks with more stable frequencies.

When a server S_i completes a job, and its remaining capacity c_i is greater than zero, it can immediately be used by others, until the currently assigned S_i's deadline, $d_{i,k}$. If there are no pending jobs waiting to execute, S_i's residual capacity c_r is updated to the current value of remaining server's capacity c_i and c_i is set to zero. The server is kept in Active state, maintaining its deadline $d_{i,k}$ and supplying its residual capacity to other servers.

Whenever a new server is scheduled for execution it first tries to use residual capacities, released by early completions of other active servers, with deadlines less than or equal to the one assigned to the served job.

Let A be the set of all active servers. The set of active servers A_r eligible for residual capacity reclaiming is given by $A_r = \{S_r | S_r \in A, d_r \leq d_{i,k}, c_r > 0\}$, where d_r is the current deadline of early completed jobs.

The consumed residual capacity c_r is selected from the earliest deadline active server S_r from the set of eligible servers

A_r. S_r is defined as $\exists^1 S_r \in A_r : min_{d_r}(A_r), A_r \neq \emptyset$.

Server S_i updates its pointer to S_r and consumes its residual capacity c_r, left by an early completion of S_r. S_i executes with the deadline d_r of the pointed server S_r.

Whenever the residual capacity is exhausted, and there is pending work to do, S_i disconnects from S_r and selects the next available server S'_r (if any).

All available residual capacities, until S_i's deadline, are then greedily assigned to the currently executing server, minimising deadline postponements and number of preemptions [15]. If these residual capacities are exhausted and the job is not complete, the server starts using its own capacity c_i (pointing to itself), either until job's completion or c_i's exhaustion. On a c_i's exhaustion, S_i is also kept in Active state, maintaining its deadline $d_{i,k}$.

The proposed mechanism for residual capacity reclaiming as three main advantages: (i) start consuming residual capacities before server's own budget maximises the probability of those capacities being effectively used by other servers to control overloads before they expire at their deadlines; (ii) using residual capacities with the priority of the donating server preserves system's schedulability; (iii) not immediately postponing the server's deadline on a budget exhaustion improves its probability of actually using any spare capacity that eventually will be released until its deadline to control the current overload, specially when it is not possible to steal inactive non-isolated capacities.

4.3 Non-isolated capacity stealing

CSS considers the coexistence of isolated and non-isolated servers in dynamic scheduling. When a own capacity exhaustion occurs and there is still pending work to do, an overloaded server is able to steal capacity from inactive non-isolated servers.

Let I be the set of all inactive servers. The set of inactive non-isolated servers I_s^N eligible for capacity stealing is given by $I_s^N = \{S_s | S_s \in I, max\{t, d_s\} + T_s < d_{i,k}, c_s > 0\}$, where d_s is the current deadline of each inactive non-isolated server and T_s its period.

Budget accounting will be performed to the earliest deadline inactive non-isolated server S_s from the set of eligible servers I_s^N, and is found by $\exists^1 S_s \in I_s^N : min_{d_s}(I_s^N), I_s^N \neq \emptyset$.

The server connects to the earliest deadline inactive non-isolated server S_s, but continues to run with its own deadline $d_{i,k}$, since we are stealing S_s's capacity and not its priority. If a job arrives for S_s, capacity stealing is interrupted and S_s reaches the Active state with the remaining budget. When $c_s = 0$, and there is pending work to do, the next capacity c'_s is used (if any).

When a server is connected to an inactive non-isolated server and it is preempted, the server is immediately disconnected, maintaining the Active state. Also, whenever a replenishment event occurs on the capacity being stolen, the server is immediately disconnected and stops using that capacity.

The maximum capacity that can be stolen from a inactive non-isolated server S_s is the minimum value between the non-isolated server's remaining capacity and the time left to currently executing server's deadline, and is defined by $c_s = min(c_s, d_i - t)$.

Before stealing any future capacity of an inactive non-isolated server S_s it is necessary to check whether or not an update of S_s's deadline and capacity replenishment are needed since a deadline greater than the actual time implies that some other active overloaded server as already updated S_s's parameters and stolen some portion of the S_s's reserved capacity. If the previously generated absolute deadline d_s of the selected non-isolated server S_s is lower than the actual time ($d_s < t$), a new deadline ($d_s = t + T_s$) is generated and server's capacity is recharged to the maximum value ($c_s = Q_s$). Otherwise, the currently executing server steals capacity c_s using current values. In either case, S_s is kept in the Inactive state.

4.4 Specific replenishment time

CSS presents a major difference with respect to CBS, CASH and BACKSLASH on a server's budget exhaustion. Rather than immediately recharge server's budget and postpone its deadline, CSS explicitly sets a recharging time r_i for each server S_i, implementing a hard reservation technique [22].

Setting the replenishment and deadline update time to server's deadline serves the purposes of the capacity reclaiming and non-isolated capacity stealing approaches presented above. In CSS, on a budget exhaustion the currently executing server keeps its currently assigned deadline, stealing available capacities from inactive non-isolated servers, if any, or using any spare capacity that eventually will be released until then. Active servers' capacity and deadline update time is then set to $r_i = d_i$.

If a server S_i is in Active state and $t = r_i$, the taken action depends on the existence, at time t, of pending jobs to be executed, that is, if there is a job $J_{i,k}$ such that $a_{i,k} \leq t < f_{i,k}$. A server with no pending jobs reaches the Inactive state and no replenishment is necessary. However, for a server with pending jobs, a new deadline is generated to $d_{i,k} = max\{a_{i,k}, d_{i,k-1}\} + T_i$, the server's capacity is replenished to its maximum value ($c_i = Q_i$), the recharging time is set to the server's new deadline ($r_i = d_{i,k}$) and the server's residual capacity is set to zero ($c_r = 0$). A server with pending jobs continues in the Active state.

Note that a residual capacity of an active server is only valid until the server's deadline. If it was not consumed by another server it must be discharged.

Marzario *et al.* [17] state that adding only a fixed recharging time forces the system to be idle even when there are pending jobs and propose to advance the recharging times of all servers waiting for budget replenishment, reclaiming spare time and fairly distributing it among needing servers. We have different a goal: minimise deadline shifts, greedily consuming spare capacities before using each servers' capacity.

Furthermore, advancing recharging times is against our purpose of executing periodic activities with stable frequencies,

and can even be inadequate on a cooperative distributed service execution framework, requiring more effort in service coordination since contracted periods are not being respected. If pending jobs are a consequence of early arrivals, after their respective servers have released residual capacities originated by early completions of previous jobs, executing periodic services with a stable frequency suggests that those early arrived jobs should only begin their execution in the expected period of arrival.

We also consider the existence of non real-time tasks in cooperative environments. Guaranteed cooperative real-time services should only use capacities that were determined by previous admission control, either using residual capacities, their own capacities or stealing inactive non-isolated capacities of other servers. In the presence of early arrivals, real-time tasks should not use more resources' percentage than expected.

5 The CSS algorithm

In this section we formally describe the CSS scheduling algorithm. Each task τ_i is served by a dedicated (isolated or non-isolated) server, characterised by a maximum capacity Q_i and a period T_i. Budget accounting is dynamically performed on the pointed server. Initially all servers are in the Inactive state.

5.1 Definition

1. When a job $J_{i,k}$ arrives at time $a_{i,k}$ for server S_i

   (a) if S_i is *Inactive*, S_i becomes *Active* and it is inserted in the ready queue

   - if $a_{i,k} < d_{i,k}$, the job is served with the last generated server deadline $d_{i,k}$, using the current capacity c_i
   - otherwise, S_i's capacity is recharged to its maximum value $c_i = Q_i$, a new deadline is generated to $d_{i,k} = max\{a_{i,k}, d_{i,k-1}\} + T_i$, recharging time is set to $r_i = d_{i,k}$ and residual capacity is set to $c_r = 0$

   (b) if S_i is *Active* the job is buffered and will be served later

2. When a server S_j in Active state *is selected as the running server*

   (a) S_j connects to the earliest deadline active server with residual capacity $c_r > 0$, such that $d_r \leq d_{j,k}$ (if any) and runs with deadline d_r

   (b) when $c_r = 0$, S_j selects the next earliest deadline capacity c'_r (if any) with deadline $d'_r \leq d_{j,k}$ and updates its deadline to d'_r

   (c) when all available residual capacities c_r are exhausted and there is pending work, S_j uses its own capacity c_j, pointing to itself, and runs with its own deadline $d_{j,k}$

(d) when $c_j = 0$ and there is still pending work to do, the server connects to the inactive non-isolated server S_k^N with the earliest deadline from which it will steal its capacity (if any), such that $t < d_{S_k^N} \leq d_{j,k}$. The server continues to run with its deadline $d_{j,k}$ (not with the deadline of the non-isolated server S_k^N).

(e) when $c_s = 0$ the next capacity c'_s with deadline $t < d_{S_k'^N} \leq d_{j,k}$ is used (if any), until job's completion or $d_{j,k}$

(f) if S_j is using capacity c_s of a non-isolated server S_k^N and it is preempted, then S_j stops using c_s. S_j points to itself and is kept in the Active state

3. Whenever job $J_{i,k}$ executes, the used capacity c_r, c_i or c_s is decreased by the same amount

4. When a job $J_{i,k}$, served by S_i, finishes

   (a) the next pending instance $J_{i,k+1}$ (if any) is executed using the current capacity and deadline

   (b) if there are no pending jobs, the residual capacity is updated with remaining capacity $c_r = c_i$, c_i is set to zero, and S_i keeps in Active state, keeping its recharging time r_i and deadline $d_{i,k}$

5. If the server is in Active state and $t = r_i$

   (a) if there is pending work to do, the capacity is recharged to its maximum value $c_i = Q_i$, the deadline is set to $d_{i,k+1} = max\{a_{i,k+1}, d_{i,k}\} + T_i$, the recharging time is set to $r_i = d_{i,k+1}$, and the residual capacity c_r is set to zero

   (b) otherwise, the server becomes Inactive

6. Whenever the processor becomes idle for an interval of time Δ, the residual capacity c_r with the earliest deadline is decreased by the same amount, until all residual capacities are exhausted

5.2 Example of an efficient overload control

Consider the set of periodic tasks detailed in Table 1. Task τ_1 is served by a non-isolated server, while tasks τ_2 and τ_3 are served by isolated servers, having deadlines equal to their periods and a reserved capacity equal to their average execution times. A task that completes before its average execution time frees residual capacity that is used by other tasks, and a task that needs more than the reserved capacity experiences overload.

Task	Q_i	T_i	Server	Type
τ_1	2	5	S_1	non-isolated
τ_2	4	10	S_2	isolated
τ_3	3	15	S_3	isolated

Table 1. Task set

A possible execution of the task set is presented in Figure 1. When a server is using capacity from another server, either a residual or stolen capacity, a pointer indicates where the budget accounting is being performed.

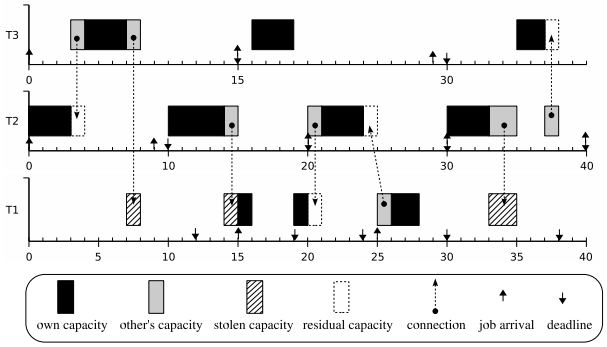

Figure 1. Overloads in the CSS algorithm

At time $t = 3$ task τ_2 has an early completion, and a residual capacity $c_r = 1$, with deadline $d_r = 10$, is available. Server S_3 is scheduled for execution and connects to server S_2, since there is a high priority residual capacity available. Task τ_3 consumes $c_r = 1$ before starting to using its own capacity, at time $t = 4$. At time $t = 7$, an overload is handled by stealing capacity of the inactive non-isolated server S_1. A new deadline for the stolen capacity c_s is set to time $t = 12$.

Note that at time $t = 9$ a new job for task τ_2 arrives but its execution only starts at time $t = 10$, since, as detailed before, advancing execution times is against our purpose of executing periodic activities with stable frequencies.

At time $t = 15$, during an overload being handled by stealing the non-isolated server's capacity, a new job for server S_1 arrives. S_2 stops using S_1's capacity, and the non-isolated server S_1 reaches the active state, keeping the current values for its capacity and deadline. An active non-isolated server behaves as an active isolated server. Since there is not any residual capacity available, S_1 starts consuming its remaining capacity.

At time $t = 16$, server S_1 has no remaining capacity and stops executing. At time $t = 19$, a replenishment of server's capacity occurs and S_1 continues to execute the pending job. Since at time $t = 20$, S_1 completes its job's execution, it frees a residual capacity $c_r = 1$, with deadline $d_r = 24$, that is used by server S_2, before consuming its own capacity at time $t = 21$.

At time $t = 25$ a job for task τ_1 arrives, and the non-isolated server S_1 becomes active. It first consumes the residual capacity $c_r = 1$, with deadline $d_r = 30$, generated at time $t = 24$ by an early completion of task τ_2, before consuming its own capacity[1].

At time $t = 33$ an overload of task τ_2 is first efficiently handled by stealing capacity of the inactive non-isolated server S_1, and then, at time $t = 38$, consuming the available residual capacity generated by an early completion of task τ_3. This is possible, since a server remains in the Active state, until its deadline, even if it has exhausted its capacity.

[1] In the cooperative service execution environment of [20], the algorithm served by S_1 would have more time to run, due to early completions of other servers, and, as such, potentially produce better results in the coalition formation or service proposal formulation process.

This example shows that overloads can be efficiently handled without postponing deadlines, either by using residual capacities and by stealing capacities of inactive non-isolated servers.

6 Theoretical validation

Here we analyse the schedulability of a hybrid set of hard and soft periodic tasks. Each task is scheduled using a dedicated isolated or non-isolated server with parameters (Q_i, T_i), where Q_i is server's maximum capacity and T_i its period.

Each task can use residual capacities, its own capacity or steal inactive non-isolated capacities as well as offer its residual capacity to other tasks.

In [1] it is proved that a server with parameters (Q_i,T_i) cannot occupy a bandwidth greater than $\frac{Q_i}{T_i}$. That is, if $D_{S_i}(t_1, t_2)$ is server's S_i bandwidth demand in the interval $[t_1, t_2]$, it is shown that $\forall t_1, t_2 \in N : t_2 > t_1, D_{S_i}(t_1, t_2) \leq \frac{Q_i}{T_i}(t_2 - t_1)$.

This isolation property allow us to use a bandwidth reservation strategy to allocate a fraction of a resource to a task whose demand is not known a priori. The most important consequence of this property is that soft tasks, characterised by mean values, can be scheduled together with hard tasks, even in the presence of overloads.

We state that our dynamic budget accounting mechanism does not affect schedulability. By assigning each soft task a maximum bandwidth, calculated using the mean execution time and the desired activation period, and isolating the effects of task overloads, a hybrid task set can be guaranteed using the classical Liu and Layland condition [16].

Before proving the schedulability of the proposed algorithm it is important to prove that all generated capacities are exhausted before their respective deadlines.

Lemma 1 *Given a set of isolated capacity based servers, each isolated capacity generated during scheduling is consumed before its deadline or it is discharged at server's deadline*

Proof

Let $a_{i,k}$ denote the instant at which a new job $J_{i,k}$ arrives and server S_i^I is in Inactive state. At $a_{i,k}$, a new capacity $c_i = Q_i$ is generated.

Let $\forall_{i,k}\ d_{i,k} = max\{a_{i,k}, d_{i,k-1}\} + T_i$ be the deadline associated with capacity c_i.

Let $\forall_{i,k}\ r_i = d_{i,k}$ be the replenishment time associated with capacity c_i.

Let $[t, t + \Delta t)$ denote a time interval during which server S_i^I is executing, consuming its own capacity c_i. Consequently, S_i^I has used an amount equal to $c_i' = c_i - \Delta t \geq 0$ of its own capacity during this period. As such, c_i must be decreased to c_i', until its value is equal to zero.

Let $f_{i,k}$ denote the instant that server S_i^I completes job $J_{i,k}$.

Assume that there are no pending jobs. The available capacity $c_i > 0$ can be used by other servers.

Let $c_r = c_i$ be the residual capacity available to other servers. At instant $f_{i,k}$, server's capacity c_i is set to zero and another active server S_j is scheduled for execution.

If the inequality $d_{i,k} \leq d_{j,l}$ holds, server S_j can use residual capacity c_r until deadline $d_{i,k}$ or $c_r = 0$.

Let $[t, t + \Delta t)$ denote a time interval during which server S_j is executing, consuming residual capacity c_r. Consequently, S_j has used an amount equal to $c'_r = c_r - \Delta t \geq 0$ of residual capacity of an active server S_i during this period. As such, c_r must be decreased to c'_r, until its value is equal to zero.

At replenishment time $t = r_i$ any remaining residual capacity c_r of server S_i, not used by another active server, is set to zero.

□

Lemma 2 *Given a set of isolated and non-isolated capacity based servers, each non-isolated capacity generated during scheduling is consumed before its deadline or it is discharged at server's deadline*

Proof

We analyse the following cases: a) a non-isolated capacity is generated when an overloaded active server needs to steal the inactive non-isolated server's capacity; and b) a non-isolated capacity is generated when a new jobs arrives for a inactive non-isolated server.

Case a.

Let $a_{i,k}$ denote the instant when an active overloaded server starts using the non-isolated capacity c_i of the inactive non-isolated server S_i^N.

If the inequality $d_{i,k-1} \leq a_{i,k}$ holds, a new deadline $d_{i,k} = a_{i,k} + T_i$ is generated, the server's capacity c_i is recharged to the maximum value $c_i = Q_i$ and the replenishment time r_i is set to $r_i = d_{i,k}$. Otherwise, the server maintains the current values of c_i, $d_{i,k}$, and r_i.

Let $[t, t + \Delta t)$ denote a time interval during which server S_j is executing, stealing non-isolated capacity c_i of server S_i^N. Consequently, S_j has used an amount equal to $c'_i = c_i - \Delta t \geq 0$ of non-isolated capacity of server S_i^N during this period. As such, c_i must be decreased to c'_i, until its value is equal to zero.

If a new job arrives at $a_{i,k} < a'_{i,k} < r_i$, the inactive server S_i^N reaches the Active state, using the current values of c_i, $d_{i,k}$, and r_i. If at instant $a'_{i,k}$, capacity c_i is being stolen by server S_j, then S_j stops consuming c_i at instant $a'_{i,k}$.

At time $t = r_i$ any remaining capacity c_i of inactive non-isolated servers, not stolen by another active server, is set to zero.

Case b.

Let $a_{i,k}$ denote the instant when a new job $J_{i,k}$ arrives for the inactive non-isolated server S_i^N.

If the inequality $d_{i,k-1} \leq a_{i,k}$ holds, a new deadline $d_{i,k} = a_{i,k} + T_i$ is generated, the server's capacity c_i is recharged to the maximum value $c_i = Q_i$ and the replenishment time r_i is set to $r_i = d_{i,k}$. Otherwise, the server maintains the current values of c_i, $d_{i,k}$, and r_i.

At time $a_{i,k}$ the non-isolated server S_i^N reaches the Active state and behaves like an isolated active server. As such, its capacity c_i is consumed as follows from lemma 1.

□

Theorem 1 *The schedulability of all hard tasks is unaffected under a dynamic budget accounting mechanism that always selects a server with higher or equal priority if and only if*

$$\sum U^{hard} + U^{soft} \leq 1$$

Proof

Let τ_h be a set of periodic hard tasks, where each task is scheduled by a dedicated isolated server with Q_i equal to the WCET of τ_i and T_i equal to the minimum inter arrival time of each job, with total utilisation U^{hard}.

Let τ_s be a set of soft tasks, where each task is scheduled by a group of isolated and non-isolated servers with mean values for Q_i and T_i, with total utilisation U^{soft}.

Lemma 1 states that each isolated capacity is always consumed before its deadline or discharged at server's deadline, hence it follows that each hard task instance has to finish by its deadline. From Lemma 2 we know that each non-isolated capacity is also consumed before its deadline or is discharged at server's deadline.

Since the worst case response time of a hard task is independent of whether a server is executing work for itself or whether its capacity is used by another server, system's schedulability is independent of whether the dynamic budget accounting mechanism is in operation or not. In the worst case, the longest time a server can be connected to another server is bounded by the currently pointed server's capacity and deadline.

□

7 Evaluation

Two sets of experiments have been performed to verify the effectiveness of the CSS algorithm in reducing the mean tardiness of periodic jobs. In the first set, a comparison is made against BACKSLASH and CASH, scheduling only isolated

servers, serving a set of periodic tasks. The second set evaluates the higher flexibility in overload management introduced by CSS with non-isolated capacity stealing.

The results reported in this section were observed from multiple and independent simulation runs, with initial conditions and parameters, but different seeds for the random values used to drive the simulation [21]. The mean values of all generated samples were used to produce the charts. Each simulation ran until $t = 100000$ and was repeated several times to ensure that stable results were obtained.

The mean tardiness of a set of periodic tasks was determined by $\sum_{i=0}^{n} trd_i/n$, where trd_i is the tardiness of task T_i, and n the number of periodic tasks.

7.1 Capacity sharing performance

We compare the performance of CSS when scheduling a set of periodic tasks, served only by isolated servers, against CASH and BACKSLASH, since the three algorithms greedily assign residual capacities as early as possible to the highest priority server. However, they propose different approaches on servers' budget exhaustion with pending jobs. We evaluate the effect of those approaches in lowering the mean tardiness of periodic jobs.

Different sets of 6 periodic servers, with varied capacities Q, ranging from 20 to 50, and period distributions P, ranging from 60 to 600, were used, creating different types of traffic, from short to long deadlines and capacities. The execution time $e_{i,j}$ of each job $J_{i,j}$ varied in the range $[0.7Q_i, 1.4Q_i]$. The purpose of random workloads is to evaluate the performance of each algorithm when tasks' parameters differ in dynamic real-time scenarios.

Figure 2 shows the performance of the three algorithms as a function of system's load, measuring the mean tardiness of periodic tasks under random workloads for different probabilities of jobs' overload.

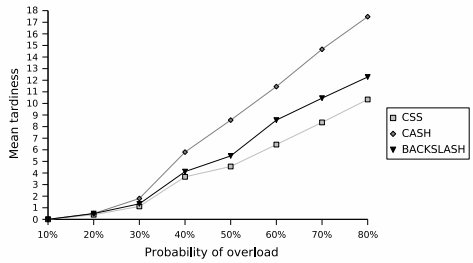

Figure 2. Mean tardiness under random workloads

As expected, all the algorithms perform better when there is more residual capacity available to handle overloads. As the probability of jobs' overload increases, CSS outperforms the other algorithms in lowering the mean tardiness of periodic jobs.

In CASH, once a task's budget is exhausted it is immediately recharged and its deadline extended. As such, its priority

is effectively lowered, lowering its probability of spare capacity reclaiming before missing its deadline. BACKSLASH also immediately updates budget and deadline, but spare capacity reclaiming is done with virtual (original) deadlines. While BACKSLASH and CSS share the same concept of using original deadlines for spare capacity reclaiming, CSS keeps a server in Active state until its deadline, without deadline postponement, effectively improving its probability of actually using any spare capacity that eventually will be released until then, minimising the mean tardiness of periodic jobs. Allowing a task to use resources allocated to the next job of the same task, may cause future jobs to miss their deadlines by larger amounts [14].

In the next section, we demonstrate the higher flexibility achieved by CSS in overload control, by allowing a needed server to steal capacities reserved to non-isolated servers, further minimising the mean tardiness of periodic jobs.

7.2 Capacity sharing and stealing performance

The second set of simulations evaluates the effect of non-isolated capacity stealing on the performance of soft real-time tasks.

The workload consisted of a hybrid set of periodic isolated and non-isolated servers. The maximum capacity and inter-arrival times of the isolated servers were randomly generated in order to achieve a desired processor utilisation factor of $U_{isolated}$. The maximum capacity and period of the non-isolated servers were uniformly distributed in order to obtain an utilisation of $U_{non-isolated} = 1 - U_{isolated}$.

In the first simulation, periodic tasks were served by 1 non-isolated server and 4 isolated servers, with utilisation of $U_{non-isolated} = 0.2$ and $U_{isolated} = 0.8$, as detailed in Table 2. We varied the execution time $e_{i,j}$ of each job $J_{i,j}$ in the range $[0.8Q_i, 1.2Q_i]$.

Server	Q_i	T_i	Type
S_1	2	10	non-isolated
S_2	3	15	isolated
S_3	4	20	isolated
S_4	5	25	isolated
S_5	6	30	isolated

Table 2. Servers' parameters

To evaluate the weight of non-isolated capacity stealing in lowering the mean tardiness of tasks, we varied the probability of arrival of new jobs to non-isolated servers in the range $[1.0, 0.1]$, and measured the mean tardiness of isolated and non-isolated jobs when using both residual capacities and non-isolated capacity stealing or when only using residual capacities.

Figure 3 shows the results. As expected, when overloaded active servers have more opportunities to steal non-isolated capacities, the mean tardiness of jobs lowers accordingly. When only using residual capacities, the mean tardiness is higher as the probability of non-isolated jobs' arrival lowers, since there is less residual capacities available, released by active non-isolated servers. The experiment shows that with low variation in jobs' computation times non-isolated capacity stealing

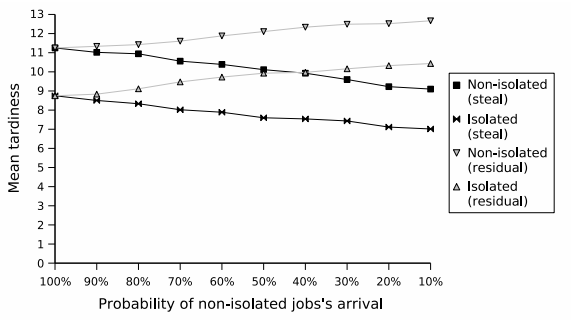

Figure 3. Small variation in execution times

produces better results, although the use of only an efficient residual capacity reclaiming mechanism achieves a slightly poorer performance.

Figure 3 also shows that the performance of non-isolated servers is worse than the achieved performance of isolated servers. Two reasons explain this behaviour. First, when a new job arrives for a inactive non-isolated server, some of its reserved capacity might have been stolen by a needed active overload server. As such, if there is not any residual capacity available at that particular time, the job must be executed with a lower capacity than expected, probably resulting in a deadline miss[2]. Second, there is a big difference on the performance of a server for different configurations of Q_i and T_i, even if they result in the same server utilisation [4]. It is well known that the higher the priority the smaller the capacity available, since there is a tradeoff between capacity size and interference. A server with parameters $(2Q_i, 2T_i)$ has the same utilisation but a higher probability of using residual capacities and steal inactive non-isolated time due to the increased period.

The second simulation has been generated with the same characteristics of the first simulation, except that a greater variance of jobs' execution time was introduced, ranging from $[0.6Q_i, 1.8Q_i]$. Figure 4 clearly shows a perceptibly improved performance of servers when it is possible to steal inactive non-isolated capacities, in the presence of a large variation in jobs' computation times. Severe overloads can be efficiently handled with non-isolated capacity stealing, reducing the mean tardiness of periodic jobs.

8 Conclusion

The work reported in this paper integrates and extends recent advances in dynamic deadline scheduling with resource reservation. Namely, while achieving isolation among tasks, it can efficiently reclaim residual capacities to reduce deadline postponements and steal capacity from inactive non-isolated servers, reducing the mean tardiness of periodic jobs.

The proposed algorithm offers the flexibility to consider the coexistence of guaranteed and best-effort servers in the same system. In systems were some services can appear less frequently, and when they do can be served in a best-effort manner,

[2]This is the main reason for the anytime algorithms proposed in [19], since they will execute within available time. In these simulations, non-isolated jobs completed their randomly generated execution times, evaluating the CSS algorithm in a more generic scenario.

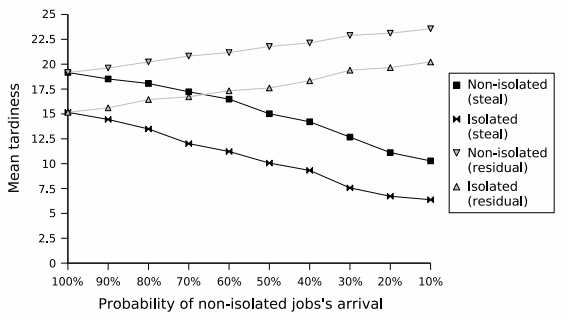

Figure 4. Large variation in execution times

giving priority to overload control of guaranteed services, it has been demonstrated that the proposed algorithm can achieve a higher performance when considering the mean tardiness of periodic guaranteed services. The achieved results become even more significant when tasks' computation times have a large variance.

The proposed dynamic budget accounting mechanism selects, at the time instant when a capacity is needed, the server to which the budget accounting is going to be performed. That server can be any of the system's servers and not necessarily the currently executing server. This eliminates the need of several server states and extra queues to manage residual and stolen capacities.

References

[1] L. Abeni. Server mechanisms for multimedia applications. Technical report, Scuola Superiore S. Anna, 1998.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE RTSS*, page 4, Madrid, Spain, December 1998.

[3] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings of the 24th IEEE Real-Time System Symposium*, pages 396–407, December 2003.

[4] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems*, 22(1-2):49–75, 2002.

[5] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of 21th IEEE RTSS*, pages 295–304, Orlando, Florida, 2000.

[6] M. Caccamo, G. C. Buttazzo, and D. C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, February 2005.

[7] A. Colin and S. M. Petters. Experimental evaluation of code properties for wcet analysis. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 190–199, December 2003.

[8] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings of the 14th Real-Time Systems Symposium*, pages 222–231, 1993.

[9] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE RTSS*, page 308, Washington, DC, USA, 1997.

[10] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems: The International Journal of Time-Critical Computing*, 9(1):31–67, 1995.

[11] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. *Readings in multimedia computing and networking*, pages 491–505, 2001.

[12] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *Proceedings of the 17th IEEE RTSS*, page 206, Washington, DC, USA, 1996.

[13] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 110–123, December 1992.

[14] C. Lin and S. A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 410–421, 2005.

[15] G. Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the 12th ECRTS*, pages 193–200, Stockholm, Sweden, 2000.

[16] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):40–61, 1973.

[17] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Proceedings of the 10th IEEE RTAS*, page 211, Toronto, Canada, 2004.

[18] L. Nogueira and L. M. Pinho. Dynamic qos-aware coalition formation. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, Colorado, April 2005.

[19] L. Nogueira and L. M. Pinho. Iterative refinement approach for qos-aware service configuration. In *Proceedings of the 5th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006) (to appear)*, Braga,Portugal, October 2006.

[20] L. Nogueira and L. M. Pinho. Time-bounded distributed qos-aware service configuration in heterogeneous cooperative environments. Technical report, IPP Hurray Research Group. Available at http://hurray.isep.ipp.pt/, January 2006.

[21] N. Pereira, E. Tovar, B. Batista, L. M. Pinho, and I. Broster. A few what-ifs on using statistical analysis of stochastic simulation runs to extract timeliness properties. In *Proceedings of the PARTES'04 Workshop*, Piza, Italy, 2004.

[22] R. Rajkumar, K. Juvva, A. Molano, , and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[23] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE RTSS*, pages 2–11, San Juan, Puerto Rico, 1994.

[24] P. Ölveczky and M. Caccamo. Formal simulation and analysis of the cash scheduling algorithm in real-time maude. Technical report, Department of Informatics, University of Oslo, October 2005.