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

This paper proposes a dynamic scheduler that supports the coexistence of guaranteed and non-guaranteed bandwidth servers 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. The effectiveness of the proposed approach in reducing the mean tardiness of periodic jobs is demonstrated through extensive simulations. The achieved results become even more significant when tasks' computation times have a large variance.

1 Introduction

It is well known that 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 [6].

Several solutions have already been proposed to maximise resource usage and achieve a guaranteed service and inter-task isolation using average execution estimations and isolating an overload of a particular tasks, not jeopardising the schedulability of other tasks [1, 9, 10, 11, 12, 22]. Other works further increase resource utilisation by reclaiming the unused computation times of some tasks, exploiting early completions [15, 4, 16, 5, 14].

However, a more flexible overload control in highly dynamic real-time systems can be achieved with the combination of guaranteed and best-effort servers and reducing isolation in a controlled fashion in order to donate reserved, but still unused, capacities to currently overloaded servers.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

This paper considers the coexistence of *non-isolated* and *isolated* servers in the same system. For an isolated server, a specific amount of a resource is ensured to be available every period. 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 increasing use of imprecise computation models and anytime algorithms in dynamic real-time systems [2, 19, 18].

The paper introduces and evaluates the Capacity Sharing and Stealing (CSS) algorithm for sets of hard, soft and non-real-time tasks that in the presence of isolated and non-isolated bandwidth servers can: (i) achieve isolation among tasks; (ii) efficiently reclaim unused computation time, exploiting early completions; (iii) reduce the number of deadline postponements, assigning all excess capacity to the currently executing server; and (iv) steal reserved capacity from inactive non-isolated servers in overload situations.

2 Related work

Optimal fixed priority capacity stealing algorithms that minimise soft tasks' response times whilst guaranteeing that the deadlines of hard tasks are met were proposed in [13, 8]. However, they present some drawbacks. [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 [7].

In [3], each server can consume capacity of other servers to advance the execution of the served task in overload situations for enhancing soft aperiodic responsiveness. A server can receive less bandwidth than expected, loosing isolation among served tasks.

The HisReWri algorithm [2] identifies the tasks that did execute when a hard task has released some of its maximum allocation budget and retrospectively assigns their execution times to the hard task. If there is 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 [1] was proposed to efficiently handle soft real-time requests with a variable or unknown execution behaviour under the EDF scheduling policy, achieving isolation among tasks through a resource reservation mechanism which bounds the effects of tasks overruns. Several extensions were proposed next.

GRUB [15] uses excess capacity to reduce the number of tasks' preemptions, assigning all the excess bandwidth to the currently executing server and postponing its 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 [4].

CASH [4] 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, reducing the number of deadline shifts and executing periodic tasks with more stable frequencies. However, CASH may not schedule tasks as expected, since it immediately recharges servers' budget without suspending the tasks [16]. An improvement to CASH's residual bandwidth reclaiming and the ability to work in the presence of shared resources has been recently reported in [5].

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 are 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. The main problem of this approach is that allowing a task to use resources allocated to the next job of the same task may cause future jobs of that task to miss their deadlines by larger amounts. Considering the mean tardiness of a set of periodic tasks on higher system loads, BACKSLASH can be outperformed by an algorithm that do not borrows from future resources [14].

This paper proposes a different approach to handle capacity exhaustion. In particular, in the approaches discussed above, when a server consumes its entire budget, the server budget is recharged and a new deadline is generated and it continues to execute the current job using its new capacity and deadline until the job is completed. The next pending job, if any, is executed using the remaining budget and deadline. We suspend budget recharging and deadline update until a specific time. This enables the overloaded server to steal capacities from inactive non-isolated servers and to eventually use any new residual capacities that is released by some other servers, keeping its current priority.

IRIS [16] identifies problems in CBS when scheduling acyclic tasks (tasks that are continuously active for large intervals of time) and also proposes to suspend each task's replenishment until a specific time, implementing a hard reservation technique [21]. Residual capacity reclaiming is only performed after all the servers had exhausted their reserved capacities. IRIS proposes a work-conserving mechanism that guarantees a minimum budget in a fixed interval of time and fairly distributes residual capacities among needed servers.

We carefully considered this 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. Furthermore, our work focuses on minimising the mean tardiness of guaranteed jobs by consuming residual capacities as early, and not necessarily as fairly, as possible.

3 System model and notation

The paper considers 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 and a set of isolated and non-isolated servers.

Isolated servers have a guaranteed budget until their deadlines while inactive non-isolated servers can have some or all of its reserved capacity stolen by active overloaded servers. Non-isolated servers were thought to serve aperiodic or sporadic tasks that can be served in a best-effort manner.

Both types of servers are characterised by a pair (Q_i, T_i), where Q_i is the reserved 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 $U_i = \frac{Q_i}{T_i}$.

At time t, a server can be in one of the following states:

- **Active:** the served task is (i) ready to execute; (ii) is executing using a residual capacity, the capacity of its server or stealing capacity from a inactive non-isolated server; (iii) or the server is supplying its residual capacity to other servers until its deadline.
- **Inactive:** the server has no pending jobs and is not supplying its residual capacity to other servers. In-

active non-isolated capacities can be stolen by active overloaded servers.

State transitions are determined by the arrival of a new job, capacity exhaustion, or the non-existence of pending jobs at replenishment time. An inactive server reaches the Active state on a job arrival. On the other hand, an active server becomes Inactive if all its reserved capacity is consumed and there are no pending jobs to serve, either while supplying its residual capacity to other servers or exhausting its capacity and finishing its job. Similarly, active servers with no pending jobs at replenishment time become inactive.

On an early completion of its current job, a server remains active supplying its residual capacity until its deadline. If a server is supplying residual capacity, it is contributing to the global system's activity and, as such, can be considered as being active. This eliminates the need of a global queue to manage residual capacities and additional server states.

Each server receives a job for computation at time $a_{i,j}$ serves it assigning a dynamic absolute deadline $d_{i,j} = a_{i,j} + T_i$. The arrival time of a particular job is only revealed during execution, and the exact execution requirements $e_{i,j}$ can only be determined by actually executing the job to completion.

The server that is selected as the running server is the one with the earliest deadline and pending work among the set of servers in the Active state A. When no server is selected, the processor is idle or it is executing non-real time tasks.

Hard tasks can be directly scheduled by EDF, through dedicated isolated CSS servers characterised by their WCET, or hierarchical scheduling. Soft tasks are characterised by average values and can be served by isolated or non-isolated servers.

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

The reader should refer to [17] for a theoretical validation of the proposed model.

4 Capacity sharing and stealing

The main contribution of the work presented in this paper is the combination of two sources of extra bandwidth for an efficient overload control: (i) residual capacity allocated but unused when jobs complete in less than their budgeted execution time; and (ii) stealing capacity from inactive non-isolated servers used to schedule best-effort jobs. The main principles of the proposed approach are detailed in the next sections.

4.1 Dynamic budget accounting

Whenever a server is executing a task, 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 reserved capacity c_i, or is stealing capacity c_s from an inactive non-isolated server. The server to which the budget accounting is going to be performed is dynamically determined at the time instant when a capacity is needed.

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. On the other hand, it 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 is 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 capacities of inactive non-isolated servers, until its deadline, job completion, or non-isolated capacity exhaustion (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

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 the 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.

Since the execution requirements of each job are not known beforehand, it makes sense to devote as much excess capacity as possible to the currently executing server, maximising its chances to complete the current job before the deadline, 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 a deadline miss.

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 then 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 starts consuming the S_r's residual capacity c_r, running 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). As such, all available residual capacities are greedily assigned to the currently executing server until its deadline. This has been proved to minimise deadline postponements and the number of preemptions [15].

If all available residual capacities are exhausted and the current job is not complete, the server starts using its own capacity c_i (it points to itself), either until job's completion or c_i's exhaustion. On a c_i's exhaustion, S_i keeps in the Active state and maintains its deadline $d_{i,k}$.

4.3 Non-isolated capacity stealing

When the reserved capacity of some server is exhausted and there is still pending work, the server is allowed to steal inactive non-isolated capacities to handle its overload.

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

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

The currently executing overloaded server S_i connects to the earliest deadline inactive non-isolated server S_s, but continues to run with its own deadline $d_{i,k}$ (S_i is stealing the S_s's capacity and not its priority). When the stolen capacity is exhausted and the job has not been completed, the next non-isolated capacity c'_s is used (if any).

Keeping S_i's deadline allows to interrupt the current capacity stealing on a job arrival for S_s. This way, S_s reaches the Active state with the remaining budget, preserving system's schedulability. Note that while serving a task, an active non-isolated server behaves as isolated servers (it can share residual capacity and steal from other non-isolated servers).

Whenever S_i is connected to an inactive non-isolated server S_s and it is preempted or whenever a replenishment event occurs on the capacity being stolen, S_i is immediately disconnected from S_s and stops using that capacity capacity, keeping the Active state.

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 has 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

An overloaded server whose budget has exhausted can only continue its execution and steal inactive non-isolated capacities if its current capacity and deadline are not automatically updated when its capacity is exhausted. As such, CSS suspends capacity recharging and deadline update of each server S_i until a specific time r_i.

Setting the replenishment and deadline update of an active server to its currently assigned deadline, $r_i = d_i$, satisfies the purposes of the capacity reclaiming and non-isolated capacity stealing mechanisms presented above.

For each server S_i, when $t = r_i$, 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 $a_{i,k} \leq t < f_{i,k}$. An active server without pending work (it must be supplying its residual capacity to other servers) reaches the Inactive state and its residual capacity is discharged. 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 ($c_r = 0$).

Advancing the recharging times when there is pending work is against our purpose of executing periodic activities with stable frequencies. If pending jobs are a consequence

of early arrivals, executing periodic services with a stable frequency suggests that those early arrived jobs should only begin their execution in the expected period of arrival.

5 The CSS scheduler

In this section, CSS is formally described and an example of an efficient overload control is presented. 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.

1. When a job $J_{i,k}$ arrives at time $a_{i,k}$ for server S_i
   (a) if S_i is Inactive, it 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 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 Active server S_j *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, the next pending instance $J_{i,k+1}$ (if any) is executed using the current capacity and deadline. 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 Active, keeping its recharging time r_i and deadline $d_{i,k}$
5. If the server is Active and $t = r_i$, 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. 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

Consider the following periodic task set, described by average execution times and period: $\tau_1 = (2, 5)$, $\tau_2 = (4, 10)$, $\tau_3 = (3, 15)$. Task τ_1 is served by a non-isolated server, while tasks τ_2 and τ_3 are served by isolated servers. A possible execution of this task set is presented in Figure 1. When a server is using a residual or stolen capacity from another server a pointer indicates where the budget accounting is being performed.

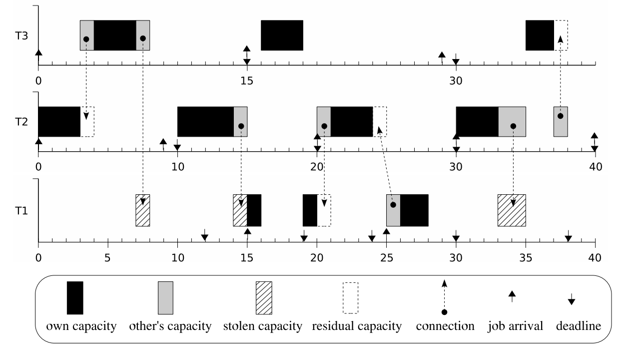

Figure 1. Overload handling with CSS

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 earliest deadline residual capacity available of server S_2. 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 the job is only released at time $t = 10$. Remember that advancing execution times is against our purpose of executing periodic activities with stable frequencies.

At time $t = 15$, after S_2 completes its job by stealing some of the inactive non-isolated capacity of S_1, a new job for server S_1 arrives. S_1 reaches the active state, keeping its current available capacity and corresponding deadline. Now, it behaves as an active isolated server and tries to use available residual capacities. Since there is not any residual capacity available, S_1 starts to consume its remaining reserved 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.

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. Note that a server remains in the Active state until its deadline, even if it has exhausted its capacity.

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 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 introduced by CSS to an efficient overload management 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 [20]. 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.

6.1 Capacity reclaiming

The performance of CSS when scheduling a set of periodic tasks served only by isolated servers was compared 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 whose effect in lowering the mean tardiness of periodic jobs was evaluated.

Different sets of 6 periodic servers, with varied capacities ranging from 20 to 50, and period distributions ranging from 60 to 600 were used, creating different types of load, from short to long deadlines and capacities. The execution time of each job varied in the range $[0.7Q_i, 1.4Q_i]$. The purpose of using random workloads was 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 the system's load, measuring the mean tardiness of periodic tasks under random workloads for different probabilities of jobs' overload.

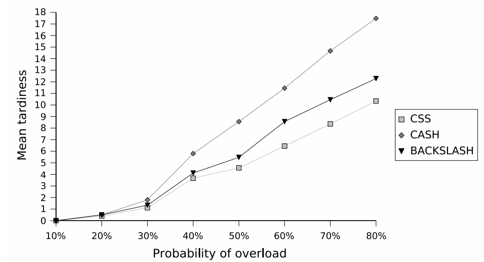

Figure 2. Performance in dynamic scenarios

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, since CSS keeps a server in Active state until its deadline without deadline postponement, it effectively improves servers' probability of actually using any spare capacity that eventually will be released until then, minimising the mean tardiness of periodic jobs. Furthermore, 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].

6.2 Capacity reclaiming and stealing

The second set of simulations evaluated the effect of non-isolated capacity stealing on the performance of soft real-time tasks, either with short or long variations from mean execution times.

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 $U_{non-isolated} = 1 - U_{isolated}$.

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

In the first simulation, periodic tasks were served by 1 non-isolated server $S_1 = (2, 10)$ and 4 isolated servers $S_2 = (3, 15)$,$S_3 = (4, 20)$,$S_4 = (5, 25)$, $S_5 = (6, 30)$, with utilisation of $U_{non-isolated} = 0.2$ and $U_{isolated} = 0.8$. The execution time of each job shortly varied in the range $[0.8Q_i, 1.2Q_i]$.

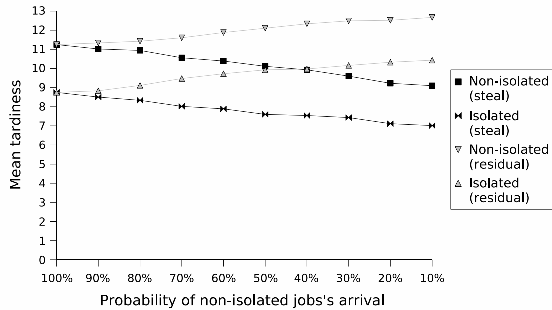

Figure 3. Small variation in execution times

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

Furthermore, 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 (anytime algorithms were not used to evaluate CSS in a more generic scenario). 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 [3]. 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 one, except that a greater variance of jobs' execution time was introduced, ranging from $[0.6Q_i, 1.8Q_i]$. Note that in this experiment the average value of the jobs' execution requirements is greater than the reserved capacity of their servers, necessarily leading to a greater tardiness. 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. One can conclude that severe overloads can be efficiently handled with residual capacity reclaiming and non-isolated capacity stealing, reducing the mean tardiness of periodic jobs.

7 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 and steal capacity from inactive non-isolated servers, effectively reducing the mean tardiness of periodic jobs. Furthermore, the proposed dynamic budget accounting mechanism eliminates the need of several server states and extra queues to manage residual and stolen capacities.

CSS offers the flexibility to consider the coexistence of

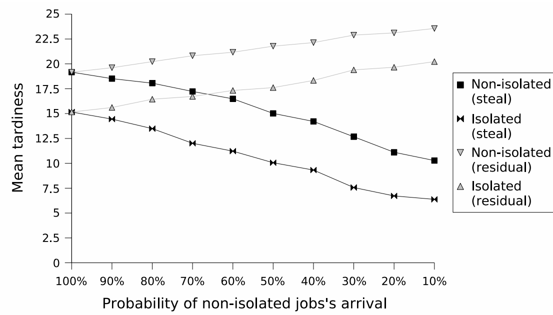

Figure 4. Large variation in execution times

guaranteed and best-effort servers in the same system. It has been demonstrated that the proposed algorithm can achieve a higher performance when considering the mean tardiness of periodic guaranteed services in systems where some services can appear less frequently, and when they do they can be served in a best-effort manner, giving priority to the overload control of guaranteed services. The achieved results become even more significant when tasks' computation times have a large variance.

Acknowledgements

This work was partly supported by FCT, through the CISTER Research Unit (FCT UI 608) and the Reflect project (POSC/EIA/60797/2004), and the European Comission through the ARTIST2 NoE (IST-2001-34820).

References

[1] 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.

[2] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings of the 25th IEEE RTSS*, pages 328–225, December 2004.

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

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

[5] 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.

[6] A. Colin and S. M. Petters. Experimental evaluation of code properties for wcet analysis. In *Proceedings of the 24th IEEE RTSS*, pages 190–199, December 2003.

[7] R. I. Davis. Approximate slack stealing algorithms for fixed priority preemptive systems. Technical report, Department of Computer Science, University of York, November 1993.

[8] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings of the 14th RTSS*, 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*, 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 RTSS*, 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 RTSS*, 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] 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.

[17] L. Nogueira and L. M. Pinho. Capacity sharing and stealing in server-based real-time systems. Technical report, HURRAY-TR-051205. Available at http://hurray.isep.ipp.pt/, December 2005.

[18] L. Nogueira and L. M. Pinho. Dynamic adaptation of stability periods for service level agreements. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 77–81, Sydney, Australia, August 2006.

[19] L. Nogueira and L. M. Pinho. Iterative refinement approach for qos-aware service configuration. *IFIP From Model-Driven Design to Resource Management for Distributed Embedded Systems*, 225:155–164, 2006.

[20] 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 1st International Workshop on Probabilistic Analysis Techniques for Real-Time Embedded Systems*, Pisa, Italy, September 2004.

[21] 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.

[22] 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.