IPP HURRAY!

www.hurray.isep.ipp.pt

Technical Report

A Capacity Sharing and Stealing Strategy for Open Real-time Systems

Luis Miguel Nogueira

Luis Miguel Pinho

HURRAY-TR-100503

Version:

Date: 05-10-2010

A Capacity Sharing and Stealing Strategy for Open Real-time Systems

Luis Miguel Nogueira, Luis Miguel 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:

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

Abstract

This paper focuses on the scheduling of tasks with hard and soft real-time constraints in open and dynamic real-time systems. It starts by presenting a capacity sharing and stealing (CSS) strategy 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) reclaiming unused reserved capacity when jobs complete in less than their budgeted execution time and (ii) stealing reserved capacity from inactive non-isolated servers used to schedule best-effort jobs.

CSS is then combined with the concept of bandwidth inheritance to efficiently exchange reserved bandwidth among sets of inter-dependent tasks which share resources and exhibit precedence constraints, assuming no previous information on critical sections and computation times is available. The proposed Capacity Exchange Protocol (CXP) has a better performance and a lower overhead when compared against other available solutions and introduces a novel approach to integrate precedence constraints among tasks of open real-time systems.

A Capacity Sharing and Stealing Strategy for Open Real-time Systems

Luís Nogueira * , Luís Miguel Pinho

CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto (ISEP/IPP)
Rua Dr. António Bernardino de Almeida 431
4200-072 Porto, Portugal
Phone: +351 22 8340529 Fax: +351 228340525

Abstract

This paper focuses on the scheduling of tasks with hard and soft real-time constraints in open and dynamic real-time systems. It starts by presenting a capacity sharing and stealing (CSS) strategy 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) reclaiming unused reserved capacity when jobs complete in less than their budgeted execution time and (ii) stealing reserved capacity from inactive non-isolated servers used to schedule best-effort jobs.

CSS is then combined with the concept of bandwidth inheritance to efficiently exchange reserved bandwidth among sets of inter-dependent tasks which share resources and exhibit precedence constraints, assuming no previous information on critical sections and computation times is available. The proposed Capacity Exchange Protocol (CXP) has a better performance and a lower overhead when compared against other available solutions and introduces a novel approach to integrate precedence constraints among tasks of open real-time systems.

Key words: Open real-time systems, Dynamic scheduling, Resource reservation, Residual capacity reclaiming, Reserved capacity stealing, Shared resources, Precedence constraints

* Corresponding author.
Email addresses: `lmn@isep.ipp.pt` (Luís Nogueira), `lmp@isep.ipp.pt` (Luís Miguel Pinho).

1 Introduction

As an increasing number of users runs both real-time and traditional desktop applications in the same system the issue of how to provide an efficient resource utilisation in this highly dynamic, open, and shared environment becomes very important. The need arises from the fact that independently developed services can enter and leave the system at any time, without any previous knowledge about their real execution requirements and tasks' inter-arrival times.

For most of these systems, the classical real-time approach based on a rigid off-line design and worst-case execution time (WCET) assumptions would keep resources unused for most of the time. Usually, tasks' WCET is rare and much longer than the average case. At the same time, it is increasingly difficult to compute WCET bounds in modern hardware without introducing excessive pessimism [1]. Such a waste of resources can only be justified for very critical systems in which a single missed deadline may cause catastrophic consequences.

A more flexible scheduling approach is then needed in order to increase resource usage. Flexibility is particularly important for small embedded devices used in consumer electronics, telecommunication systems, industrial automation, and automotive systems. In fact, in order to satisfy a set of constraints related to weight, space, and energy consumption, these systems are typically built using small microprocessors with low processing power and limited resources.

Guarantees based on average estimations are typically acceptable for soft real-time tasks since a deadline miss does not constitute a system or application failure but it is only less satisfactory for the user. Nevertheless, when scheduling soft tasks based on average estimated needs any chosen approach must handle the case when a task needs to execute more than its guaranteed reserved time. Not only it is desirable to achieve temporal isolation among soft tasks as well as the schedulability of hard tasks must not be compromised.

In [2], Mercer et al. propose a scheme based on capacity reserves to remove the need of knowing the WCET of each task under the Rate Monotonic [3] scheduling policy. A reserve is a couple (C_i, T_i) indicating that a task τ_i can execute for at most C_i units of time in each period T_i. If a task instance needs to execute for more than C_i, the remaining portion of the instance is scheduled in background.

Based on a similar idea of capacity reserves, Abeni and Buttazo [4] proposed the Constant Bandwidth Server (CBS) scheduler to handle soft real-time requests with a variable or unknown execution behaviour under the Earliest Deadline First (EDF) [3] scheduling policy. To avoid unpredictable delays on

hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task gets a fraction of the CPU and it is scheduled in such a way that it will never demand more than its reserved bandwidth, independently of its actual requests. This is achieved by assigning each soft task a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded. As a consequence, overruns occurring on a served task will only delay that task, without compromising the bandwidth assigned to other tasks.

However, with CBS, if a server completes a task in less than its budgeted execution time no other server is able to efficiently reuse the amount of computational resources left unused. To overcome this drawback, CBS has been extended by several reclaiming schemes [5; 6; 7; 8; 9] proposed to support an efficient sharing of computational resources left unused by early completing tasks. Such techniques have been proved to be successful in improving the response times of soft real-time tasks while preserving all hard real-time constraints.

Nevertheless, not all computational tasks in modern open real-time systems follow a traditional periodic pattern. For example, aperiodic complex optimisation tasks may take varying amounts of time to complete depending on the desired solution's quality or current state of the environment [10; 11; 12; 13; 14; 15]. Furthermore, the existing reclaiming schemes are unable to donate reserved, but still unused, capacities to currently overloaded servers.

Based upon a careful study of the ways in which unused reserved capacities can be more efficiently used to meet deadlines of tasks whose resource usage exceeds their reservations, our previous work [16] proposed the coexistence of the traditional *isolated* servers with a novel *non-isolated* type of servers, combining an efficient reclamation of residual capacities with a controlled isolation loss. The goal of the Capacity Sharing and Stealing (CSS) scheduler is to reduce the mean tardiness of periodic guaranteed jobs by handling overloads with additional capacity available from two sources: (i) by reclaiming unused allocated capacity when jobs complete in less than their budgeted execution time; and (ii) by stealing allocated capacities from inactive non-isolated servers used to schedule aperiodic best-effort jobs.

However, CSS assumes tasks to be independent. A challenging problem in open real-time systems is how to schedule inter-dependent tasks that share resources and exhibit precedence constraints without a complete previous knowledge about their actual runtime behaviour. The Capacity Exchange Protocol (CXP) [17] builds upon CSS and integrates its capacity sharing and stealing strategy with the concept of bandwidth inheritance [18] to mitigate the cost of blocking

on soft real-time tasks whose actual execution behaviour is only known by executing tasks until completion. While preserving the isolation principles of independent tasks, upon blocking, a task is allowed to be executed on more than its dedicated server, efficiently exchanging reserved capacities among servers to reduce the undesirable effects caused by inter-task blocking.

In this paper we provide a complete and consistent description of these protocols and extend the conducted evaluation, simultaneously dealing with capacity sharing, stealing and exchanging. More important, the paper also provides a proof of correctness of the proposed runtime exchange of reserved capacities. Hard schedulability guarantees can be provided either for independent and inter-dependent task sets, even when hard and soft real-time tasks do share resources and exhibit precedence constraints in open real-time systems.

In the remainder of this paper, we describe the system model and used notation in Section 2. Section 3 analyses the most significant scheduling approaches proposed to improve the performance of soft real-time tasks and introduces the need for the novel capacity sharing and stealing approach described in Section 4. The correctness of the proposed runtime exchange of reserved capacities for independent task sets is proved in Section 5. CXP is described in detail in Sections 6 and 7, as a way to efficiently support shared resources and precedence constraints among inter-dependent task sets of open real-time systems. Although the goal of CXP is to minimise the cost of blocking among soft real-time tasks, Section 8 describes how hard schedulability guarantees can still be provided even when hard and soft real-time tasks share resources, at the expense of some pessimism on the computation of blocking times when tasks access (nested) critical sections. Section 9 presents and analyses the achieved evaluation results. Finally, Section 10 concludes this paper.

2 System model

This paper focus on dynamic open real-time systems where all services execute on a single shared processor, the sum of the reserved capacities is no more than the maximum capacity of the processor, and the scheduler does not have any previous complete knowledge about the execution requirements of soft real-time tasks. We make the reasonable assumption that whenever a service arrives to the system it advertises its requirements on a certain amount of the system's resources based on expected average needs for soft real-time tasks and WCET measures for hard real-time tasks. If, given the current system's load, the required amount can be guaranteed, the service is accepted and the requested amount is reserved.

A service is composed of a set of hard and/or soft real-time tasks. Each real-

time task τ_i can generate a virtually infinite sequence of jobs. The jth job of task τ_i arrives at time $a_{i,j}$, is released to the ready queue at time $r_{i,j}$, starts to be executed at time $s_{i,j}$ with deadline $d_{i,j} = r_{i,j} + p_i$, with p_i being the period of τ_i, and finishes its execution at time $f_{i,j}$. These times are characterised by the relations $a_{i,j} \leq r_{i,j} \leq s_{i,j} \leq f_{i,j}$.

For a hard real-time task τ_i, the system must provide an a priori guarantee that every job must complete at a time $f_{i,j} \leq d_{i,j}$. As such, p_i refers to the minimum inter-arrival time between successive jobs of τ_i so that $a_{i,j+1} \geq a_{i,j} + p_i$ and its execution requirements $e_{i,j}$ are characterised by the task's WCET.

For soft real-time tasks, the timing constraints are more relaxed. In particular, for a soft task τ_i, p_i represents the expected inter-arrival period between successive jobs. As such, the arrival time $a_{i,j}$ of a particular job is only revealed at runtime and the exact execution requirements $e_{i,j}$ can only be determined by actually executing the job to completion until time $f_{i,j}$.

Each soft or hard real-time task τ_i is scheduled through an abstract entity S_i called server. As such, all the jobs generated by task τ_i are dedicated to server S_i. Each server S_i is characterised by a pair (Q_i, T_i), where Q_i is the server's maximum reserved capacity and T_i its period. For a hard real-time task τ_i, its dedicated server S_i has a reserved capacity Q_i equal to the task's WCET and a period T_i equal to the task's period. For soft real-time tasks, Q_i and T_i are set based on the served tasks' expected average values. It is important to point out that this paper does not deal with policies to optimally assign or dynamically change the servers' parameters according to the actual needs of the served soft real-time tasks either based on some heuristic algorithms or feedback control schemes as appear, for example, in [19].

At each instant, the following values are associated with a server S_i: its currently assigned deadline d_k^i, its remaining execution capacity $0 \leq c_k^i \leq Q_i$, the amount of residual capacity $r_k^i \leq c_k^i$ that can be reclaimed by other servers, and its currently assigned replenishment time $h_k^i = d_k^i$. If at time t, S_i finishes the execution of its currently served job without exhausting its reserved execution capacity c_k^i and it has no pending work, the remaining amount $c_k^i > 0$ sets the server's residual capacity $r_k^i = c_k^i$ that can be reclaimed (c_k^i is subsequently set to zero). By pending work we refer to the case when there exists at least a served job such that $r_{i,j} \leq t < f_{i,j}$.

This paper considers two different types of servers: *isolated* servers used to schedule periodic and sporadic guaranteed tasks and *non-isolated* servers for aperiodic best-effort tasks. For an isolated server S_i it is ensured that its amount of reserved capacity Q_i is available every period T_i to its dedicated task τ_i. On the other hand, a non-isolated server S_j can have some or all of its reserved capacity Q_j stolen by one or several needed overloaded servers

and, as such, it is not guaranteed that its dedicated task τ_j can execute for Q_j every period T_j. At any given time, the earliest deadline server in the ready queue with unfinished jobs that can use some eligible capacity is selected for execution, based on the EDF priority assignment [3]. When no server is selected, the processor is idle or it is executing non-real time tasks.

Served tasks of the same or from different services may simultaneously need exclusive access to one or more of the system's resources R, during part or all of their executions. If task τ_i is using resource R_i, it locks that resource. Since no other task can access R_i until it is released by τ_i, if τ_j tries to access R_i it will be blocked by τ_i. Blocking can also be indirect (or transitive) if although two tasks do not share any resource, one of them may still be indirectly blocked by the other through a third task. The conditions under which nested critical sections are allowed as well as resource sharing among soft and hard real-time tasks is discussed in detail in Section 8.

Furthermore, tasks within a service may also exhibit precedence constraints among them. A task τ_i is said to precede another task τ_k if τ_k cannot start until τ_i is finished. Such precedence relation is formalised as $\tau_i \prec \tau_k$ and guaranteed if $f_{i,j} \leq s_{k,j}$. Precedence constraints are defined in the service's description at admission time by a directed acyclic graph $\mathcal{G}$, where each node represents a task and each directed arc represents a precedence constraint $\tau_i \prec \tau_k$ between two tasks τ_i and τ_k. Given a partial order $\prec$ on the tasks, the release times and deadlines are said to be consistent with the partial order if $\tau_i \prec \tau_k \Rightarrow r_{i,j} \leq r_{k,j}$ and $d_{i,j} < d_{k,j}$.

The schedulability of hard real-time tasks can be guaranteed as long as it is possible to perform an accurate analysis and bound the execution times of hard tasks, their minimum inter-arrival times, and the duration of the accessed critical sections and maximum blocking time. Please refer to Sections 5 and 8 for a detailed analysis.

3 Improving the system's performance

Since the actual execution time of some tasks can be affected by several factors, they may use less than the reserved amount of resources, dynamically releasing extra residual capacity. Similarly, all except hard real-time tasks occasionally (or even frequently) need more resources than they have reserved. As such, an efficient reclamation and redistribution of unused reserved capacities can significantly improve the performance of both soft real-time and best-effort applications.

By the very nature of open real-time systems, the availability of residual ca-

pacities is unknown beforehand and can only be scheduled dynamically when it is detected. Similarly, overload situations are not known until an unfinished task has consumed all of its reserved resources. Many researchers have examined these issues in various contexts and have developed a number of effective techniques for improving the system's performance through a better redistribution of unused reserved capacities. Some of the most relevant works are discussed in the next paragraphs.

In [20], Bernat and Burns propose a capacity sharing protocol for enhancing soft aperiodic responsiveness in a fixed priority environment, where each task is handled by a dedicated server. The protocol allows an overloaded server to steal capacity from other servers to advance the execution of the served tasks, thus loosing isolation among the served tasks.

The capacity sharing protocol of [20] has been extended by the HisReWri algorithm [21]. The algorithm identifies those tasks that did execute when a hard task has released some of its maximum allocated capacity 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 [4] is a well known algorithm which is able to provide temporal protection among tasks, isolating the behaviour of each task from the rest of the system. As such, it is possible to guarantee the real-time performance of a task by considering it in isolation. Such property is particularly useful when mixing hard and soft real-time tasks in the same system. Since CBS can properly cope with both periodic and aperiodic activations, it has been used as the baseline reservation-based algorithm in several other schedulers with the ability to reclaim residual capacities originated by early completions.

GRUB [5] reduces the number of task preemptions by assigning all the excess capacity to the currently executing CBS server. 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. However, GRUB always postpones a server's deadline before starting a new job, regardless of the current value of the server's budget. 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 [6]. GRUB has later been extended to properly schedule both real-time and non-real-time tasks by following a hard reservation approach [22].

CASH [6] uses a global queue of residual capacities originated by early completions, ordered by deadline. Whenever a CBS server is scheduled for execution

it will first use any queued capacity whose deadline is less than or equal to its own, 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 the servers' capacities without suspending the tasks on every capacity exhaustion [7]. An improvement to CASH's residual bandwidth reclaiming and the ability to work in the presence of shared resources has been more recently proposed in [8].

IRIS [7] adds a hard reservation approach [23] to CBS and uses such property to reclaim unused computation times. It identifies the deadline aging problem of CBS when scheduling acyclic tasks (tasks that are continuously active for large intervals of time) and proposes to suspend each task's replenishment until a specific time. It is also a fair algorithm in the sense that residual capacity is equally distributed among the servers that need to execute more than the reserved time but it may suffer from an excessive number of context switches. Furthermore, residual capacity reclaiming is only performed after all the servers had exhausted their reserved capacities, potentially wasting valuable bandwidth that could otherwise have been used to advance the execution of overloaded servers.

BACKSLASH [9] 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 [21]. At every capacity exhaustion, servers' capacities 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 [9].

While the scheduling schemes discussed above generally improve the system's performance, as the number of applications with soft real-time constraints continues to grow, new scheduling requirements are emerging, particularly where it may be preferable to have approximate results of a poorer but acceptable quality delivered within the available computation time, than late results with the desirable optimal quality [10; 11; 12; 13; 14; 15].

Consider, for example, a route optimiser that is part of the navigation system of an automated vehicle [12]. Given the state of the external and internal world, the system is continuously searching for the best path. The task execution is unbounded, data-driven, not predictably regular, and as a result its operation is not easily parcelled for a periodic execution. With an anytime approach, the

optimisation task can be interrupted at any time and still be able to provide a solution and a measure of its quality. As such, systems can take advantage of the flexibility offered by anytime algorithms as long as a scheduling mechanism that can regulate their behaviour is developed.

As such, while is still crucial to support an efficient reclaiming of residual capacities, a more flexible overload control of guaranteed services can be achieved if isolation is also reduced in a controlled fashion in order to donate reserved, but still unused, capacities to currently overloaded servers. However, all the previous extensions to CBS are only able to reclaim the unused allocated capacity made available when jobs complete in less than their budgeted execution time but unable to reclaim the unused capacities of idle servers.

As it will be shown in the next section, the proposed capacity sharing and stealing approach is able to distinguish between reclaiming "residual capacity" due to earlier completions of periodic tasks (predictable arrivals) and reclaiming "non-isolated capacity" from inactive servers which handle aperiodic best-effort tasks (non-predictable arrivals) in order to advance the execution of overloaded servers.

4 The Capacity Sharing and Stealing approach

The Capacity Sharing and Stealing (CSS) scheduler integrates and extends some of the best principles of previous reservation-based scheduling approaches to improve the responsiveness of soft real-time tasks in the presence of overruns while ensuring that the schedulability of hard tasks is not compromised. To ease the algorithm's discussion, the main principles of the proposed approach are discussed in the next paragraphs and the CSS scheduler is formally presented in Section 4.3.

CSS assumes that all tasks in the system are independent and no task is allowed to suspend itself waiting for a shared resource or a synchronisation event. The system consists of n CSS servers and a global scheduler based on the EDF priority assignment. A single ready queue exists and, at each instant, the active server with the earliest deadline S_i is selected and its corresponding task τ_i is dispatched to execute.

The algorithm considers two different types of servers: *isolated* servers used to schedule periodic and sporadic guaranteed tasks and *non-isolated* servers for aperiodic best-effort tasks. For an isolated server S_i it is ensured that its amount of reserved capacity Q_i is available every period T_i to its dedicated task τ_i. On the other hand, a non-isolated server S_j can have some or all of its reserved capacity Q_j stolen by one or several needed overloaded servers

and, as such, it is not guaranteed that its dedicated task τ_j can execute for Q_j every period T_j.

At time t, an isolated or non-isolated server S_i is said to be *active* if (i) the served task is ready to execute; (ii) is executing; or (iii) the server is supplying its residual capacity $r_k^i > 0$ to other servers until its currently assigned deadline d_k^i. Otherwise, S_i is *inactive* if (i) there are no pending jobs to serve; and (ii) the server has no residual capacity that can be reclaimed by other servers, that is, $r_k^i = 0$.

State transitions are determined by the (i) release of a new job at time $r_{i,k}$, if $r_{i,k} \geq d_{k-1}^i$, or (ii) non-existence of pending jobs either at the server's current replenishment time h_k^i or at the exhaustion of the server's residual capacity $r_k^i = 0$ (Figure 1). Note that if a soft real-time job arrives earlier than expected, that is $a_{i,k} < d_{k-1}^i$, and its dedicated server S_i is inactive, the task is only released to the ready queue at the server's next replenishment time h_k^i and only then S_i becomes active. On the other hand, an active server becomes inactive (i) at a time $t < h_k^i$, if all its reserved capacity is consumed (either as its own execution capacity c_k^i or as residual capacity r_k^i consumed by other servers) and there are no pending jobs to serve; or (ii) at its currently set replenishment time h_k^i, if there are no pending jobs to serve, even with some reserved capacity left.

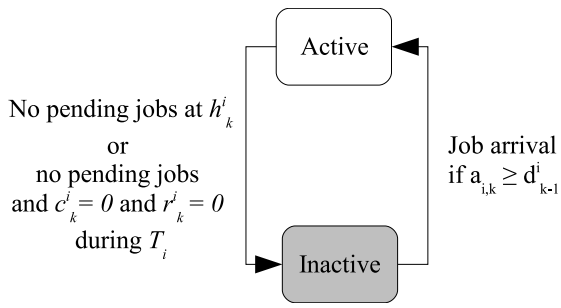

Fig. 1. State transitions of a CSS server S_i

At the beginning, all servers are inactive with deadlines $d_0^i = 0$ and are only activated with the arrival of a job of their dedicated tasks. To ensure the correct behaviour of the system on phased arrivals, the deadline of a server associated to a hard real-time task is assigned in such a way that the server's deadline is coincident with its dedicated task's deadline. On the other hand, a soft real-time task simply inherits its server's deadline, set to $d_1^i = ri, 1 + T_i$, to enhance its responsiveness.

As with all reservation-based schedulers, CSS is characterised by three mechanisms: *accounting*, *enforcement*, and *replenishment*. Accounting measures the CPU time consumed by a server S_i in order to properly determine its remaining available capacity c_k^i. As such, whenever a task served by S_i executes for a period of time Δ_t, the server's available execution capacity c_k^i is decreased

by Δ_t.

Enforcement and replenishment, as opposed to CBS, are performed at a server's currently assigned recharging time h_k^i rather than immediately after the server's capacity exhaustion, thus following a hard reservation approach [23]. When the server's capacity is exhausted, the server is said to be depleted and cannot be recharged ($c_{k+1}^i = Q_i$) and its deadline postponed ($d_{k+1}^i = d_k^i + T_i$) until its currently assigned recharging time h_k^i is reached.

Recall that CBS presents some drawbacks when serving tasks that are active for long intervals of time, covering therefore many periods of a server [7; 22]. Since CBS automatically recharges a server's capacity and postpone its deadline on every capacity exhaustion, if the server's deadline, although postponed, is still the earliest, the renewed capacity can be used within the same period. This leads to a temporal over execution that may be followed by a starvation period, altering the rate of periodic tasks.

Furthermore, advancing a server's recharging time whenever there is pending work is against our purpose of executing periodic tasks with stable frequencies. Note that if pending work is a consequence of early arrivals of soft real-time 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. When pending work is due to a server's overload, instead of allowing a task to use resources allocated to the next job of the same task by recharging the server's capacity and postponing its deadline, CSS allows an overload server either to steal reserved capacities from inactive non-isolated servers or reclaim any new residual capacities that eventually are released until its currently assigned deadline.

4.1 Capacity reclaiming

Whenever a job of the currently executing sever S_i is completed in less than its budgeted execution time and the server has no pending work, its remaining reserved capacity $c_k^i > 0$ can (and should) be used by any other active server to advance the execution of its own tasks.

CSS proposes to efficiently reclaim unused computation times as early as possible. Note that when using a residual capacity of another server, a task must be scheduled using the current deadline of the server from which the residual capacity belongs to. Since capacities expire at their deadlines, it makes sense to reclaim residual capacities before consuming the server's reserved capacity in order to increase the probability of effectively using them.

As such, with CSS, the remaining reserved capacity $c_k^i > 0$ of a server S_i

without pending work is immediately released as residual capacity $r_k^i = c_k^i$ and c_k^i is set to zero. Then, r_k^i can immediately be reclaimed by any eligible active server S_j until the currently assigned S_i's deadline d_k^i.

Definition 1 *The set of active servers A_r eligible for residual capacity reclaiming whenever a server S_j is scheduled for execution is given by $A_r = \{S_r | S_r \in A, d_k^r \leq d_k^j, c_k^r > 0\}$, where A is the set of all active servers, d_k^r is the current deadline of early completed jobs and d_k^j is the currently assigned deadline of server S_j.*

When scheduled with CSS, a server S_j starts by reclaiming the residual capacity $r_k^{edf} > 0$ supplied by the earliest deadline active server S_{edf} from the set of eligible servers A_r, either until the job's completion or c_k^{edf}'s exhaustion.

Definition 2 *The earliest deadline active server S_{edf} from the set of eligible servers A_r is defined as $\exists^1 S_r \in A_r : min_{d_k^r}(A_r), A_r \neq \emptyset$.*

Please note that the $\exists^1$ relation is guaranteed by the *min* function which, whenever there is more than one server with the same earliest deadline, always returns the first server on the list.

Furthermore, since the execution requirements of each soft real-time job are not known beforehand, it also makes sense to devote as much excess capacity as possible to the currently executing server. As such, while there is pending work to do, remaining residual capacities are greedily consumed by the currently executing server according to an EDF policy.

We carefully considered this fairness issue. The increased computational complexity of fairly assigning residual capacities to all active servers and the fact that fairly distributing residual capacities to a large number of servers (usually in proportion of the servers' bandwidths) 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 capacity to the currently executing server S_j. Such a greedy capacity reclaiming not only has a reduced computational complexity, it also minimises deadline postponements and the number of preemptions and tends to be fair in the long run [5].

As such, with CSS, a server S_j only starts to consume its own reserved capacity c_k^j to advance the execution of the currently served job of its dedicated task whenever all available eligible residual capacities in the system are exhausted, either until the job's completion or c_k^j's exhaustion.

CSS proposes to reduce isolation in a controlled fashion to donate reserved, but still unused, non-isolated capacities to currently overloaded servers to minimise soft tasks' response times whilst guaranteeing that the deadlines of hard tasks are met.

In fixed priority environments, similar approaches have been proposed [24; 25]. However, those proposals present some drawbacks that invalidate their use in dynamic open real-time systems. The work in [24] relies on a pre-computed table that define the residual capacity present on each invocation of a hard task. In contrast, [25] determines the amount of available capacity at run time, but the execution time overhead introduced by the optimal dynamic approach is infeasible in practice [26].

With CSS, inactive non-isolated servers can have some or all of their reserved capacity stolen by an active overloaded server S_i that has already consumed all of its own reserved capacity c_k^i.

Definition 3 *The set of inactive non-isolated servers I_s eligible for capacity stealing by the currently executing server S_i which has exhausted all of its own reserved capacity c_k^i is given by $I_s = \{S_s | S_s \in I, d_k^s \leq d_k^i, c_k^s > 0\}$, where I is the set of all inactive non-isolated servers, d_k^s is the current deadline of each inactive non-isolated server S_s.*

Since the parameters of inactive servers are not automatically updated, whenever the currently executing server S_i is determining the set of eligible inactive non-isolated servers I_s it needs to verify if an update of the current values of the deadline and reserved capacity of each inactive non-isolated server S_s is needed. As such, if the previously generated absolute deadline d_k^s of the selected non-isolated server S_s is lower than the current time ($d_k^s < t$), a new deadline ($d_k^s = t + T_s$) is generated and the server's capacity is recharged to its maximum value ($c_k^s = Q_s$). Otherwise, S_s's current values are used. In either case, S_s is kept in the inactive state.

After determining I_s, S_i is able to steal the non-isolated capacity of the earliest deadline inactive non-isolated server S_{edf} from the set of eligible inactive non-isolated servers I_s.

Definition 4 *The earliest deadline inactive non-isolated server S_{edf} from the set of eligible servers I_s is defined as $\exists^1 S_s \in I_s : min_{d_k^s}(I_s), I_s \neq \emptyset$.*

Non-isolated capacity stealing also follows a greedy approach. Whenever the inactive capacity c_k^{edf} being stolen is exhausted and the job has not yet been completed, the next non-isolated capacity $c_k^{edf'}$ is used (if any) by S_i to advance

its execution.

However, capacity stealing is interrupted whenever (i) the currently executing server S_i is preempted; (ii) a replenishment event occurs on the inactive capacity c_k^s being stolen and the new deadline assigned to S_s either becomes larger than the one currently assigned to S_i or it ceases to be the earliest inactive non-isolated one; or (iii) a new job arrives for the inactive non-isolated server S_s, whose reserved capacity is being used by S_i. As expected, when a new job arrives for the inactive non-isolated server S_s, it becomes active with its currently remaining capacity c_k^s and not with its fully recharged capacity. However, an active non-isolated server can also reclaim eligible residual capacities, steal inactive non-isolated capacities, and share its own residual capacity with other servers whenever its job is completed in less than its budgeted execution time.

4.3 The CSS scheduler

When a new job $J_{i,k}$ arrives at time $a_{i,k}$ for server S_i, if S_i is active, the job is buffered and will be served later. If S_i is inactive and if $a_{i,k} < d_k^i$, the server becomes active and the job is served with the last generated deadline d_k^i, using the current server's capacity c_k^i. Otherwise, S_i's capacity is recharged to its maximum value $c_k^i = Q_i$, a new deadline is generated to $d_k^i = max\{a_{i,k}, d_{k-1}^i\} + T_i$, the server's recharging time is set to $h_k^i = d_k^i$ and its residual capacity is set to $r_k^i = 0$.

Whenever a server is executing, the consumed capacity must be decreased by the same amount. By dynamically managing a pointer to the server from which the capacity is going to be decreased, the proposed dynamic accounting mechanism of CSS eliminates the need of extra queues or additional server states, reducing its overhead. The server from which the accounting is going to be performed is dynamically determined at the time instant when a capacity is needed.

CSS uses the following rules to manage reserved capacities:

- **Rule A (residual capacity release):** Whenever a server S_j completes its kth job of its associated task τ_j and it has no pending work, its remaining reserved capacity $c_k^j > 0$ is released as residual capacity $r_k^j = c_k^j$ and c_k^j is set to zero. The released residual capacity r_k^j can immediately be reclaimed by eligible active servers until the currently assigned S_j's deadline d_k^j. S_j is kept active with its current deadline until its residual capacity r_k^j is exhausted by other servers.
- **Rule B (residual capacity reclaim):** The next active server S_i scheduled for execution points to the earliest deadline server S_{edf} from the set of

eligible active servers A_r for capacity reclaiming. S_i consumes the pointed residual capacity r_k^{edf}, running with the deadline d_k^r of the pointed server S_{edf}. Whenever r_k^{edf} is exhausted and there is pending work, S_i disconnects from S_{edf} and selects the next available server S'_{edf} (if any).

- **Rule C (dedicated capacity consumption):** If all eligible residual capacities are exhausted and the current k^{th} job of server S_i is not yet completed, S_i consumes its own reserved capacity c_k^i either until the job's completion or c_k^i's exhaustion (whatever comes first). If c_k^i is exhausted and there is still pending work to do, S_i is kept active with its current deadline d_k^i.
- **Rule D (inactive non-isolated capacity steal):** A server S_i with pending work and no available execution capacity ($c_k^i = 0$) connects to the earliest deadline server S_{edf} from the set of eligible inactive non-isolated server I_s. S_i steals the pointed inactive capacity c_k^{edf}, running with its current deadline d_k^i. Whenever c_k^{edf} is exhausted and the job has not yet been completed, the next non-isolated capacity $c_k^{edf'}$ is used (if any).

Note that the proposed dynamic capacity accounting mechanism ensures that at time t, the currently executing server S_i is using a residual capacity $r_k^j > 0$ originated by an early completion of another active server S_j, its own reserved capacity $c_k^i > 0$, or is stealing capacity $c_k^s > 0$ from an inactive non-isolated server S_s. To preserve schedulability, it ensures that the longest time a server can be connected to another server S_j is bounded by the currently pointed server's capacity c_k^j and deadline d_k^j.

CSS is then able to (i) achieve isolation among guaranteed tasks; (ii) efficiently reclaim unused computation times, exploiting early completions; (iii) allow an overloaded server to steal non-isolated reserved capacities from inactive servers. As will be demonstrated in Section 9, the proposed approach is able to reduce the mean tardiness of periodic guaranteed jobs in highly dynamic open real-time systems.

4.4 Handling overloads with CSS

The next example details how CSS can handle soft tasks' overloads without postponing deadlines by greedily reclaiming residual capacities and stealing inactive non-isolated capacities used to schedule aperiodic best-effort services.

Consider the following periodic task set, described by average execution times and period: $\tau_1 = (2, 5)$, $\tau_2 = (4, 10)$, $\tau_3 = (3, 15)$. τ_1 is served by the non-isolated server S_1, while tasks τ_2 and τ_3 are served by the isolated servers S_2 and S_3, respectively, with periods and capacities equal to the tasks' expected values. A possible scheduling of this task set with CSS is detailed in Figure

2. When a server is connected to another server, either reclaiming a residual capacity or stealing an inactive non-isolated capacity, an arrow indicates where the capacity accounting is being performed.

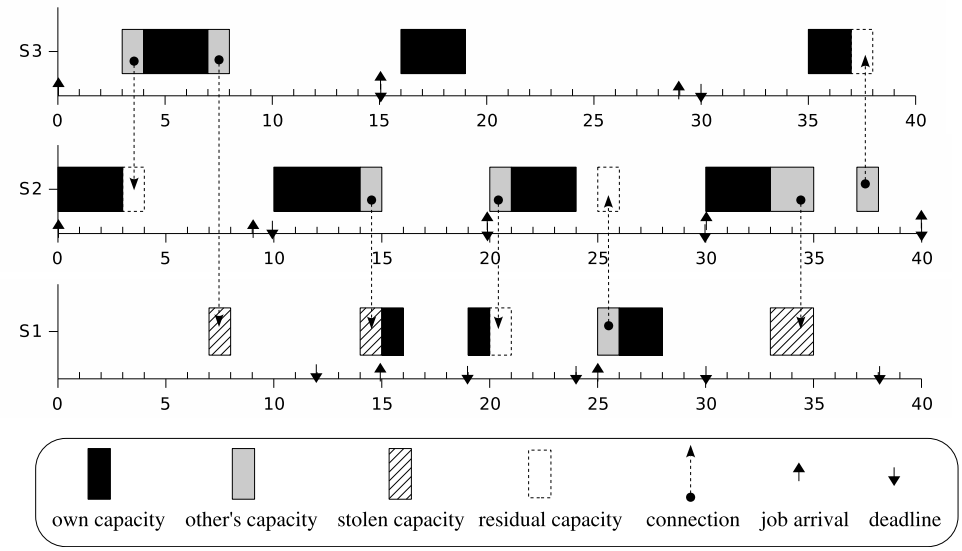

Fig. 2. Handling overloads with CSS

At time $t = 3$, τ_2 finishes its job and releases a residual capacity $r_1^2 = 1$ with deadline $d_1^2 = 10$ (Rule A). Server S_3 is scheduled for execution, connects to S_2, the earliest deadline residual capacity available, and starts to execute its task τ_3, consuming the reclaimed residual capacity r_1^2 (Rule B). When this residual capacity is exhausted at time $t = 4$, S_2 becomes inactive and S_3 continues to execute τ_3 by using its own reserved capacity until it is exhausted at time $t = 7$ (Rule C).

At time $t = 7$, S_3 has pending work and no capacity left ($c_1^3 = 0$). Since there is inactive non-isolated capacity available, S_3 is able to continue its execution by stealing the inactive non-isolated capacity of server S_1 (Rule D). Since $d_0^1 < t$, a new deadline $d_1^1 = t + T_1 = 5 + 7 = 12$ is generated and the server's execution capacity is recharged to its maximum value $c_1^1 = Q_1 = 2$.

Note that at time $t = 9$ a new job of τ_2 arrives for the inactive server S_2 but the job is only released to the ready queue at time $t = 10$. Recall that advancing execution times is against our purpose of executing periodic activities with stable frequencies.

At time $t = 15$, after S_2 has completed its job by stealing some of the inactive non-isolated capacity of S_1, a new job for server S_1 arrives. Note that S_1 becomes active, keeping its currently available capacity $c_2^1 = 1$ and corresponding deadline $d_2^1 = 19$.

At time $t = 16$, server S_1 exhausts its capacity and stops executing since

there are is no available inactive non-isolated capacity to steal in the system. The server remains active and, at time $t = 19$ (the server's replenishment time h_2^1), a replenishment of S_1's capacity occurs and it continues to execute the pending job. At time $t = 20$, completes the job and releases the residual capacity $r_3^1 = c_3^1 = 1$ with deadline $d_3^1 = 24$ and sets c_3^1 to zero. This residual capacity r_3^1 is used by server S_2 before consuming its own capacity at time $t = 21$.

At time $t = 25$, a new job of τ_1 arrives and the inactive non-isolated server S_1 becomes active. Note that, like any other active server in the system, it first reclaims the residual capacity $r_3^2 = 1$ with deadline $d_3^2 = 30$, released at time $t = 24$ by an early completion of τ_2, before consuming its own capacity.

At time $t = 33$ an overload of τ_2 is first handled by stealing the inactive non-isolated capacity of server S_1 and then, at time $t = 38$, by consuming the available residual capacity released after an early completion of task τ_3. Recall that with CSS a server is kept active even if it has exhausted its capacity. As shown, this behaviour enables an overloaded server to take advantage of any eligible residual capacity that is released until its currently assigned deadline.

5 Theoretical validation for independent task sets

In this section we analyse the schedulability condition for a hybrid set of hard and soft real-time tasks. CSS is able to reduce the mean tardiness of soft real-time tasks through an efficient management of unused reserved capacities. If each hard real-time task is scheduled by an isolated CSS server with a reserved capacity equal to the task's WCET and period equal to the task's period, it behaves like a standard hard task scheduled by EDF. The main difference is that, with CSS, hard tasks can use extra capacities and yield their residual capacities to other tasks.

In [27], it is proven that a CBS 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 the server's 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 average values, can be scheduled together with hard tasks, even in the presence of overloads.

Here, we state that the runtime capacity exchange performed by CSS does not affect the system's schedulability. By assigning each soft task a specific capacity, based on an average execution time estimation, the desired activation

period, and isolating the effects of tasks' overloads, a hybrid task set can be guaranteed using the classical Liu and Layland condition [3].

Before proving the schedulability test, we start by ensuring that all generated capacities are exhausted before their respective deadlines.

Lemma 1 *Given a set I of isolated servers, each isolated capacity generated during scheduling is either consumed or discharged until its deadline.*

Proof

Let $a_{i,k}$ denote the instant at which a new job $J_{i,k}$ arrives and the isolated associated server $S_i \in I$ is inactive. At $a_{i,k}$, a new capacity $c_k^i = Q_i$ is generated and S_i is released to the ready queue.

Let $\forall_{i,k}\ d_k^i = max\{a_{i,k}, d_{k-1}^i\} + T_i$ be the deadline and $\forall_{i,k}\ h_k^i = d_k^i$ the replenishment time associated with the isolated capacity c_k^i.

Let $[t, t + \Delta_t[$ denote a time interval during which server S_i is executing, consuming its own capacity c_k^i. Consequently, S_i has used an amount equal to $c_k^{i'} = c_k^i - \Delta_t \geq 0$ of its own capacity during Δ_t. As such, the server's reserved capacity c_k^i must be decreased to $c_k^{i'}$, until it is exhausted.

Let $f_{i,k}$ denote the time instant at which server S_i completes its job $J_{i,k}$. Assume that there are no pending jobs for server S_i at time $f_{i,k}$ and $c_k^i > 0$. According to Rule A, the available residual capacity $r_k^i = c_k^i$ can immediately be reclaimed by other servers and the server's capacity c_k^i is set to zero.

At instant $f_{i,k}$, another active server S_j is scheduled for execution. According to Rule B, if the inequality $d_k^i \leq d_l^j$ holds, let $[t, t + \Delta_t[$ denote the time interval during which server S_j is executing, consuming the residual capacity r_k^i of server S_i. Consequently, r_k^i must be decreased to $r_k^{i'} = r_k^i - \Delta_t \geq 0$, until the residual capacity of server S_i is exhausted or the currently assigned deadline d_k^i of server S_i is reached.

At replenishment time $t = h_k^i$, any remaining residual capacity r_k^i of server S_i not used by another active server is discharged.

□

Lemma 2 *Given a set S of isolated and non-isolated servers, each non-isolated capacity generated during scheduling is either consumed or discharged until its deadline.*

Proof

To prove this lemma we analyse the following cases: a) a new non-isolated

capacity is generated whenever an overloaded active server needs to steal the inactive non-isolated server's capacity; and b) a non-isolated capacity is generated whenever a new job arrives for the inactive non-isolated server.

Case a.

Let $a_{j,k}$ denote the time instant when an active overloaded server S_j starts to consume the non-isolated capacity c_k^i of the inactive non-isolated server S_i.

If the inequality $d_{k-1}^i \leq a_{j,k}$ holds, a new deadline $d_k^i = a_{j,k} + T_i$ is generated for the non-isolated capacity c_k^i, the server's capacity c_k^i is recharged to its maximum value $c_k^i = Q_i$ and the replenishment time h_k^i is set to $h_k^i = d_k^i$. Otherwise, the inactive non-isolated server S_i keeps its current values of c_k^i, d_k^i, and h_k^i.

Let $[t, t + \Delta_t[$ denote the time interval during which server S_j is executing, stealing the non-isolated capacity c_k^i of server S_i. Consequently, the consumed non-isolated capacity c_k^i must be decreased to $c_k^{i'} = c_k^i - \Delta_t \geq 0$, until it is exhausted.

If a new job arrives at any time $a_{i,k} < a'_{i,k} < h_k^i$, the inactive non-isolated server S_i becomes active, using its current values for c_k^i, d_k^i, and h_k^i. If, at time $a'_{i,k}$, the capacity c_k^i was being stolen by an active overloaded server, capacity stealing is immediately interrupted.

While active, the behaviour of the non-isolated server S_i is equal to any other active isolated server in the system. As such, the accounting for the remaining capacity c_i is proven by Lemma 1.

Case b.

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

If the inequality $d_{k-1}^i \leq a_{i,k}$ holds, a new deadline $d_k^i = a_{i,k} + T_i$ is generated, the server's capacity c_k^i is recharged to its maximum value $c_k^i = Q_i$ and the replenishment time h_k^i is set to $h_k^i = d_k^i$. Otherwise, server S_i keeps its current values for c_k^i, d_k^i, and h_k^i.

At time $a_{i,k}$ the non-isolated server S_i becomes active and it is inserted into the ready queue. As such, its capacity c_k^i is consumed as follows from Lemma 1.

□

Theorem 1 *Let Γ_{hard} be a set of n periodic hard real-time tasks, with each*

task $\tau_i \in \Gamma_{hard}$ being scheduled by a dedicated isolated server S_i with a reserved capacity Q_i equal to the task's WCET and T_i equal to the task's period. Let Γ_{soft} be a set of soft real-time tasks scheduled by a group of isolated and non-isolated severs with total utilisation U_{soft}. Then Γ_{hard} is feasible if and only if

$$\sum_{\tau_i \in \Gamma_{hard}}^{n} \frac{Q_i}{T_i} + U_{soft} \leq 1$$

Proof

The theorem follows immediately from Lemma 1 and Lemma 2. In fact, Lemma 1 ensures that each generated isolated capacity is always exhausted before or discharged at its deadline. The same is true for any generated non-isolated capacity, according to Lemma 2.

Since the worst case response time of a hard task is independent of whether the reserved capacity of some server is being used by that server to execute its dedicated task or it is being consumed by any other server in the system, the system's schedulability is independent of whether the proposed dynamic capacity accounting mechanism of CSS 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.

□

6 Sharing resources in open systems

As discussed in the previous sections, CSS can effectively reduce the mean tardiness of periodic soft real-time tasks through an efficient management of unused reserved capacities under the assumption that tasks do not share any of the system's resources. In fact, if classic mutual exclusion semaphores are used with CSS, a particular problem arises, usually referred as priority inversion. If a higher priority task is blocked on a semaphore by a lower priority task and another medium priority arrives, the latter can preempt the lower priority task causing an unbounded blocking delay to the higher priority task [28].

A great amount of work has already been addressed to minimise the adverse effects of blocking when considering shared resources among tasks. Resource sharing protocols such as the Priority Ceiling Protocol [28], Dynamic Priority Ceiling [29], and the Stack Resource Policy [30] have been proposed to provide guarantees to hard real-time tasks accessing mutually exclusive resources. Based on these protocols, several scheduling solutions were already proposed [31; 32; 8; 33]. However, they cannot be directly applied to open real-time

systems since they all require a previous knowledge of the maximum resource usage for each task.

Resource sharing among tasks of open real-time systems started to be addressed in [18]. The proposed Bandwidth Inheritance (BWI) protocol extends the CBS scheduler to work in the presence of shared resources, adopting the Priority Inheritance Protocol (PIP) [28] to handle tasks' blocking. Although the PIP was initially thought in the context of fixed priority scheduling, it has been shown that it can be applied to dynamic priority scheduling, holding its basic properties: it limits the worst-case blocking that must be endured by a job $J_{i,k}$ to the duration of at most $min(n, m)$ critical sections where n is the number of jobs with lower priority than $J_{i,k}$ and m the number of different semaphores used by $J_{i,k}$.

The approach in BWI is that when a task executing in a lower priority server blocks a higher priority one it is inherited by the blocked server, allowing a task to be executed on more than its dedicated server, thus not requiring any prior knowledge about the tasks' structure and temporal behaviour while guaranteeing that tasks that do not access those shared resources are not affected by the behaviour of other tasks.

However, its main drawback is its unfairness in bandwidth distribution. A blocking task can use most (or all) of the reserved capacity of one or more blocked tasks, without compensating the tasks it blocked. Blocked tasks may then lose deadlines that could otherwise be met. At the same time, servers keep postponing their deadlines and recharging their capacities on every capacity exhaustion, potentially severely delaying blocked tasks with earlier deadlines which will finish later than tasks with longer deadlines. It is known 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 [16; 9]. This violation of the original capacity distribution can have a huge negative impact in the overall system's performance.

Figure 3 illustrates these problems with a simple example. Three servers $S_1 = (2, 5)$, $S_2 = (1, 3)$, and $S_3 = (1, 4)$ serve tasks τ_1, τ_2, and τ_3, respectively. Tasks τ_1 and τ_2 share access to resource R for the entire duration of their execution times, while τ_3 is independent from the other two.

Note how an early arrival of the second job of task τ_1 at time $t = 4$ allows τ_1 to consume 3 units of reserved bandwidth in the interval $[0, 5]$, more than its initial reservation. The nonexistence of a compensation mechanism and the automatically deadline update are responsible for the deadline miss of the second job of task τ_2.

To address this lack of a compensation mechanism, BWE [34] and CFA [35] try to fairly compensate blocked servers in exactly the same amount of capacity

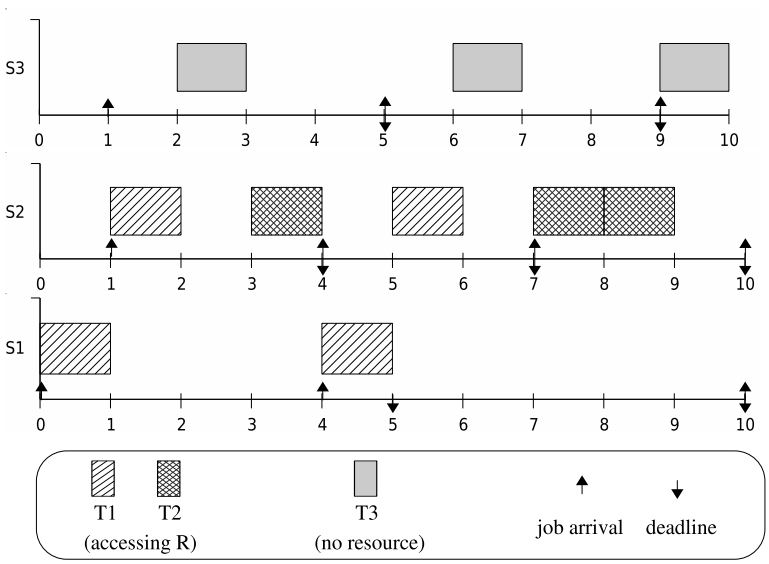

Fig. 3. BWI's drawbacks

that was consumed by a blocking task while executing in a blocked server. To achieve this, BWI maintains a global $n * n$ matrix (n is the number of servers in the system) in order to record the amount of capacity that should be exchanged between servers, a capacity list at each server to keep track of available budgets, and dynamically manages resource groups (groups of tasks that access a particular resource) at each blocking and releasing of a shared resource. CFA requires each server to manage two task lists with different priorities and a counter that keeps track of the amount of borrowed capacity from a higher priority server, converting the inheritor into a debtor. Contracted debts are payed by blocking servers, until the blocked servers' counters are successively decremented to zero.

The increased computational complexity of these attempts to fairly compensate borrowed capacities and the fact that CSS tends to fairly distribute residual capacities in the long run, lead us to propose a simple and efficient capacity exchange protocol that merges the benefits of a smart greedy capacity sharing policy with the concepts of bandwidth inheritance. Adding to the lower complexity of our approach, achieved results detailed in Section 9 demonstrate that taking advantage of all of the available capacity instead of only exchanging capacities within the same resource group leads to a better system's performance in dynamic open real-time systems.

6.1 The Capacity Exchange Protocol

The Capacity Exchange Protocol merges the benefits of the capacity sharing and stealing approach of CSS with the concept of bandwidth inheritance, allowing a task τ_i to be executed on more than its dedicated server S_i and ef-

ficiently exchanging capacities among servers to reduce the undesirable effects caused by inter-application blocking.

CXP adds, to each CSS server, a list of served tasks ordered by the tasks' deadlines. Initially, each server has only its dedicated task in the task list and, as long as no task is blocked, servers behave as in the original CSS scheduler. With blocking, the following rules are introduced:

- **Rule E (capacity inheritance):** When a high priority task τ_i is blocked by a lower priority task τ_j when accessing the shared resource R, τ_j is inherited by server S_i. The execution time of τ_j is now accounted to the currently pointed server by S_i. If task τ_j has not yet released the shared resource R when S_i exhausts all the capacity it can use, τ_j continues to be executed by the earliest deadline server with available capacity that needs to access R, until τ_j releases R.
- **Rule F (capacity inheritance compensation):** If a blocking task τ_j is inherited by a blocked server S_i, delaying the execution of its dedicated task τ_i, then τ_i is also added to S_j's task list. When task τ_i is unblocked it is executed by the earliest deadline server which has τ_i in its task list until it is finished or the server exhausts all the capacity it can use (whatever comes first).
- **Rule G (unfinished tasks inheritance):** If at time t, no active server with pending jobs can continue to execute through one of the rules B, C, or D, and there is at least one active server S_r with residual capacity greater than zero, it is possible to use those available residual capacities with deadlines greater than the one assigned to the current job $J_{p,k}$ of the earliest deadline server S_p with pending work to execute $J_{p,k}$ through bandwidth inheritance.

Rule E describes the integration of the bandwidth inheritance mechanism in the dynamic capacity accounting of CSS. The currently executing server always consumes the pointed capacity, either its own or another available valid capacity in the system.

Rule F proposes to exchange reserved capacities among servers due to blocking without the goal of a fair compensation, reducing the complexity and overhead of CXP. It allows a blocked task τ_i that has been delayed in its execution to be executed by the earliest deadline server with available capacity which has τ_i in its task list. Note that, with bandwidth inheritance, this server may now be different from S_i.

In general, the hard reservation approach may cause the loss of more deadlines since once a server's capacity is depleted, capacity recharging is suspended until the server's next activation. To minimise its drawbacks and take advantage of a more constant rate in tasks' execution, Rule G allows the use of bandwidth

inheritance to execute unfinished tasks, including those from servers that do not directly or indirectly share any resource with the selected server, if at a particular time no active server in the system is able to reclaim new residual capacities or steal inactive non-isolated capacities to continue executing its pending work after a capacity exhaustion.

Note that since the queue of active servers is ordered by deadlines, CXP easily keeps track of the earliest deadline server with pending work and no capacity left S_p, as well as the earliest deadline server with available residual capacity S_r, when traversing the queue to select the next running server. If the end of the active queue is reached without finding a server with pending work and available capacity, server S_r is selected as the running server and inherits the first task of S_p' list. S_r executes the task, consuming its own residual capacity. Since a server always starts to consume the earliest residual capacity available, no modification to the capacity accounting mechanism is needed to correctly account for the consumed capacity.

Note that Rules A and B of the original CSS scheduler ensure that residual capacities originated by earlier completions can be reclaimed by any active eligible server. Blocked servers can then take advantage of any residual capacity, even if it is released by a server that does not share any resource with the reclaiming server.

6.2 Minimising the cost of blocking with CXP

While preserving the isolation principles of independent tasks and inheritance properties of critical sections of BWI, CXP introduces significant improvements in the system's performance. Figure 4 illustrates how CXP can minimise the cost of blocking by efficiently exchanging reserved capacities among servers, scheduling the same set of tasks used to analyse the BWI's drawbacks in Figure 3.

At time $t = 1$, task τ_2 is added the task list of server S_1 (Rule F). At time $t = 2$, task τ_2 is unblocked and it is executed by server S_1, since it is the earliest deadline server with remaining capacity with τ_2 in its task list (the same happens at time $t = 8$). Note that capacities are exchanged between all the system's servers and not only within a specific resource group, maximising the use of extra capacities to handle overloads and still meet deadlines. An overload of the independent task τ_3 was handled by reclaiming the residual capacity originated by an earlier completion of task τ_1 at time $t = 12$.

Since the execution and inter-arrival times of soft real-time jobs are not known in advance it is important to minimise the impact of misbehaved tasks that exceed their expected execution times or have a shorter inter-arrival time of

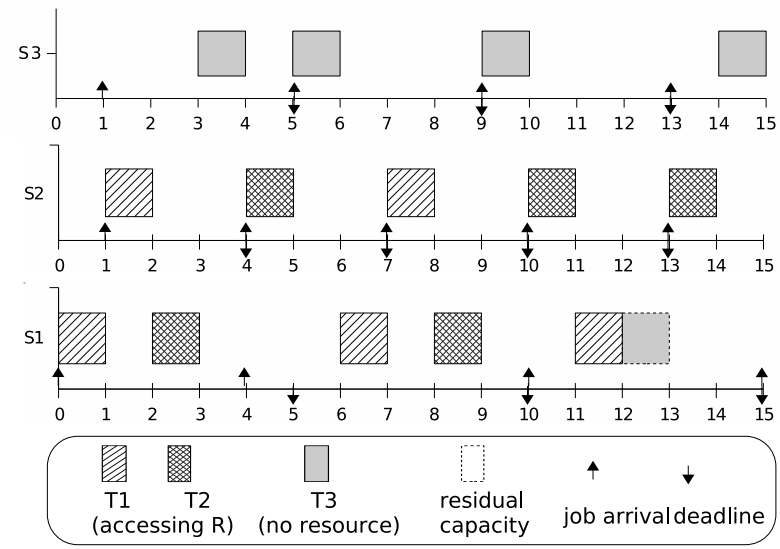

Fig. 4. Sharing resources with CXP

jobs. Note that despite the earlier arrival of the second job of task τ_1 at time $t = 4$, the deadline of server S_1 is not set to $d_2^1 = 9$ and the job is only released at time $t = 5$.

7 Handling precedence constraints in open systems

It is well known that precedence constraints, *i.e* when the execution of the data's producer must precede the execution of the consumer of that data, can be guaranteed in real-time scheduling by priority assignment. In fact, with dynamic scheduling, any task will always precede any other task with a later deadline. This suggests that precedence constraints that are consistent with the tasks' deadlines do not affect the schedulability of the task set.

In fact, the idea behind the consistency with the partial order is to enforce a precedence constraint by using an earlier deadline [36]. Formal work exists showing how to modify deadlines in a consistent manner so that EDF can be used without violating the precedence constraints. Garey et al. [37] show that the consistency of release times and deadlines can be used to integrate precedence constraints in the task model. Spuri and Stankovic [36] introduce the concept of quasi-normality to give more freedom to the scheduler so that it can also obey shared resource constraints, and provide sufficient conditions for schedules to obey a given precedence graph. The authors prove that with deadline modification and some type of inheritance it is possible to integrate precedence constraints and shared resources. Mangeruca et al. [38] consider situations where the precedence constraints are not all consistent with the tasks' deadlines and show how schedulability can be recovered by considering

a constrained scheduling problem based on a more general class of precedence constraint.

However, all these works base their modifications of deadlines on a previous knowledge of the tasks' execution times. To make use of these previous results in open real-time systems, the consistency of release times and deadlines with the partial order must be enforced considering estimated execution times when applying some known technique [37; 39; 38; 40; 41] at admission time. Nevertheless, such approach immediately raises two questions: (i) what happens if a precedent soft real-time task requires more than its reserved capacity? (ii) how can a task know if all its predecessors have already finished?

CXP provides answers for both questions and can be used to handle blocking due to precedence violations in the same way as for a critical section blocking, minimising the impact of misbehaved tasks on the overall system's performance. We base our approach on the idea that if task $\tau_j \prec \tau_i$ has not yet finished at time $s_{i,k}$, when the kth job of τ_i is selected to execute, it is blocking its successor.

Given a partial order $\prec$ on the tasks, described by a directed graph G, servers' state changes in CXP allow an easy verification of the current condition of a precedent task τ_j. Recall that a server that has completed its job is only kept active until its deadline if it is supplying residual capacity to other servers. By adding the following rule to CXP, we are able to handle precedence constraints among tasks of open real-time systems without any previous complete knowledge of their actual behaviour during runtime.

- **Rule H (unfinished precedent tasks):** If S_j, the dedicated server of a precedent task $\tau_j \prec \tau_i$, is active at time $s_{i,k}$ when server S_i is scheduled for execution, S_i checks the current value of S_j's residual capacity r_k^j. If it is set to zero, the current task τ_j of S_j has not yet been completed and must be added to S_i's task list.

Note that the addition of Rule H to CXP does not introduce any overhead. Since CXP reclaims available residual capacities as earlier as possible, whenever a server S_i is scheduled for execution it already checks the current state of the residual capacity of active earlier deadline servers. By combining this property with bandwidth inheritance, precedence constraints can be handled as an access to a shared resource in open real-time systems without introducing overhead in the protocol.

The next example illustrates how CXP can easily handle precedence constraints among tasks whose actual computation times are only revealed at run time. Figure 5 shows a possible scheduling of three servers $S_1 = (2, 8)$, $S_2 = (4, 10)$, and $S_3 = (3, 15)$ used to serve three periodic soft real-time tasks, based on their estimated average execution times and periods, exhibiting the following precedence constraints $\tau_1 \prec \tau_2 \prec \tau_3$.

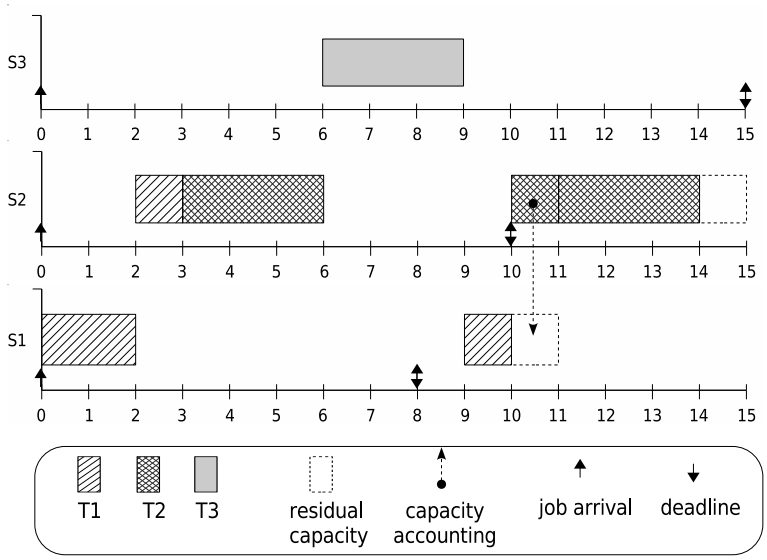

Fig. 5. Handling tasks' precedences with CXP

At time $t = 3$, the successor server S_2 knows it has to complete its predecessor's task since S_1 is still active and its residual capacity is set to zero. As such, task τ_1 needs to be executed in server S_2, prior to the execution of τ_2.

On the other hand, at times $t = 6$ and $t = 10$, both servers S_3 and S_1 can start executing their dedicated tasks. At time $t = 6$, S_2 becomes inactive by completing τ_2 and exhausting its capacity. Its inactive state clearly indicates that task τ_2 has been completed. Similarly, at time $t = 10$, the predecessor server S_1 is active but with residual capacity available. This is only possible when a server has completed its current task using less that its budgeted capacity, releasing residual capacity.

8 Theoretical validation for inter-dependent tasks

As shown, CXP is particularly suitable to schedule soft real-time tasks without requiring any offline knowledge of how many services will be concurrently executed neither which resources will be accessed and by how long they will

be held. However, enabling resource sharing among hard (HRT) and soft real-time (SRT) tasks in open systems is not straightforward. Demanding that SRT tasks declare the maximum duration of the critical sections on each accessed resource at admission time is against the basic purpose of an open system itself. Nevertheless, HRT tasks still need to be guaranteed based on the knowledge of their worst-case behaviour.

One way to achieve such guarantee in an open system is to implement the critical sections as library functions whose WCET can be determined. Of course, this comes at the cost of some pessimism. However, serving HRT tasks must always be based on a reserved capacity equal to their WCETs.

Furthermore, if nested critical sections are allowed, the system's libraries must also impose a totally ordered access to resources, since for a deadlock to be possible a blocking chain must exist in which there is a circular relationship. Deadlocks can be detected and exceptions raised if a misbehaving task attempts to acquire resources in a improper order, by following the chain of accessed resources and detecting a resource that is already in the list.

In the remaining of this section, we assume that resources are orderly accessed through shared libraries and discuss how to assign the maximum capacity Q_i and period T_i to an isolated server which has to serve a hard real-time task in an open system with n hard reservation servers with a total utilisation of $\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1$. We start by proving the correctness of the proposed capacity exchange mechanism of CXP.

Definition 5 *At a particular time instant t, the total amount of available execution capacity C_a in the system is the sum of the remaining reserved capacities greater than zero that can be used to execute a task (either the remaining execution or residual capacities of active servers or the remaining execution capacities of inactive non-isolated servers whose capacity can be stolen by active servers).*

Lemma 3 *Just after a task τ_i releases the shared resource R, the total amount of available execution capacity C_a in the systems is the same as in the non-resource sharing case.*

Proof

While task τ_i is accessing the shared resource R during t units of time, it can block some other task. It follows from the bandwidth inheritance protocol that when a task blocks another one it inherits the latter's server. Furthermore, as proven by Theorem 1, the dynamic budget accounting mechanism used in CXP does not affect the system's schedulability.

Hence, the total amount of available system's execution capacity C_a when

task τ_i releases the shared resource R is independent of whether the task was executed only by its dedicated server S_i 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.

□

Lemma 4 *No capacity is exchanged after its deadline.*

Proof

Let $a_{i,k}$ denote the time instant at which the kth instance of task τ_i arrives and its dedicated server S_i is inactive. At $a_{i,k}$, a new execution capacity $c_k^i = Q_i$ is always generated for an isolated server. For a non-isolated server S_i, if $a_{i,k} < d_k^i$, the server becomes active with the remaining execution capacity $c_k^i = Q_i - c_k^{i'}$, where $c_k^{i'}$ is the amount of reserved capacity stolen by overloaded active servers. Otherwise, it becomes active with its full reserved capacity $c_k^i = Q_i$.

Let $\forall_{i,k}\ d_k^i = max\{a_{i,k}, d_{k-1}^i\} + T_i$ be the deadline and $\forall_{i,k}\ h_k^i = d_k^i$ be the replenishment time associated with the generated capacity c_k^i.

Let L be the task list of server S_i. L is composed at least by jobs of task τ_i, but can also contain, due to blocking, inherited tasks (Rule E) and tasks that were delayed by the execution of τ_i in high priority servers (Rule F).

Let $[t, t + \Delta_t[$ denote a time interval during which server S_i is executing the earliest unblocked task of L, consuming its own reserved capacity c_k^i. Consequently, c_k^i must be decreased to $c_k^{i'} = c_k^i - \Delta_t \geq 0$, until it is exhausted.

Let $f_{i,k}$ denote the time instant when server S_i completes the last job of L. The remaining execution capacity $c_k^i > 0$ is released as residual capacity $r_k^i = c_k^i$ and c_k^i is set to zero. At the time instant $f_{i,k}$, the next active server S_j with pending work and remaining execution capacity is scheduled for execution according to the EDF policy. If the inequality $d_k^i \leq d_l^j$ holds, server S_j can use the released residual capacity r_k^i until its associated deadline d_k^i or r_k^i's exhaustion.

Let $[t, t + \Delta_t[$ denote a time interval during which server S_j is executing, consuming the residual capacity r_k^i. Consequently, the residual capacity r_k^i of server S_i must be decreased to $r_k^{i'} = r_k^i - \Delta_t \geq 0$, until it is exhausted.

If at some instant t all active servers have exhausted the amount of execution capacities they can use and there are unfinished jobs, the job of the earliest deadline unfinished task τ_u is added to the task list of the earliest deadline active server S_{edf} with residual capacity $r_k^{edf} > 0$. Assume that S_i is the selected

server.

Let $[t, t + \Delta_t[$ denote a time interval during which server S_i is executing, consuming its own residual capacity r_k^i. Consequently, r_k^i must be decreased to $r_k^{i'} = r_k^i - \Delta_t \geq 0$, until it is exhausted.

At replenishment time $t = h_k^i$ any remaining unused residual capacity r_k^i of server S_i is discharged.

□

Theorem 2 *Given a system with n servers with utilisation $U = \sum_{i=1}^{n} \frac{Q_i}{T_i}$ which uses CXP for accessing shared resources, it can be guaranteed that, at any time, the system's utilisation U is no more than the case when the served tasks do not share access to some resources.*

Proof

Without resource sharing, CXP ensures that no server consumes more than its reserved capacity Q_i every period T_i and the amount of capacity that can be reclaimed or stolen is limited in the worst case by the reserved capacity and deadlines of the pointed servers. By directly applying the results of Lemma 3 and Lemma 4, the same properties hold in CXP when tasks share access to resources.

□

Theorem 3 *A blocked task scheduled by CXP never has less available time to complete its execution than under the basic BWI protocol.*

Proof

From Rule F, CXP guarantees that a blocked task τ_i resumes its execution in the earliest deadline server which has τ_i in its task list, which may be different from its dedicated server S_i. With BWI, however, the blocked task τ_i is only able to resume its execution when its dedicated server S_i has no more blocking tasks in its task list and is the earliest deadline among all active servers.

As a consequence, the time that is available for a blocked task τ_i to complete its execution may be increased with CXP but never reduced when compared against BWI.

□

After proving the correctness of the capacity exchange mechanism of CXP, we now discuss how to provide guarantees to hard real-time tasks starting with some important definitions that help to clarify the following analysis.

Definition 6 *Two tasks are in the same resource group if they directly or indirectly share some resource.*

Definition 7 *Given a task τ_i served by server S_i, the blocking time B_i is defined as the maximum time that all other tasks can be executed by S_i, for each job of τ_i.*

Lemma 5 *Given a task τ_i served by server S_i, only tasks in the same resource group can be added to S_i's task list and contribute to B_i, for each instance of τ_i.*

Proof

Initially, each active server has exactly one task in its task list. It follows from the bandwidth inheritance protocol that if a task τ_i is blocked by task τ_j when accessing a resource R, then τ_j is added to the task list of server S_i. If τ_j is also blocked on another resource, the chain of blocking is followed and all the blocked tasks are added to S_i until a non-blocked task is reached. The task list of all other servers remains unchanged. Hence, the number of tasks that can contribute to B_i is restricted to those tasks that belong to the same resource group.

□

Theorem 4 *If a HRT task τ_i is served by an isolated server S_i with parameters (Q_i, T_i), where the reserved capacity $Q_i = C_i + B_i$ is determined by adding the WCET C_i of τ_i to the maximum blocking B_i that can be experienced by an instance of τ_i, and T_i is the minimum inter-arrival time of τ_i's jobs, then τ_i will meet its deadline.*

Proof

From Theorem 2 it follows that each isolated server S_i always receives Q_i units of execution capacity every T_i units of time. Lemma 5 assures that the set of tasks that can be executed by S_i is restricted to those tasks in the same resource group. Hence, if a HRT task τ_i does not access any shared resource it is not affected by the behaviour of other tasks. Therefore, if each instance of τ_i consumes up to $C_i \leq Q_i$ units of execution capacity and instances are separated at least by T_i, is is guaranteed that task τ_i finishes no later than S_i's capacity exhaustion and it will meet all its deadlines.

If a HRT task τ_i accesses some shared resources during its execution, we have to consider the maximum time that other tasks can be executed by S_i through bandwidth inheritance. It follows from Lemma 5 that whether task τ_i meets its deadline depends only on the timing requirements C_i of task τ_i and on the maximum blocking time B_i that can be experienced by each instance of task

τ_i. Hence, in order not to miss any deadline of a HRT task τ_i it is sufficient to assign a capacity of $Q_i = C_i + B_i$ to the isolated server S_i.

□

From Theorem 4 it is possible to derive sufficient conditions for the schedulability of HRT tasks scheduled by CXP. HRT tasks which do not access any shared resource can be guaranteed exactly like in the original CSS algorithm by assigning them to isolated servers with capacities $Q_i = C_i$, where C_i is the WCET of task τ_i, and periods T_i equal to the minimum inter-arrival times of τ_i's jobs. A HRT task τ_i which accesses shared resources during its execution can be guaranteed if it is assigned to an isolated server S_i whose capacity $Q_i = C_i + B_i$ also accounts for the maximum blocking time B_i that can be experienced by each instance of τ_i.

8.1 Blocking time computation

An exact computation of the worst-case blocking time B_i for a HRT task τ_i is a complex problem in open systems where the unpredictable behaviour of SRT tasks may cause the associated servers to exhaust their capacities while inside the critical sections, causing many possible situations in which a SRT task can block a HRT task. Without a complete knowledge of the number, type, and behaviour of tasks that may, directly or indirectly, interact through shared resources with a HRT task τ_i, it is impossible to perform an accurate offline analysis and compute the worst case blocking B_i that can be experienced by τ_i without imposing some pessimism.

The dynamic properties of an open real-time systems only allow us to assume that the WCET of the critical sections that may be accessed by any task through the system's libraries can be indirectly computed through an offline analysis of those shared libraries. With nested critical sections, the WCET must consider the worst possible path in the blocking chain. The reader may refer to [42] for an extensive survey of the current methods and tools to compute WCETs.

This may be considered too pessimistic since, to guarantee a set of n HRT tasks, the blocking times must all be summed together at admission time, but the dynamic nature of an open system and lack of information impose such pessimism. It is impossible to completely identify the conditions under which any task that is dynamically admitted in the system can interfere with a HRT task. Of course, this comes at the cost of a lower system's utilisation to guarantee HRT tasks. However, with CXP, SRT tasks can benefit from the unused reserved capacities of HRT tasks, minimising this waste of resources.

If a resource group is guaranteed to be composed only by HRT tasks, it is possible to explore all possible blocking situations and compute a more accurate and less pessimistic value for B_i, using, for example, an algorithm similar to the one presented in [43].

9 Evaluation

The performance of the proposed scheduling algorithms in dynamic open real-time environments was evaluated through extensive simulations with a special attention being devoted to introduce a high variability in the characteristics of the conducted simulations.

The reported results were observed from multiple and independent simulation runs, with initial conditions and parameters, but different seeds for the random values [1] used to drive the simulations, obtaining independent and identically distributed variables. Although the outputs of individual simulation runs are not independent, it is still possible to obtain independent observations across the results of several simulation runs (or simulation replicas) with a reasonably good statistical performance [45]. Each simulation replica ran until $t = 250000$, producing a large variety of inheritance and preemption situations among tasks, and was repeated several times to ensure that stable results were obtained.

The conducted experiments can be divided in two major sets. The first one evaluates the effectiveness of CSS in reducing the mean tardiness of independent periodic tasks. It starts by comparing the performance of CSS against other similar approaches considering only sets of isolated servers in Section 9.1, while Section 9.2 details the impact of allowing overload servers to steal inactive non-isolated capacities in the improvement of the overall system's performance.

The second set evaluates how the proposed flexible management of reserved capacities of CXP can minimise the degree of deviation from the ideal system's behaviour caused by inter-application blocking due to shared resources (Section 9.3) or precedence constraints (Section 9.4).

[1] The random values were generated by the Mersenne Twister algorithm [44] with an uniform distribution.

Similarly to CSS, CASH [6] and BACKSLASH [9] also greedily assign residual capacities as early as possible to the highest priority server but propose different approaches to deal with a server's capacity exhaustion. The first conducted study evaluated the effect of those approaches in lowering the mean tardiness of independent periodic jobs. The mean tardiness was determined by $\sum_{i=0}^{n} trd_i/n$, where trd_i is the tardiness of task τ_i, and n the number of periodic tasks. For a fair comparison, only isolated servers were used with CSS.

Random workloads were created in order to evaluate the performance of each algorithms when the tasks' parameters differ in open real-time scenarios with a high variability in the jobs' overload probabilities. 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 $[0.7Q_i, 1.4Q_i]$ of its dedicated server's reserved capacity Q_i.

Figure 6 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.

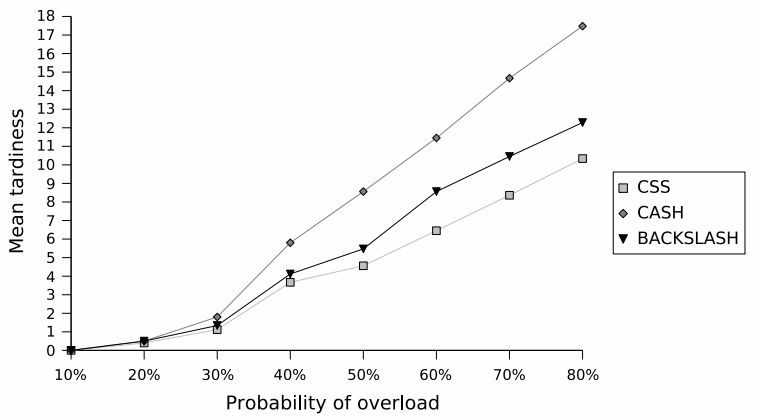

Fig. 6. Performance in dynamic scenarios

As expected, all the algorithms perform better when there is more residual capacity available to handle overloads. Furthermore, they all behave very similarly when tasks have a lower probability (until near 30%) of exhausting their servers' reserved capacity. The behaviour of a server is determined by two parameters: (i) the server's reserved capacity, which defines the fraction of the processor allocated to the task it is serving; and (ii) the server's period, which defines the time granularity of the allocation. As such, without applying any technique to dynamically adapt the server's parameters based on the average

response times of the served tasks, as for example the one proposed in [19], it is clear that the system's performance will severely decrease as the probability and size of tasks' overloads increases.

Nevertheless, without such adaptation, the approach follow CSS outperforms the other algorithms in lowering the mean tardiness of periodic jobs with increased probabilities of jobs' overloads. Recall that CASH and BACKSLASH automatically update a server's capacity and deadline on every capacity exhaustion. As these results demonstrate, 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, particularly with a high probability of overloads.

Furthermore, by keeping a depleted server active until its deadline allows it to take advantage of new residual capacities released by earlier completions of other servers, which also contributes to the better results achieved by CSS. Recall that while BACKSLASH and CSS do share the same concept of using original deadlines for residual capacity reclaiming, a CSS server does not use capacities reserved for future jobs of its dedicated task.

9.2 Allowing capacity stealing

A second study evaluated the impact 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 $[0.1, 1.0]$. 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 $[0.8Q_i, 1.2Q_i]$ of its dedicated server's reserved capacity Q_i.

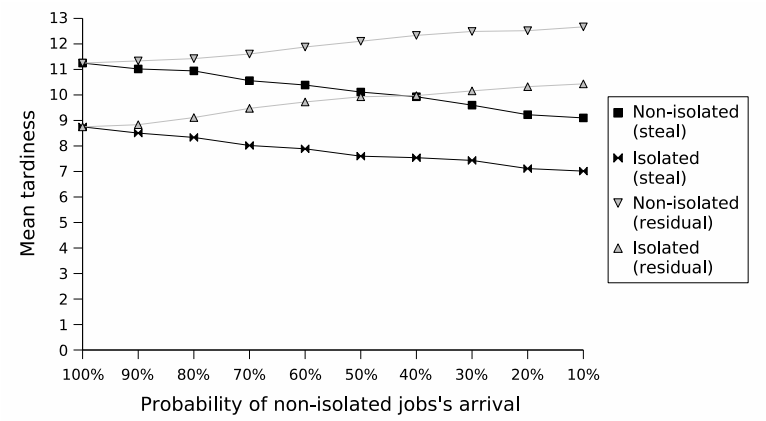

Fig. 7. Small variation in execution times

The achieved results are shown in Figure 7. As expected, when overloaded active servers have more opportunities to steal non-isolated capacities, the obtained mean tardiness 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 a low variation in the jobs' computation times, the ability to steal non-isolated capacity achieves better results, although the single use of an efficient residual capacity reclaiming mechanism is able to achieve a similar, albeit lower, performance.

Furthermore, Figure 7 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 the now active non-isolated server cannot reclaim any available residual capacity, the job must be executed with less capacity than expected, probably resulting in a deadline miss. 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 [20]. 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 $[0.6Q_i, 1.8Q_i]$ of the dedicated server's reserved capacity Q_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 8 clearly shows a perceptibly improved system's performance when it is possible to steal inactive non-isolated capacities in the presence of a large variation in jobs' computation times. One can conclude

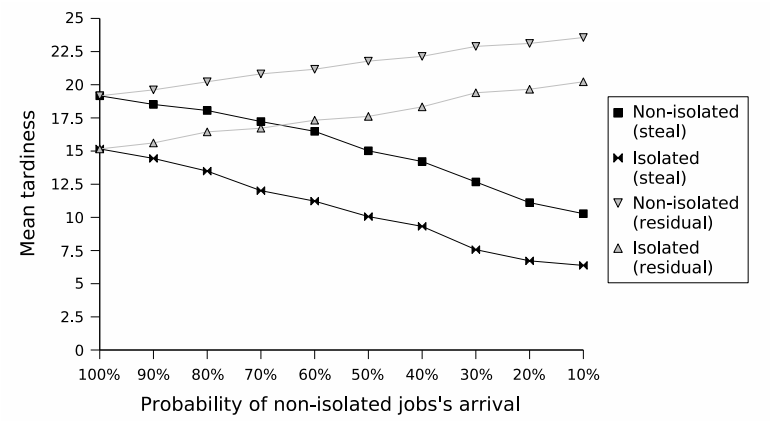

Fig. 8. Large variation in execution times

that, with CXP, severe overloads can be efficiently handled through a residual capacity reclaiming and non-isolated capacity stealing approach, reducing the mean tardiness of periodic jobs.

9.3 Sharing resources among tasks

The first conducted study compared the cumulative capacity that was consumed by the shortest period (SP) and longest period (LP) tasks of a randomly generated task set when tasks share resources to the amount of capacity that would be consumed if the same set of tasks did not shared any resources. The cumulated capacities consumed by the SP and LP tasks were recorded every 200 time ticks and the mean values of all generated samples plotted in Figures 9 and 10, respectively.

Different sets of 5 tasks were randomly generated, with varied execution requirements ranging from 20 to 60 units, and period distributions ranging from 100 to 300 time units, always ensuring a system's utilisation $U \leq 1$. An isolated server was assigned to each task, with a reserved capacity Q_i equal to the task's execution requirements and period T_i equal to the task's period. The execution requirements of each job were always equal to the reserved capacity of its dedicated server and all jobs accessed the shared resource R during all their executions, with a new job being released immediately after a task has completed its current job.

The achieved results show that with BWI, and due to blocking, while higher priority tasks can consume less than their initial allocations, tasks with longer deadlines can consume more than their reserved capacities since BWI is affected by the absence of a compensation mechanism. In contrast, the efficient capacity exchange mechanism of CXP ensures that both tasks are able to get their allocated capacities even when accessing shared resources thus providing

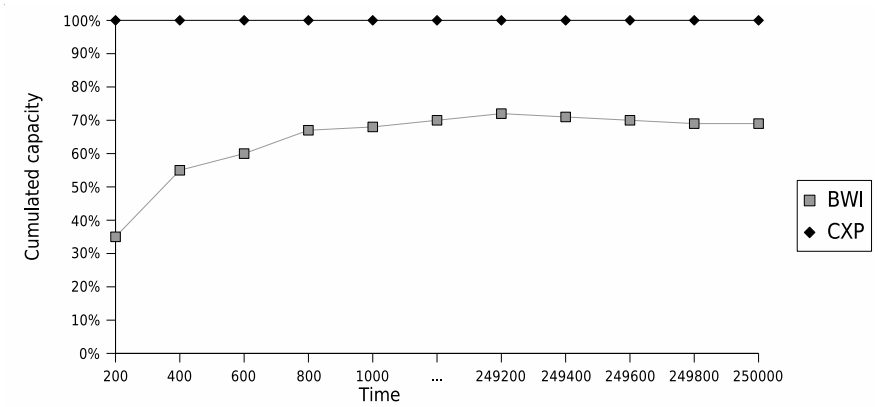

Fig. 9. Capacity consumed by the SP task

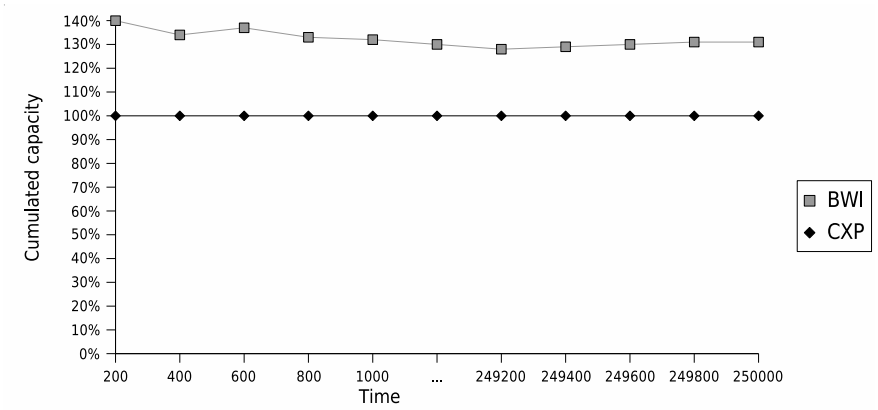

Fig. 10. Capacity consumed by the LP task

a better fairness than BWI and sustaining the conclusions drawn from the examples in Section 6.

A second study compared the efficiency of the studied protocols BWI, BWE, CFA and CXP in lowering the mean tardiness of a set of periodic jobs with variable execution times in highly dynamic scenarios. At each simulation run, a random number of servers with a system's utilisation up to 70% contended for the system's resources with a dynamic traffic that demanded up to 30% of the system's capacity. Resource sharing protocols that require a prior knowledge of the maximum resource usage time for each task such as the Priority Ceiling Protocol, the Dynamic Priority Ceiling, or the Stack Resource Policy were not considered in the studies since they cannot be directly applied to open real-time systems.

All servers were generated with varied reserved capacities Q_i ranging from 15 to 50 units of execution and period distributions ranging from 50 to 500 time units, creating different types of load, from short to long deadlines and capacities. Tasks arrived at randomly generated times and remained in the system

for a variable period of time with each job having an execution time in the range $[0.8Q_i, 1.2Q_i]$ of its dedicated server's reserved capacity Q_i, originating both overloads and residual capacities due to early completions. There were 6 resources, whose access and duration of use was randomly distributed by the servers, creating direct and transitive blocking situations and distinct resource groups. For a fair comparison, only isolated servers were used in CXP.

Figure 11 illustrates the performance of the evaluated protocols as a function of the system's load, measuring the mean tardiness of periodic tasks under random workloads for different probabilities of jobs' overload.

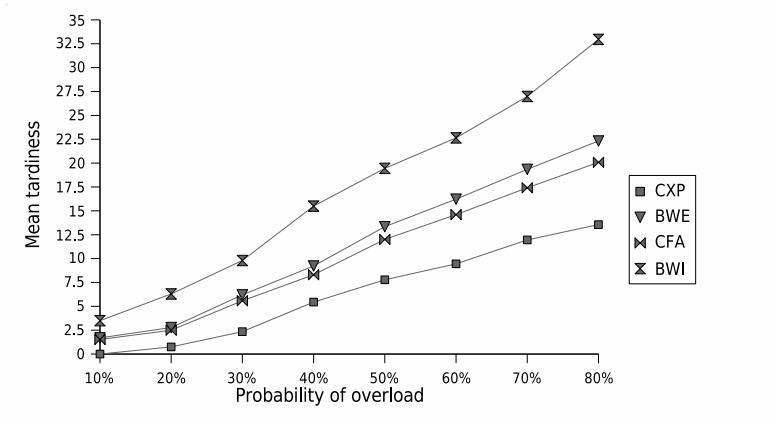

Fig. 11. Performance in dynamic scenarios

As expected, the achieved results clearly justify the use of a capacity exchange mechanism to minimise the impact of blocking on the system's performance. Without any compensation for the extra work on blocked servers, BWI obtains the poorest result. Recall that with BWI, a blocked task is only able to use the remaining capacity of its dedicated server, if any.

BWE and CFA achieve similar performances when handling tasks with variable execution times. Both algorithms are unable to reclaim residual capacities originated by early completions, wasting available resources to handle overloads and minimise the number of deadline misses. Also, both algorithms immediately recharge a server's capacity and extend its deadline at every capacity exhaustion, allowing a task to use resources allocated to a future job, contributing for future jobs of that task to miss their deadlines by larger amounts.

On the other hand, by reclaiming as much extra capacity as possible, CXP outperforms BWE and CFA in lowering the mean tardiness of periodic tasks in highly dynamic scenarios. CXP not only exchanges capacities between all active servers, not restricting capacity exchange to the same resource group, but it also reclaims all the available residual capacity to handle overloads of soft real-time tasks.

Furthermore, these better results in highly dynamic scenarios were achieved with a less complex approach to exchange reserved capacities among servers. Figure 12 illustrates the average overhead introduced by the optimisations of BWE, CFA, and CXP in terms of the needed scheduling time and memory consumption during the previous study, using the base BWI protocol as a reference.

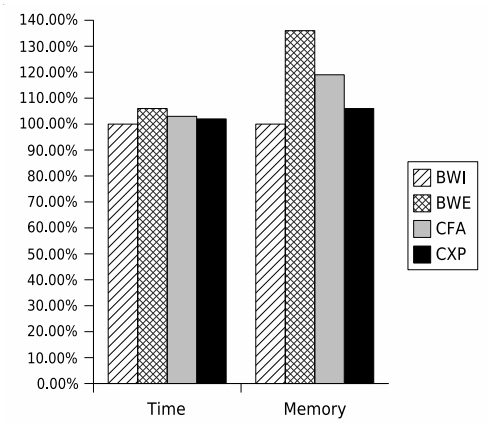

Fig. 12. Overhead using BWI as reference

As expected, the optimisations performed by BWE, CFA, and CXP introduce some overhead when compared against BWI in terms of needed time and memory. Although all the three algorithms need only slightly more time than BWI to determine which capacity is going to be accounted by the currently executing server, they substantially differ in terms of storage information demands. BWE requires a global $n * n$ matrix to record the amount of capacity that must be exchanged between servers and an extra list at each server to keep track of available capacities. CFA enhances BWI by adding a new task queue to each server and one extra variable for each contracted debt between servers S_i and S_j. On the other hand, CXP focuses on minimising the cost of blocking by exchanging reserved capacities as early, and not necessarily as fairly, as possible. As such, it does not not account the amount of borrowed capacity on each server neither manages individual resource groups.

9.4 Precedence constraints

Another study compared the time and memory needed by CXP to schedule the same task set with and without precedence constraints among its tasks. 10000 tasks sets were randomly generated, with different system's utilisation in the range $[0.6, 1.0]$. For each task set, a random set of precedence constraints consistent with the tasks' deadlines was determined. Each job had random execution requirements in the range $[0.7Q_i, 1.3Q_i]$ of its dedicated server's reserved capacity Q_i.

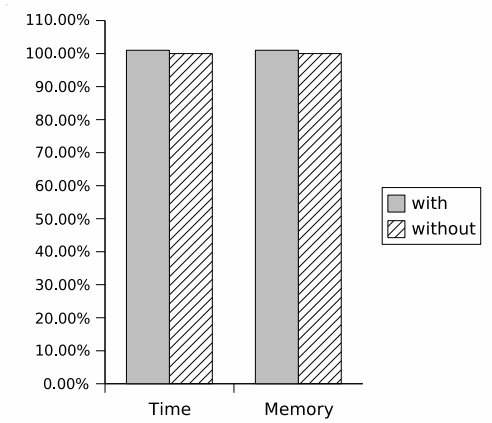

Fig. 13. Overhead of handling precedence constraints

The achieved results, plotted in Figure 13, allow us to conclude that CXP is able to efficiently handle precedence constraints among tasks whose exact behaviour is not known beforehand without any significant overhead. Recall that precedence constraints are handled by CXP as an access to a shared resource and the proposed residual capacity reclaiming policy already checks the current state of earlier deadline servers, since residual capacities are consumed before the server's reserved capacity.

10 Conclusions

This paper integrates and extends recent advances in dynamic deadline scheduling with resource reservation. Namely, while achieving isolation among tasks, the proposed Capacity Sharing and Stealing (CSS) approach can efficiently reclaim residual capacities originated by earlier completions and steal reserved unused capacities from inactive non-isolated servers, effectively reducing the mean tardiness of soft real-time tasks.

CSS is then combined with the concept of bandwidth inheritance to tackle the challenging problem of how to schedule tasks that share resources and exhibit precedence constraints without any previous knowledge of how many services will need to be concurrently executed neither which resources will be accessed and by how long they will be held. Rather than trying to account borrowed capacities and exchanging them later in the exact same amount, the proposed Capacity Exchange Protocol (CXP) focus on greedily exchanging extra capacities as early, and not necessarily as fairly, as possible and introduces a novel approach to integrate precedence constraints into the task model.

The achieved results clearly justify the use of a capacity exchange mechanism that reclaims as much capacity as possible and does not restrict itself to exchange capacities only within a resource sharing group. It is proven that CXP

achieves a better system's performance when compared against other available solutions and has a lower overhead.

References

[1] A. Colin, S. M. Petters, Experimental evaluation of code properties for wcet analysis, in: Proceedings of the 24th IEEE RTSS, 2003, pp. 190–199.

[2] C. W. Mercer, S. Savage, H. Tokuda, Processor capacity reserves: Operating system support for multimedia applications, in: Proceedings of the IEEE International Conference on Multimedia Computing and Systems, 1994, pp. 90–99.

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

[4] L. Abeni, G. Buttazzo, Integrating multimedia applications in hard real-time systems, in: Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain, 1998, p. 4.

[5] G. Lipari, S. Baruah, Greedy reclamation of unused bandwidth in constant-bandwidth servers, in: Proceedings of the 12th EuroMicro Conference on Real-Time Systems, Stockholm, Sweden, 2000, pp. 193–200.

[6] M. Caccamo, G. Buttazzo, L. Sha, Capacity sharing for overrun control, in: Proceedings of 21th IEEE RTSS, Orlando, Florida, 2000, pp. 295–304.

[7] L. Marzario, G. Lipari, P. Balbastre, A. Crespo, Iris: A new reclaiming algorithm for server-based real-time systems, in: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada, 2004, p. 211.

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

[9] C. Lin, S. A. Brandt, Improving soft real-time performance through better slack reclaiming, in: Proceedings of the 26th IEEE RTSS, 2005, pp. 410–421.

[10] N. Hawes, Anytime deliberation for computer game agents, Ph.D. thesis, School of Computer Science, The University of Birmingham (November 2003).

[11] M. Agrawal, D. Cofer, T. Samad, Real-time adaptive resource management for advanced avionics, IEEE Control Systems Magazine 23 (1) (2003) 6–86.

[12] J. Shackleton, D. Cofer, S. Cooper, Anytime scheduling for real-time embedded control applications, in: Proceedings of the 23rd Digital Avionics Systems Conference, Vol. 2, Salt Lake City, UT, USA, 2004, pp. 101–110.

[13] R. Bhattacharya, G. J. Balas, Anytime control algorithm: Model reduction approach, Journal of Guidance, Control, and Dynamics 27 (5) (2004) 767–776.

[14] J. van den Berg, D. Ferguson, J. Kuffner, Anytime path planning and replanning in dynamic environments, in: Proceedings of the IEEE International Conference on Robotics and Automation, Orlando, Florida, USA, 2006, pp. 2366– 2371.

[15] L. Nogueira, L. M. Pinho, Time-bounded distributed qos-aware service configuration in heterogeneous cooperative environments, Journal of Parallel and Distributed Computing 69 (6) (2009) 491–507.

[16] L. Nogueira, L. M. Pinho, Capacity sharing and stealing in dynamic server-based real-time systems, in: Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium, Long Beach,CA,USA, 2007, p. 153.

[17] L. Nogueira, L. M. Pinho, Shared resources and precedence constraints with capacity sharing and stealing, in: Proceedings of the 22th IEEE International Parallel and Distributed Processing Symposium, Miami,Florida,USA, 2008, p. 97.

[18] G. Lamastra, G. Lipari, L. Abeni, A bandwidth inheritance algorithm for real-time task synchronization in open systems, in: Proceedings of the 22nd IEEE Real-Time Systems Symposium, London, UK, 2001, pp. 151–160.

[19] G. Buttazzo, E. Bini, Optimal dimensioning of a constant bandwidth server, in: Proceedings of the 27th IEE International Real-Time Systems Symposium, Rio de Janeiro, Brasil, 2006, pp. 169–177.

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

[21] G. Bernat, I. Broster, A. Burns, Rewriting history to exploit gain time, in: Proceedings of the 25th IEEE RTSS, 2004, pp. 328–225.

[22] L. Abeni, C. Scordino, G. Lipari, P. L., Serving non real-time tasks in a reservation environment, in: Proceedings of the 9th Real-Time Linux Workshop, Linz, Austria, 2007, pp. 1–10.

[23] R. Rajkumar, K. Juvva, A. Molano, , 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, pp. 150–164.

[24] J. P. Lehoczky, S. Ramos-Thuel, An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems, in: Proceedings of the 13th RTSS, 1992, pp. 110–123.

[25] R. I. Davis, K. W. Tindell, A. Burns, Scheduling slack time in fixed priority preemptive systems, in: Proceedings of the 14th RTSS, 1993, pp. 222–231.

[26] R. I. Davis, Approximate slack stealing algorithms for fixed priority preemptive systems, Tech. rep., Department of Computer Science, University of York (November 1993).

[27] L. Abeni, Server mechanisms for multimedia applications, Tech. rep., Scuola Superiore S. Anna (1998).

[28] L. Sha, R. Rajkumar, J. P. Lehoczky, Priority inheritance protocols: an

approach to real-time synchronisation, IEEE Transaction on Computers 39 (9) (1990) 1175–1185.

[29] M.-I. Chen, K.-J. Lin, Dynamic priority ceilings: a concurrency control protocol for real-time systems, Real-Time Systems 2 (4) (1990) 325–346.

[30] T. P. Baker, A stack-based resource allocation policy for realtime processes., in: Proceedings of the IEEE Real-Time Systems Symposium, Lake Buena Vista, Florida, USA, 1990, pp. 191–200.

[31] K. Jeffay, Scheduling sporadic tasks with shared resources in hard-real-time systems, in: Proceedings of the IEEE Real-Time Systems Symposium, Phoenix, Arizona, USA, 1992, pp. 89–99.

[32] M. Caccamo, L. Sha, Aperiodic servers with resource constraints, in: Proceedings of the 22nd IEEE Real-Time Systems Symposium, London, UK, 2001, pp. 161–170.

[33] S. K. Baruah, Resource sharing in edf-scheduled systems: A closer look, in: Proceedings of the 27th IEEE Real-Time Systems Symposium, Rio de Janeiro,Brazil, 2006, pp. 379–387.

[34] S. Wang, K.-J. Lin, S. Peng, Bwe: A resource sharing protocol for multimedia systems with bandwidth reservation, in: Proceedings of the 4th IEEE International Symposium on Multimedia Software Engineering, New-port Beach,CA,USA, 2002, pp. 158–165.

[35] R. Santos, G. Lipari, J. Santos, Scheduling open dynamic systems: The clearing fund algorithm, in: Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications, Gothenburg, Sweden, 2004, pp. 114–129.

[36] M. Spuri, J. A. Stankovic, How to integrate precedence constraints and shared resources in real-time scheduling, IEEE Transactions on Computers 43 (12) (1994) 1407–1412.

[37] M. R. Garey, D. S. Johnson, B. B. Simons, R. E. Tarjan, Scheduling unit-time tasks with arbitrary release times and deadlines, SIAM Journal on Computing 10 (2) (1981) 256–269.

[38] L. Mangeruca, A. Ferrari, A. L. Sangiovanni-Vincentelli, Uniprocessor scheduling under precedence constraints, in: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, San Jose, CA, USA, 2006, pp. 157–166.

[39] M. Spuri, G. Buttazzo, Efficient aperiodic service under earliest deadline scheduling, in: Proceedings of the 15th IEEE Real-Time System Symposium, San Juan, Puerto Rico, 1994, pp. 2–11.

[40] J. Blazewicz, Scheduling dependent tasks with different arrival times to meet deadlines, in: Proceedings of the International Workshop on Modelling and Performance Evaluation of Computer Systems, Ispra,Italy, 1977, pp. 57–65.

[41] H. Chetto, M. Silly, T. Bouchentouf, Dynamic scheduling of real-time tasks under precedence constraints, Real-Time Systems 2 (3) (1990) 181–194.

[42] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley,

G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, The worst-case execution time problem - overview of methods and survey of tools, ACM Transactions on Embedded Computing Systems.

[43] G. Lipari, G. Lamastra, L. Abeni, Task synchronization in reservation-based real-time systems, IEEE Transactions on Computers 53 (12) (2004) 1591–1601.

[44] M. Matsumoto, T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation (TOMACS) 8 (1) (1998) 3–30.

[45] A. M. Law, W. D. Kelton, Simulation modeling and analysis, 3rd Edition, McGraw-Hill, 2000.