Technical Report

Shared Resources and Precedence Constraints with Capacity Sharing and Stealing

Luís Miguel Nogueira
Luís Miguel Pinho

HURRAY-TR-080202
Version: 0
Date: 02-11-2008

Shared Resources and Precedence Constraints with Capacity Sharing and Stealing

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

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

Abstract

This paper proposes a new strategy to integrate shared resources and precedence constraints among real-time tasks, assuming no precise information on critical sections and computation times is available. The concept of bandwidth inheritance is combined with a greedy capacity sharing and stealing policy to efficiently exchange bandwidth among tasks, minimising the degree of deviation from the ideal system's behaviour caused by inter-application blocking. The proposed Capacity Exchange Protocol (CXP) focus on exchanging extra capacities as early, and not necessarily as fairly, as possible. This loss of optimality is worth the reduced complexity as the protocol's behaviour nevertheless tends to be fair in the long run and outperforms other solutions in highly dynamic scenarios, as demonstrated by extensive simulations.

© IPP Hurray! Research Group
www.hurray.isep.ipp.pt

Shared Resources and Precedence Constraints with Capacity Sharing and Stealing

Luís Nogueira, Luís Miguel Pinho
IPP Hurray Research Group
School of Engineering (ISEP), Polytechnic Institute of Porto (IPP), Portugal
{luis,lpinho}@dei.isep.ipp.pt

Abstract

This paper proposes a new strategy to integrate shared resources and precedence constraints among real-time tasks, assuming no precise information on critical sections and computation times is available. The concept of bandwidth inheritance is combined with a greedy capacity sharing and stealing policy to efficiently exchange bandwidth among tasks, minimising the degree of deviation from the ideal system's behaviour caused by inter-application blocking. The proposed Capacity Exchange Protocol (CXP) focus on exchanging extra capacities as early, and not necessarily as fairly, as possible. This loss of optimality is worth the reduced complexity as the protocol's behaviour nevertheless tends to be fair in the long run and outperforms other solutions in highly dynamic scenarios, as demonstrated by extensive simulations.

1 Introduction

The resource reservation approach is particularly interesting to open real-time systems where new services can enter the system at any time without any previous knowledge about their execution requirements and tasks' inter-arrival times. Tasks can be accepted based only on expected requirements and handled through dedicated servers that prevent the served tasks from demanding more than the reserved amount.

The Capacity Sharing and Stealing (CSS) scheduler [14] was proposed to better handle soft tasks' overloads in highly dynamic open systems, effectively lowering the mean tardiness of periodic guaranteed services. It offers the flexibility to consider the coexistence of guaranteed and best-effort servers in the same system, reducing isolation in a controlled fashion in order to donate reserved, but still unused capacities to currently overloaded servers. However, tasks were assumed to be independent. A challenging problem in open real-time systems is how to schedule tasks that share resources and exhibit precedence constraints, without a complete knowledge about their behaviour.

The purpose of this paper is to address both problems, proposing the Capacity Exchange Protocol (CXP), integrating the concept of bandwidth inheritance [9] with the greedy capacity sharing and stealing policy of CSS. Rather than trying to account borrowed capacities and exchanging them later in the exact same amount, CXP focus on greedily exchanging extra capacities as early, and not necessarily as fairly, as possible. The achieved results suggest that the followed approach effectively minimises the impact of bandwidth inheritance on blocked tasks, outperforming other available solutions.

2 System model

This work focus on dynamic open real-time systems. Services can arrive at any time, requesting a previously unknown amount of the system's resources and stay in the system for an unknown period of time. If, given the current system's load, the request can be guaranteed, the service is accepted and the requested amount is reserved.

A service can be composed by a set of real-time and non-real-time tasks which 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 $r_{i,j}$, and starts to be executed at time $s_{i,j}$ with deadline $d_{i,j} = a_{i,j} + T_i$, with T_i being the period of τ_i. The arrival time of a particular job is only revealed during execution, and the exact execution requirements $e_{i,j}$ and which resources will be accessed and by how long will be held can only be determined by actually executing the job to completion until time $f_{i,j}$. These times are characterised by the relations $a_{i,j} \leq r_{i,j} \leq s_{i,j} \leq f_{i,j}$.

Tasks may simultaneously need exclusive access to one or more of the system's resources R, during part or all of their execution. If task τ_i is using resource R_i, it locks that resource. Since no other task can access R_i until it is re-

leased 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.

Tasks may also exhibit precedent 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 a 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 graph 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 the 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 $d_{i,j} < d_{k,j}$.

Tasks are associated to servers characterised by a pair (Q_i, T_i), where Q_i is the server's maximum reserved capacity and T_i its period. These values are based on average estimations for soft tasks. 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, independently of the behaviour of other tasks in the system. This may be possible in open systems if resources are orderly accessed through library functions whose WCET can be determined. Please refer to [13] for a detailed analysis.

3 Capacity Sharing and Stealing

The CSS scheduler [14] extends CBS [1] to efficiently handle soft-tasks' overloads in highly dynamic open real-time systems, effectively minimising the mean tardiness of guaranteed periodic jobs. It offers the flexibility to consider the coexistence of guaranteed *isolated* and best-effort *non-isolated* servers, combining the ability to efficiently reclaim unused allocated capacities when jobs complete in less than their budgeted execution time with the ability to steal reserved capacities from inactive non-isolated servers used to schedule sporadic best-effort jobs.

CSS reduces isolation in a controlled fashion in order to donate reserved, but still unused, capacities to currently overloaded servers. For an isolated server, a specific amount of the CPU is ensured to be available every period. On the other hand, an inactive non-isolated server can have some or all of its reserved capacity stolen by active overloaded servers.

A server S_i is *active* at instant t if (i) the served task is ready to execute; (ii) is executing; or (iii) the server is supplying its residual capacity to other servers until its deadline. S_i is *inactive* if (i) there are no pending jobs to serve; and (ii) the server has no residual capacity to supply to the other servers.

State transitions are determined by the (i) arrival of a new job, (ii) capacity exhaustion, or (iii) non-existence of pending jobs at replenishment time. An inactive server becomes active with the arrival of the new j^{th} job at time $a_{i,j}$, if $a_{i,j} \geq d_{i,j-1}$. If $a_{i,j} < d_{i,j-1}$, the job is only released at the next S_i's replenishment instant r_i. On the other hand, an active server becomes inactive if (i) all its reserved capacity is consumed and there are no pending jobs to serve (capacity exhaustion can occur while supplying its residual capacity to other servers or using its capacity to finish a job); or (ii) there are no pending jobs at replenishment time.

To eliminate the need of extra queues or additional servers' states to dynamically account consumed capacities, each server keeps a pointer to the currently consumed capacity, ensuring that at time t, the currently executing server S_i is using a residual capacity c_r originated by an early completion of another active server, its own reserved capacity c_i, or is stealing capacity c_s from an inactive non-isolated server (by that order). The server to which the accounting is going to be performed is dynamically determined at the time instant when a capacity is needed, using the following rules:

- **Rule A:** Whenever a server S_j completes its kth job and there is no pending work, its remaining capacity $c_j > 0$ is released as residual capacity $c_r = c_j$ that can immediately be reclaimed by eligible active servers, until the currently assigned S_j's deadline $d_{j,k}$. S_j is kept active with its current deadline.

- **Rule B:** The next server S_i scheduled for execution points to the earliest deadline server S_r from the set of eligible active servers with residual capacity $c_r > 0$ and deadlines $d_r \leq d_{i,k}$. S_i consumes the pointed residual capacity c_r, running with the deadline d_r of the pointed server. Whenever c_r is exhausted and there is pending work, S_i disconnects from S_r and selects the next available server S'_r (if any).

- **Rule C:** If all available residual capacities are exhausted and the current k^{th} job is not complete, the server consumes its own reserved capacity c_i either until job's completion or c_i's exhaustion. On a c_i's exhaustion, S_i is kept active with its current deadline $d_{i,k}$.

- **Rule D:** With pending work and no reserved capacity left, S_i connects to the earliest deadline server S_s from the set of eligible inactive non-isolated server with remaining capacity $c_s > 0$ and deadlines $d_s \leq d_{i,k}$. S_i steals the pointed inactive capacity c_s, running with its current deadline $d_{i,k}$. Whenever c_s is exhausted and the job has not been completed, the next non-isolated capacity c'_s is used (if any).

Note that at a particular time t there is only one server pointing to another server. Also note that a CSS server suspends its capacity recharging and deadline update until a specific replenishment time r_i, set to the current server's deadline, implementing a hard reservation (refer to [16] for a description of hard vs soft reservations). At replenishment time r_i, unconsumed capacities are discarded.

As jobs' execution requirements are not known beforehand, it makes sense to devote as much excess capacity as possible to the currently executing server, maximising its chances to complete the current job before its deadline (Rule B). A greedy capacity reclaiming has a reduced computational complexity and minimises deadline postponements and the number of preemptions [11].

When all valid residual capacities and the reserved capacity of server S_i are exhausted and there is still pending work, S_i is allowed to steal inactive non-isolated capacities to handle its current overload (Rule D). However, capacity stealing is interrupted whenever S_i is preempted or a replenishment event occurs on the capacity being stolen. Also, since S_i keeps its current deadline $d_{i,k} \geq d_s$ when stealing non-isolated capacities, capacity stealing is also interrupted when a new job for the inactive non-isolated server S_s arrives. Naturally, S_s becomes active with its current remaining capacity.

CSS (i) achieves isolation among guaranteed tasks; (ii) efficiently reclaims unused computation time, exploiting early completions; (iii) allows an overloaded server to steal reserved capacities from inactive non-isolated servers; and (iv) reduces the mean tardiness of periodic guaranteed jobs by assigning all available capacity to the currently executing server.

4 Sharing resources in open systems

A great amount of work has been addressed to minimise the adverse effects of blocking when considering shared resources among tasks. Resource sharing protocols such as the Priority Ceiling Protocol [18], Dynamic Priority Ceiling [6], and Stack Resource Policy [2] have been proposed to provide guarantees to hard real-time tasks accessing mutually exclusive resources. Solutions based on these protocols were already proposed [8, 5, 4, 3] but they all require a prior knowledge of the maximum resource usage and cannot be directly applied to open real-time systems.

The Bandwidth Inheritance (BWI) protocol [9], on the other hand, extends CBS to work in the presence of shared resources without requiring any prior knowledge about the tasks' structure and temporal behaviour by adopting the Priority Inheritance Protocol (PIP) [18] to handle task blocking. Although the PIP was initially thought in the context of fixed priority scheduling, 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 to the duration of at most $min(n, m)$ critical sections where n is the number of jobs with lower priority than j and m the number of different semaphores used by j.

However, the main drawback of BWI is its unfairness when distributing the original bandwidth reservations, which can have a huge negative impact in the overall system's performance. A blocking task can use most (or all) of the reserved capacity of one or more blocked tasks, without any later compensation of 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. This behaviour potentially severely delays blocked tasks with earlier deadlines, which may 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 [14, 10].

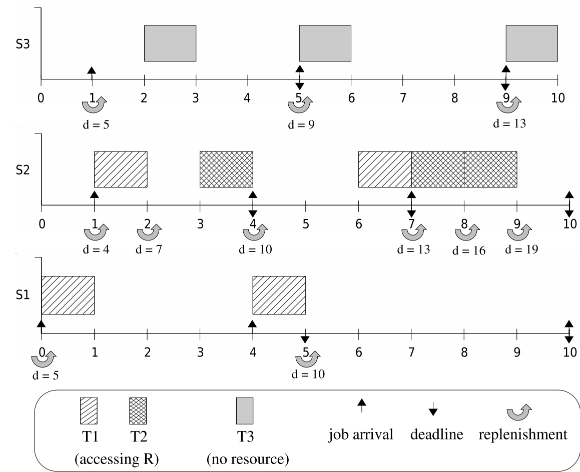

Figure 1. Sharing resources with BWI

Figure 1 illustrates these problems with a simple example. Three servers $S_1 = (2, 5)$, $S_2 = (1, 3)$, and $S_3 = (1, 5)$, characterised by reserved capacity and period respectively, serve three tasks with execution times equal to their respective servers' reserved capacity. 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 execution 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 issue, BWE [20] and CFA [17] integrate bandwidth compensation mechanisms into BWI, try-

ing to fairly compensate blocked servers in exactly the same amount of capacity that was consumed by the blocking task while executing in the blocked server. The approach used in BWE is to only exchange capacities between tasks in the same resource sharing group. When a task uses other servers' capacities, it will later try to compensate blocked servers in the exactly same amounts. To achieve this, BWE stores a global $n * n$ matrix (n is the number of servers in the system) to record the amount of budget that should be exchanged between servers, a budget list at each server to keep track of available budgets, and dynamically manages resource groups at each blocking and releasing of a shared resource. CFA's approach follows a similar goal and 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 or can be cleared at times called singularities, defined as instants in the evolution of the system in which the last pending job is executed.

While both approaches will generally improve the performance of BWI, their increased computational complexity and the fact that CSS tends to fairly distribute residual capacities in the long run [14], lead us to propose an efficient capacity exchange protocol that merges the benefits of a smart greedy capacity reclaiming and stealing policy with the concepts of bandwidth inheritance and hard reservations, allowing reserved capacities to be exchanged more intelligently and with a lower overhead.

4.1 The Capacity Exchange Protocol

Each server maintains only one list of served tasks ordered by tasks' deadlines. Initially, each server has only its dedicated task in its task list and, as long as no task is blocked, servers behave as in the original CSS scheduler. With bandwidth inheritance, a task can be added to more than one task list and be executed on more than its dedicated server, using the following rules:

- **Rule E:** When a high priority task τ_i is blocked by a lower priority task τ_j when accessing a resource R, τ_j is inherited by server S_i. The execution time of τ_j is now accounted to the server, currently pointed 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:** If a blocking task τ_j is inherited by a blocked server S_i, delaying the execution of 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:** If at time t, no active server with pending jobs can continue to execute using some of the rules B, C, or D, and there is at least one active server S_r with residual capacity greater than zero, 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 can be used to execute $j_{p,k}$ through bandwidth inheritance.

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

Rule F 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, that may now be different from S_i. Note that capacity exchange due to blocking is performed without the goal of a fair compensation, reducing the complexity and overhead of CXP.

The hard reservation approach has the advantage of a more constant rate in tasks' execution. However, in general, it 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 this drawback 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.

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 queue of active servers 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, executing it consuming its own residual capacity. Since a server always starts to consume the earliest residual capacity available, no overhead is introduced 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. It has been proved

that residual capacity tends to be reclaimed in a fair manner among needed servers across the time line [11, 14].

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 by efficiently exchanging capacities between hard reservation servers. Figure 2 illustrates CXP's behaviour when scheduling the same set of tasks used to analyse the BWI's drawbacks in Figure 1.

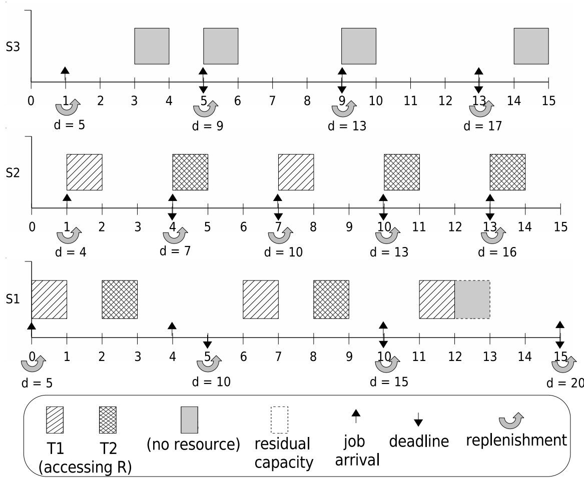

Figure 2. Sharing resources with CXP

At time $t = 1$, task τ_2 is added to server S_1's task list (Rule F). At time $t = 2$, task τ_2 is unblocked and 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 despite the earlier arrival of task τ_1's second job at time $t = 4$, S_1's deadline is not set to $d_{1,2} = 9$ and the job is only released at time $t = 5$, implementing a hard reservation (see Section 3 for details). Also 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 $t = 12$.

5 Handling precedence constraints

Additional constraints on real-time systems arise when the execution of the data's producer must precede the execution of the consumer of that data. Such precedence constraints may affect the system's schedulability, and in more complex scenarios, both shared resources and precedence constraints can be present among tasks.

It is well known that precedence constraints can be guaranteed in real-time scheduling by priority assignment since, in 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.

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. [7] show that the consistency of release times and deadlines can be used to integrate precedence constraints in the task model. Spuri and Stankovic [19] 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, proving that with deadline modification and some type of inheritance it is possible to integrate precedence constraints and shared resources. Mangeruca et al. [12] 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 at admission time. This immediately raises two questions: (i) what happens if a precedent task requires more capacity than declared? (ii) how can a task know if 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 instance of τ_i is selected to execute, it is blocking its successor.

Given a partial order $\prec$ on the tasks, 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 some residual capacity originated by an earlier completion of its previous job.

- **Rule H:** If a precedent server S_j is active at time $s_{i,k}$, S_i checks the current value of S_j's residual capacity. If its equal to zero, then τ_j has not yet been completed and must be added to S_i's task list.

Note that a server that is scheduled for execution already checks the current state of the residual capacity of earlier deadline servers as it tries to consume them before its own reserved capacity which allows us to handle precedence constraints as an access to a shared resource without requiring any previous knowledge about tasks' exact computations times.

Figure 3 shows a possible scheduling of three servers $S_1 = (2, 8)$, $S_2 = (4, 10)$, and $S_3 = (3, 15)$ used to serve three tasks, based on their estimated average execution times and periods, that exhibit the precedence constraints $\tau_1 \prec \tau_2 \prec \tau_3$.

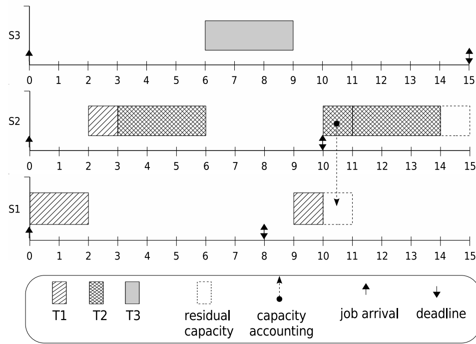

Figure 3. Precedence constraints with CXP

Time $t = 3$ illustrates the situation where the successor server knows it has to complete its predecessor's task. Since S_1 is still active and its residual capacity is set to zero, task τ_1 has not yet been completed and must continue to be executed in S_2 prior to τ_2's execution. On the other hand, at time $t = 6$ and $t = 10$, servers 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 and S_2 is not able to supply any residual capacity to other servers. At time $t = 10$, however, 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.

6 Evaluation

Extensive simulations were conducted to evaluate CXP's flexible management of the original reserved capacities in the presence of shared resources and precedence constraints in dynamic open systems. Multiple and independent runs with initial conditions and parameters but different seeds for the random values were used to drive the simulations [15], using a discrete uniform distribution.

The first 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.

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. Each job consumed the totality of its dedicated server's capacity accessing the shared resource R, with a new job being released immediately after a task has completed its current job.

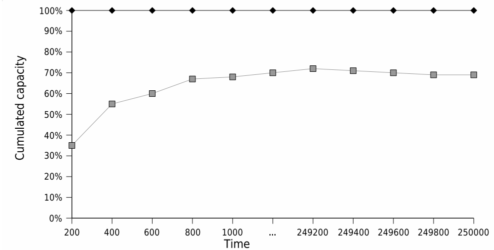

Figure 4. Capacity consumed by the SP task

Each simulation 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 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 4 and 5, respectively.

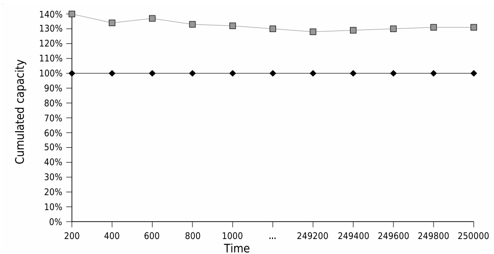

Figure 5. Capacity consumed by the LP task

Sustaining the conclusions drawn from the examples in Section 4, the results show that BWI is affected by the absence of a compensation mechanism. In contrast, CXP's efficient capacity exchange mechanism ensures that both tasks are able to get their allocated capacities even when accessing a shared resource.

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. By focusing on the mean tardiness of tasks, the study evaluated the algorithms' ability to effectively reclaim available capacities to compensate blocked tasks.

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. 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, 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 performance comparison against the other algorithms only isolated servers were used in CXP, disabling its ability to steal non-isolated capacities on overloads. The significant improvement on the system's performance achieved by allowing active overloaded servers to steal inactive non-isolated capacities, particularly in the presence of a large variation in jobs' computation times is detailed in [14].

Figure 6 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. The mean tardiness was determined by $\sum_{i=1}^{n} trd_i/n$, where trd_i is the tardiness of task τ_i, and n the number of evaluated tasks.

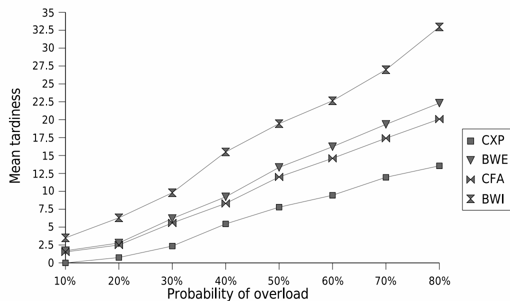

Figure 6. Performance in dynamic scenarios

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. BWE and CFA outperform BWI and achieve a similar performance when scheduling tasks with variable execution times since both algorithms are unable to reclaim residual capacities originated by early completions and exchange capacities only within resource groups, 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, contributing for future jobs of that task to miss their deadlines by larger amounts. On the other hand, CXP effectively improves system's performance, outperforming the currently available solutions. Its greedy capacity reclaiming policy does not restrict capacity exchanging within resource groups and takes advantage of early completions to advance tasks' execution, and the use of hard reservations in conjunction with bandwidth inheritance maximises the amount of capacity that can be exchanged among servers without postponing deadlines.

A third study evaluated the complexity of the different approaches followed in BWE, CFA, and CXP by measuring their overhead in terms of the needed time and memory consumption to schedule the randomly generated tasks sets of the second study, using the base BWI protocol as a reference. Although the three algorithms need almost the same time as BWI to schedule each task set they substantially differ in terms of data storage demands to manage the capacity exchange process. BWE's memory demands grow polynomially with the number of servers and shared resources. Recall that 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, which required an average of 38.2% more memory than BWI. 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, which demanded an average of 17.8% more memory than BWI. On the other hand, CXP focuses on exchanging reserved capacities as early, and not necessarily as fairly, as possible. As such, it does not account the amount of borrowed capacity on each server neither manages individual resource groups. Such policy only demanded an average of 5.7% more memory than BWI, as the number of tasks in each server's task list can be higher in CXP when applying Rules F and G.

The fourth 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. Achieved results allow us to conclude

that CXP is able to handle precedence constraints among tasks whose exact behaviour is not known beforehand without any noticeable overhead.

7 Conclusions

To schedule tasks that share access to some of the system's resources and exhibit precedence constraints, without a complete and previous knowledge of their behaviour is a very challenging problem. This paper addressed both types of constraints and proposed the Capacity Exchange Protocol (CXP), a new strategy that integrates the concept of bandwidth inheritance with the efficient greedy capacity sharing and stealing policy of CSS to minimise the degree of deviation from the ideal system's behaviour caused by inter-application blocking.

CXP focus on greedily exchanging extra capacities as early, and not necessarily as fairly, as possible, achieving a better system's performance when compared against other solutions, has a lower overhead, and introduces a novel approach to integrate precedence constraints into the task model.

Acknowledgements

This work was supported by FCT through the CISTER Research Unit (FCT UI 608) and the Reflect and CooperatES projects (POSC/EIA/60797/2004;PTDC/EIA/71624/2006), and by the European Commission through the ARTIST2 NoE (IST-2001-34820).

References

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

[2] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 191–200, Lake Buena Vista, Florida, USA, December 1990.

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

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

[5] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 161–170, London, UK, December 2001.

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

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

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

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

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

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

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

[13] L. Nogueira and L. M. Pinho. The capacity exchange protocol. Technical Report HURRAY-TR-071101, Available at http://hurray.isep.ipp.pt/, November 2007.

[14] L. Nogueira and 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*, page 153, Long Beach,CA,USA, March 2007.

[15] N. Pereira, E. Tovar, B. Batista, L. M. Pinho, and I. Broster. A few what-ifs on using statistical analysis of stochastic simulation runs to extract timeliness properties. In *Proceedings of the 1st International Workshop on Probabilistic Analysis Techniques for Real-Time Embedded Systems*, Pisa, Italy, September 2004.

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

[17] R. Santos, G. Lipari, and 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*, pages 114–129, Gothenburg, Sweden, August 2004.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronisation. *IEEE Transaction on Computers*, 39(9):1175–1185, 1990.

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

[20] S. Wang, K.-J. Lin, and 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*, pages 158–165, New-port Beach,CA,USA, December 2002.