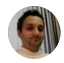
Real-Time scheduling: from hard to soft real-time systems

Article · December 2015

Source: arXiv

2 authors:

Giuseppe Lipari
Université de Lille



Luigi Palopoli
Università degli Studi di Trento

Real-Time scheduling: from hard to soft real-time systems

Giuseppe Lipari
Université de Lille 1
giuseppe.lipari@univ-lille1.fr

Luigi Palopoli
Università di Trento
paolopoli@unitn.it

December 9, 2015

Abstract

Real-time systems are traditionally classified into hard real-time and soft real-time: in the first category we have safety critical real-time systems where missing a deadline can have catastrophic consequences, whereas in the second class we find systems for which we need to optimise the Quality of service provided to the user.

However, the frontier between these two classes is thinner than one may think, and many systems that were considered as hard real-time in the past should now be reconsidered under a different light.

In this paper we shall first recall the fundamental notion of time-predictability and criticality, in order to understand where the real-time deadlines that we use in our theoretical models come from. We shall then introduce the model of a soft real-time system and present one popular method for scheduling hard and soft real-time tasks, the resource reservation framework.

Finally, we shall show how resource reservation techniques can be successfully applied to the design of classical control systems, thus adding robustness to the system and increasing resource utilisation and performance.

1 Introduction

Real-time systems are computational systems whose correctness depends not only on the correctness of the produced results, but also on the time at which they are produced. They must interact with their external environment in a timely manner: for example, real-time systems that control physical plants (digital control systems, or the so-called *cyber-physical systems*) must perform their computation and produce their results at the typical timing rates of the physical environment, and within a bounded delay.

Consider a brake-by-wire control system in a modern car. It is surely a real-time system, as it must command the brake with a maximum delay from when the driver presses the pedal, otherwise a dangerous accident may be produced. As last example, consider a multimedia player in an embedded system that must reproduce a movie in a predefined regular periodic rate to provide a high *quality of service* to its users.

It is clear from these examples that there are many types of timing constraints to be considered, depending on the definition of *correctness*, and the robustness of the system to a violation of such timing constraints.

1.1 Hard real-time

Traditionally, real-time researchers have focused on the notion of *hard real-time system*. An hard real-time system is often modelled as a set of computational tasks to be executed concurrently on the selected hardware platform by a *real-time scheduler*. Computational tasks are characterised by a Worst-Case Execution Time (WCET). They are recurrently activated by input stimuli (i.e. external interrupts, internal events or periodic timers) with a certain *activation-pattern*. Furthermore, they are assigned a *relative deadline*, that is, a maximum delay in completing the computation from when an input stimulus activated the task.

The design of an hard real-time system is subject to the constraint that all instances of all tasks must complete their computation within their assigned relative deadline. The system is modelled with a mathematical formalism, and we can answer several interesting questions:

- **Feasibility**: does there exist a schedule in which all deadlines are respected?
- **Schedulability**: given a specific scheduler, is it able to generate on-line a schedule such that all deadlines are respected?
- **Parameters assignment**: find the *optimal* assignment of scheduling parameters to tasks so that all deadlines are respected.

Several variations of the same problem have been investigated in the real-time system literature, by varying the task model (e.g. multi-frame tasks [6], generalised recurring branching tasks [7], DAG tasks [41], etc.), the platform model (single or multiprocessor, homogeneous or heterogeneous), the interaction model (independent tasks, precedence constraints [13], shared resources [40], etc.).

However, a legitimate question is: what happens if a deadline is missed?

1.2 Missing deadlines

There are several possible outcomes of a deadline miss. Let us fist consider that case in which a deadline miss causes a software bug. For example, it may happen that the program goes into an inconsistent state (some data structure contain inconsistent data, or pointers contain the wrong address in memory, etc.). In this case, the software may start producing inconsistent outputs or even crash.

Basic software development guidelines tell us that this should never happen. Even if we had previously performed a careful analysis of the code and a precise schedulability analysis which "guarantees" that no deadline miss will occur, there is always the possibility that our mathematical model does not capture precisely all possible effects, or that an unexpected hardware fault compromises the ability of a task to complete before its deadline. Therefore, even for hard real-time systems that have been deemed schedulable by off-line schedulability analyses, it is a good design norm to make sure that the software remains into a consistent state in case of a deadline miss so that it can continue its execution in degraded mode. For example, if the output of Task A is put into a buffer that is later read by another Task B, we need to make sure that the buffer content is always valid, no matter if the Task A completes before its deadline or not.

From now on, we will assume that the system is developed according to good design norms, and that the software data structures remain consistent under all operating conditions.

Having ruled out software bugs, what may happen when a deadline is missed? the answer to this question depends on 1) the application requirements, 2) the software architecture and how a deadline miss is dealt with.

As a first example, consider a digital control system consisting of an embedded board with a real-time operating system executing a set of real-time control tasks. Suppose that Task A writes the results of its computation in a buffer that is later used by the actuators to command the physical system. If A is late, the buffer will hold the previous value of the command at the time of the actuation, so the actuator will use the old data to command the plant. The impact of this on the dynamics of the physical system depends on the control law, the dynamic evolution of the physical system, the sampling period, etc. We can estimate that, if the deadline miss happens too often or too many times consecutively, the control system may become unstable to the point that we cannot control it anymore. This can lead to serious consequences in critical systems like cars and planes. All these consideration pertain to the theory of dynamic control, so we can answer our original question only after analysing the overall system, including the control dynamic.

Consider now the example of a multimedia player, where a set of tasks is in charge of playing a movie by decoding the flow of data frames into video and audio data to be synchronised and set to the appropriate output video and sound peripherals. Typically, at 40 video frames per second, one video frame must be decoded and shown on the screen every 25 milliseconds. What if the decoding task misses its deadline? If the system is well designed, we will experience some strange artifacts on the screen (e.g. large blocks of pixels, low resolutions areas, etc.) which will lower our appreciation of the show. If this happens too often, the experience may become very unpleasant. The exact characterisation of the *Quality of Service* perceived by the user depends, once again, on the requirements of the application and on the software/hardware architecture.

However, it is clear that in many cases a deadline can be missed without causing catastrophic consequences, but just with a simple degradation of the performance of the system. Furthermore, as we will see in Section 5, relaxing the hard real-time constraint (and hence allowing some deadline misses) may sometimes *improve* the performance of a control system.

One may ask why real-time research literature is so much focused on hard real-time systems. The main reason is *separation of concerns*: by imposing hard real-time constraints, we can assess the "correctness" of the system by just analysing few, application-independent parameters (like WCETs, minimum inter-arrival times of events, etc.) without the need to take into account the final application requirements.

Hence, by translating the application requirements into hard real-time parameters and constraints (periods and deadlines) we can easily derive general laws about scheduling. Of course, something similar can be done using a soft real-time task model in which we constraints the number or the extent of a deadline miss.

Organisation of this paper The paper is organised as follows. In Section 2 we present the model of a real-time task and the problem of large variations in execution times. In Section 3 we briefly present the Resource Reservation framework and the Constant Bandwidth Server algorithm. We also discuss a model of soft real-time tasks and the typical requirements. In Section 4 we give a quick overview of the existing techniques and tools for soft real-time scheduling and analysis. In Section 5 we present a technique for designing robust and efficient control systems based on resource reservation techniques for coping with uncertainty in execution times of control tasks. Finally, in Section 6 we present our conclusions.

2 System model

2.1 Hard real-time tasks

In this paper we will focus on the classical model of periodic and sporadic real-time tasks. However, many of the techniques presented later are valid for more complex task models.

A real-time tasks τ_i is characterised by a tuple $\tau_i = (C_i, D_i, T_i)$, where C_i is the *worst-case execution time*, D_i is the *relative deadline*, and T_i is the *minimum interarrival time*. A task produces a (finite or infinite) sequence of jobs $J_{i,0}, J_{i,1}, \ldots$, and each job $J_{i,j}$ is characterised by an absolute activation time $a_{i,j}$, a computation time $c_{i,j}$, and an absolute deadline $d_{i,j}$. For a periodic task T_i represents the distance between two consecutive activations, hence it must hold that $\forall j \geq 0, a_{i,j+1} = a_{i,j} + T_i$, whereas for a *sporadic task* T_i represents the *minimum interarrival time*, so it must hold that $a_{i,j+1} \geq a_{i,j} + T_i$. The absolute deadline is computed as $d_{i,j} = a_{i,j} + D_i$. By the definition of WCET, it must hold that $\forall j, c_{i,j} \leq C_i$.

A real-time tasks models a *recurrent thread* in a real-time operating system. Using the pthread library in Linux, the typical structure of the code for a periodic task is represented in Listing 1.

Listing 1: Structure of a periodic thread in Linux

```
struct per_data {
  int index;
  long period_us;
  long dline_us;
};

void *thread_code(void *arg) {
  struct per_data *ps = (struct per_data *) arg;

  // Initialization

  struct timespec next, dline, now;
  clock_gettime(CLOCK_REALTIME, &next);
  while (1) {

    // Job execution

    // Check deadline miss
    clock_gettime(CLOCK_REALTIME, &now);
    dline = next;
    timespec_add_us(&dline, ps->dline_us);
    if (timespec_cmp(&now, &dline) > 0)
      pthread_exit(0);

    // Wait until next period
    timespec_add_us(&next, ps->period_us);
    clock_nanosleep(CLOCK_REALTIME,
                    TIMER_ABSTIME, &next, NULL);
  }
  return NULL;
}
```

The struct per_data contains the period and the deadline expressed in microseconds. After initialization the thread enters a while loop where it executes the code of the jobs. At the end of the execution, we check if the deadline has been missed, in which case we abort the thread. Otherwise the thread suspends itself waiting for the next periodic activation (at time next).

In this example, we chose to abort the thread when a deadline is missed. Please note that we detect the deadline miss *after* the job has already completed its execution, beyond its deadline. This means that other lower priority threads may have been delayed by the overrun of this thread, and this may cause future deadline misses of those other threads. Therefore, the simple strategy shown in Listing 1 may or may not be the appropriate.

2.2 Scheduling

We consider an hardware platform system consisting m identical processors on which we execute the RTOS and the tasks. The task execution is controlled by a *scheduler* that decides at each instant which tasks can be executed on the processors. Scheduling algorithms can be classified into *partitioned schedulers* and *global schedulers*. In the first case, tasks are statically allocated to processors, and on each processor a scheduling algorithm decided which of the tasks pertaining on that processor must be executed. In global scheduling, every task can execute on any processor. Typically, ready tasks are inserted into a single *ready queue* and the scheduling algorithm selects which task to execute at each instant.

The most popular real-time scheduling algorithm is by far Fixed Priority (FP), because it is implemented in every OS from nano-kernels to large General Purpose OS like Windows and Linux. In FP, tasks are assigned fixed priorities and every job of that task has the same priority. The Earliest Deadline First (EDF) scheduler sorts tasks in the ready queue according to their current absolute deadlines, and executes the ones with the earliest deadline (hence the most "urgent"). EDF is less widespread but it is gaining popularity especially for soft-real-time systems. An implementation of the EDF with the Constant Bandwidth Server (see Section 3) is now available in Linux starting from version 3.14 [26].

A *schedulability test* is an algorithm that, given a set of tasks and a scheduling algorithm, tells us if during execution every task will respect its deadlines. For hard real-time systems, it is of paramount importance to run the test before deploying the software, to guarantee that every deadline will always be met. A schedulability test for soft real-time systems can give useful indications about the typical soft-real-time constraints as the number of deadline misses, the maximum tardiness (that is, the maximum extent of execution after the deadline has expired), etc.

2.3 Execution times

Executions times of tasks may vary. The variation depends on many different factors: the algorithm implemented by the task, the input data, the hardware architecture. In particular, the presence of caches, the instruction pipeline, out-of-order execution of instructions, etc. With the recent advances in processors and in multicore systems, the impact of the architecture on the variation of execution time has become larger and larger, to the point that the worst-case execution time of a task may be orders of magnitude larger than its average execution time, but it may happen very rarely. In this case, a schedulability analysis based on WCETs leads to a very low utilisation of resources.

Let see an example of what can happen when a task executes more than expected. Consider a system with 3 tasks, $\tau_1 = (1, 4)$, $\tau_2 = (2, 5)$, $\tau_3 = (2, 6)$, scheduled by EDF on a single processor system. A simple utilization test for EDF consists on computing the overall load of the system as $U = \sum_{i=1}^{3} \frac{C_i}{T_i}$. If all tasks have relative deadline equal to the period, and they are all independent of each other, all tasks are schedulable if and only if $U \leq 1$. In the example, $U = 0,983$, so the system is schedulable. However, due to a wrong estimations, it may happen that the computation time of τ_1 sometimes raises to 2. In this case, the deadline of *any* instance could be missed.

In Figure 1 we show the resulting EDF schedule when the first 3 instances of τ_1 execute for 2 units, instead of 1. Each line represents the execution of one

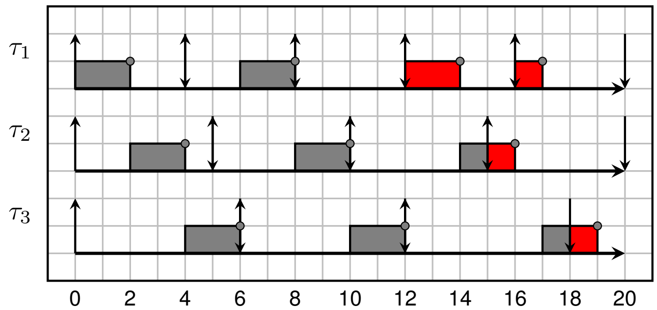


Figure 1: Example of overlaod for EDF.

task. The up arrows denote activation times, while downward arrows represents deadlines (here deadlines are equal to periods, so all absolute deadlines coincide with the next arrival time). The execution of every task is represented by a grey rectangle if it is performed within the deadline, and by a red rectangle if performed after the deadline. The completion of an instance is denoted by a small black point in the upper right corner of the rectangle. As you can see, all three tasks miss their deadlines.

To mitigate the problem, we could try to abort the third instance of task τ_1, but this does not guarantee that the other instances will meet their deadlines, unfortunately: for example, the fourth instance of the first task will still miss its deadline even when aborting its third instance.

Using FP instead of EDF, only lower priority tasks will miss their deadlines, therefore we achieve a sort of elementary temporal isolation: high priority tasks are not influenced by lower priority tasks. However, priority is not always related to criticality: for example, to optimise resource utilisation the designer may choose to set priorities in Rate Monotonic order (higher priority to tasks with shorter periods) regardless of their criticality.

This problem has been addressed in many different ways. Recently, the Mixed Criticality methodology [42] has been proposed: an an application consists of multiple criticality levels, and different levels of assurance are provided to each level. Typically, high criticality tasks are assigned an execution budget, and when this budget is exceeded, the system goes into *high criticality mode* where all low criticality tasks are discarded. This permits to guarantee the execution of higher criticality tasks under all conditions, however it does not allow us to control what happens to low criticality tasks.

Another approach is to isolate the temporal behaviour of each task by using the Resource Reservation approach, which we describe in detail in Section 3.

3 Resource reservations

In the Resource Reservation framework [38], every task is assigned to a *scheduling server*, which is characterised by a maximum *execution budget* Q and a

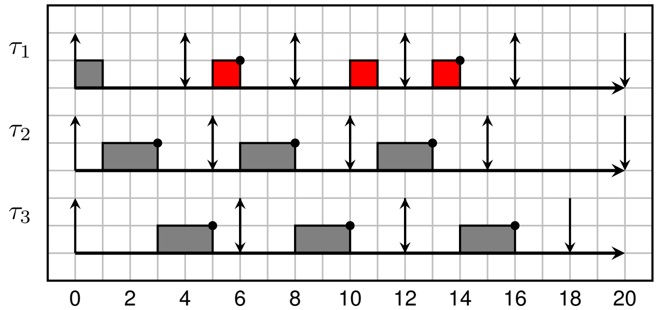


Figure 2: Same example scheduled by the CBS.

period P. The idea is that, if a task executes more than expected, it will suffer but it will not negatively influence the execution of the other tasks. One popular resource reservation algorithm is the Constant Bandwidth Server [1] which integrates with the EDF scheduler. Basically, the algorithm rules say that the execution budget is recharged at the beginning of each period, and it is decreased when the task executes. If the execution budget reaches 0 before the task has completed its execution, the task deadline is postponed by P units of time, so its dynamic priority is lowered and the budget is recharged to Q. In another version (called *hard reservation*), the task whose budget is exhausted is suspended until the end of the period when its budget is finally recharged to Q. For more details on the algorithm, please refer to the original paper. Resource reservation provide the important *temporal isolation* property: the ability of a task to meet its deadlines does not depend on the behaviour of the other tasks, but only on its execution time profile and on the budget and period assigned to it. In addition, the CBS provides the *hard schedulability* property: is we assigned a budget larger than the WCET of the task, and a reservation period no larger than the task's period, the task is guaranteed to meet all its deadlines.

The algorithm has been recently implemented in Linux, and starting from version 3.14, it is bundled in the official Linux distribution. A new scheduler is available to developers, (called `SCHED_DEADLINE`) and the superuser can use it to assign budget and periods to any Linux thread.

The same example of Figure 1 is scheduled in Figure 2 using the CBS. Task τ_1 is assigned a server $S_1 = (Q = 1, P = 4)$, task τ_2 is assigned $S_2 = (Q = 2, P = 5)$, and task τ_3 is assigned $S_3 = (Q = 2, P = 6)$. Task τ_1 is thus constrained to execute only 1 unit of execution time every 4, and it will miss many deadlines, whereas the other two tasks are not influenced.

It is important to underline that the period of the server must not be necessarily equal to the period of the task. In fact, sometimes it may be appropriate to set $P = \frac{T}{k}$, with k positive small integer. If we set k to a large integer number, the execution of the task will be *spread* over its period, and the larger is k the more the execution resemble a *fluid execution*. However a large k implies a large number of context switches and hence a much larger overhead of the

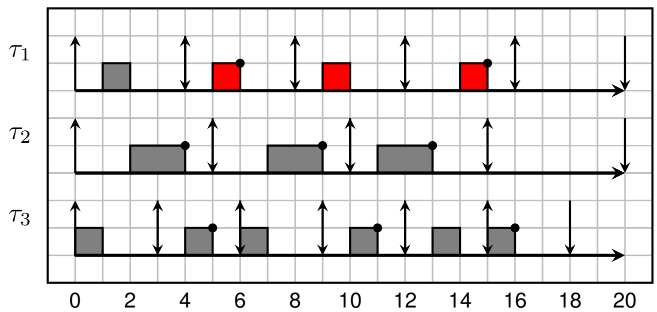


Figure 3: Task τ_3 is assigned a server with half its period.

scheduler. Using a small k is useful for controlling precisely the timing of the task output, as will be explained in Section 5. In Figure 3 we show the same example as above, where task τ_3 is assigned a server with $S_3 = (Q = 1, P = 3)$.

3.1 Soft real-time tasks

The model for soft real-time tasks is very similar to the model for hard real-time tasks. There are two main differences: 1) what to do when a deadline is missed and 2) the notion of *correctness*, that is what are the requirements of a soft real-time tasks.

We start by defining the *tardiness* of a task as the maximum delay in completing an instance after its deadline. If we denote by $R_{i,j}$ the response time of the j-th instance of task τ_i, its tardiness is defined as $\max_j(0, R_{i,j} - D_i)$. Of course, hard real-time tasks must have tardiness equal to 0.

There are many ways to handle deadline misses and overruns of soft real-time tasks, here we summarise the major ones.

- **Deadline miss detection**: the program detects that a deadline has been missed, and takes the appropriate action. We can detect the occurrence of a deadline miss at the end of the job (as in Listing 1) or at the time of the deadline miss: in any case, we are already executing after the deadline, so we have to take some recovery action to remove the overload situation. Notice that, if there is no temporal isolation, other tasks may miss their deadlines in cascade. In case of resource reservations, instead, only the failing task will be concerned.

- **Budget overrun detection**: in this case, the system informs us when the task is exceeding its execution budget. We can then suspend the task and let it continue later on (as in the resource reservation framework); or abort the task instance, so that it can start clean again in next period.

It is important to underline that the second technique (budget overrun detection) is widely used also in safety critical hard real-time systems for fault-tolerant reasons.

What to do when a deadline miss or a budget overrun is detected? again, there are many possibilities:

- We can **abort** the executing instance and use some standard value of the task output. This can be done to alleviate the overload of the system and while providing an output to the actuators (or to the following tasks in the chain). However, we must be careful in aborting an executing thread: the thread must release all mutually exclusive semaphores before aborting, and it must leave all data structures in a consistent state. This is not always easy to achieve.
- We can **continue** the task execution and finish late. Sometimes a late result is better than no result, especially if we can prove that the overall tardiness is limited. Also, it is possible that the overload will decrease by itself is the system is not permanently overloaded.
- As an alternative, we can decide to **skip** future instances, so to decrease the overload and keep the maximum tardiness in check. Also in this case, we still need to provide standard values for the outputs of the skipped instances.

Which output should be provided for aborted or skipped instances? A typical strategy is to hold the previous value of the output, both in control applications and in multimedia applications. In the first case, we need to model this fact in the design of the controller. In the second case, for video decoders the image on the screen may freeze for an instant, and hopefully return to normal operation in the following frames; for audio applications this can result in a noise, so it is sometimes better to send a forecast signal or an empty signal.

In Listing 2 we show an example of how to combine these techniques by using SCHED_DEADLINE for budget control, and by skipping all late instances if a deadline has been missed (lines 18-20).

Listing 2: Structure of a periodic thread in Linux

```
1  void *thread_code(void *arg) {
2    struct per_data *ps = (struct per_data *) arg;
3
4    struct timespec next, now;
5    clock_gettime(CLOCK_REALTIME, &next);
6    while (1) {
7     // Wait until next period
8     timespec_add_us(&next, ps->period_us);
9     clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
10                    &next, NULL);
11
12     // Job execution
13
14     // Check deadline miss
15     clock_gettime(CLOCK_REALTIME, &now);
16     while (timespec_cmp(&now, &next) > 0) {
17       // Skip late instances
```

```
      timespec_add_us(&next, ps->dline_us);
    }
  }
  return NULL;
}
```

3.2 Soft real-time requirements

Even if deadlines can sometimes be missed, we need to do an off-line analysis of the system to estimate the number and frequency of missed deadlines. We also need on-line techniques to keep these misses in check. Typically, we have three types of requirements:

- Bound on the **number of deadline misses in an interval of time**. We express this constraint as $\binom{m}{n}$, meaning that we can miss at most m deadline over n instances. This is sometimes used in the design of control systems to ensure stability of the system. For example, we may require that, over 10 instances, no more than 2 will miss their deadlines as $\binom{2}{10}$. We can also impose several of these constraints: for example, we want at most 2 deadline misses but never consecutively. In this case, we express the constraint as $\binom{2}{10} \wedge \binom{1}{2}$.
- Bound on the **tardiness** of a task. Task's instance can complete after their deadlines, but the delay must be bounded. Of course, this makes sense only if we decide to not abort the instance in case of a deadline miss.
- **Probabilistic bounds**: the same requirements can be expressed in terms of probability. For example, we look for upper bounds on the probability of having a deadline miss. Clearly, probabilistic analysis needs a probabilistic characterisation of the execution time of a task.

4 Algorithms for soft real-time tasks

The resource reservation framework provide the *temporal isolation* property which is useful to analyse each soft real-time task in isolation. This simplifies the analysis and the algorithms for dealing with variations in execution times. Many analysis algorithms and scheduling methods have been proposed in the literature for dealing with soft real-time tasks. In the following sections we review some of the most relevant papers on analysis of soft real-time tasks in the context of resource reservations.

4.1 Probabilistic analysis

If we want to analyse a soft real-time task with large variations in execution times, one approach is to reason in terms of probability. An initial proposal in this regard was made by Abeni and Buttazzo [2]. They proposed a methodology for computing the probability of finishing time of a soft real-time task served by a CBS with given budget and period, starting from the probability distribution of the execution times. Recent works by Palopoli et al. [34, 32, 3] have refined the initial analysis providing analytical bounds on the deadline miss probability.

4.2 Reclaiming

Typically, a soft real-time task is assigned a server with budget not inferior to its average execution time. This means that, when the task needs to execute more than its assigned budget, it will probably miss its deadline. However, when the task needs to execute for less than its budget, the remaining part is discarded. Therefore, it is interesting to see if it is possible to redistribute the unused budget to the other more needing tasks, so to reduce the overall probability of deadline miss.

Many algorithms have been proposed in the literature for implementing budget reclaiming mechanisms. In this paper we will limit ourselves to the methods that extend the CBS server.

The CASH (Capacity Sharing) algorithm [11] recuperates the unused budgets in a queue. The queue contains pairs (q_r, d_r), where q_r is the remaining budget, and d_r is the deadline of the corresponding server. The unused budget can be donate to a server with scheduling deadline not earlier to d_r. This method works well for periodic tasks but it is difficult to extend to sporadic tasks. The CASH algorithm has later been extended to multiprocessor systems [37].

The GRUB (Greedy Reclamation of Unused Bandwidth) algorithm [28] uses an analogy with a fluid scheduler to keep track of the *active bandwidth* of the system U_{act}. The remaining bandwidth $1 - U_{act}$ is donated to the server currently executing. It works for any kind of task (periodic, sporadic and even aperiodic). A variant called ShRUB (Shared Reclamation of Unused Bandwidth) [36] distributes the spare bandwidth across all active servers. GRUB has been extended to power-aware scheduling [39] since reclaiming and power-aware scheduling are two faces of the same problem. The GRUB algorithm and its variants have been designed for single processor systems.

Other relevant works in this area include [27, 31, 30].

4.3 Adaptive reservations

When using resource reservations, one difficult problem to solve is how to assign the server's budget. A too small budget leads to many deadline misses, whereas a too large budget wastes precious resources. In addition, tasks may exhibit structural variations in execution times. For example, a task may run one of two different algorithms, depending on the *operational mode* of the system, and these two algorithms may exhibit completely different execution time probability distributions.

Therefore, one interesting approach is to *adapt* the budget to the needs of the task at on-line. This can be done by using a feedback control scheme: while the task executes, the *scheduling error* (that is the tardiness, or the number of deadline misses) is measured, and a feedback control algorithm adjusts the budget in order to reduce the scheduling error to zero.

Adaptive reservations have been proposed in [4] and later in [5, 15, 35] as a mean to support feedback based adaptive budget control. In particular, implementations have been proposed for the Linux OS in [17, 35] for dealing with multimedia applications as video players. If a model of the application is known (for example, a model of the MPEG decoder), it is possible to give guarantees about the stability of the controller and on the QoS. A complete QoS architecture has been proposed for single processors and for distributed systems [18] and for industrial automation applications [14].

For multimedia applications that have not been designed according to the soft real-time paradigm (but they still exhibit real-time requirements), it is possible to automatically detect the best "period" of the reservation server, using simple filters inside the OS. This idea has been implemented in a mechanism for the Linux kernel [16].

4.4 Shared resources

If a task can block due to a locked semaphore on a shared resource, its remaining budget is not valid anymore. Moreover, if a task that holds as shared resource is suspended because its budget has been exhausted, blocked may suffer a form of priority inversion that can seriously compromise schedulability. Therefore, it is necessary to modify the reservation algorithm and the synchronisation protocol to take into account these interactions.

There are three main approaches to do this.

- In the first, we maintain the reservation algorithm unchanged but we check if there is enough budget left before entering a critical section: if the length of the critical section is larger than the remaining budget, the task is suspended. In this way we avoid the second problem. Blocking times are then calculated using the synchronisation protocol (e.g. Stack Resource Policy or Priority Inheritance) and taken into account into the schedulability test. This is the approach followed by the BROE [9] algorithm.

- Another approach is to avoid suspending the task if is runs inside a critical section. The overrun are accounted for in the analysis together with the blocking times. This is the approach followed by the SIRAP algorithm [8].

- The third approach is to use *bandwidth inheritance*: if a task blocks on a critical section, the holding task can inherit the budget of the blocked task, and use the budget with the shorter deadline. This is the approach of the BWI (Bandwidth Inheritance) Protocol [29] and its multiprocessor extension M-BWI [19]. This last protocol does not require any specific knowledge of the length of the critical sections (such a knowledge is needed only for analysis) and can still guarantee temporal isolation between non interacting tasks.

In our opinion, the latter is the most adequate for soft real-time systems, as the isolation property does not depend on the correct estimation of the length of the critical sections and of the blocking time. Also, when adding a new task/reservation in the system, it is not necessary to recalculate the resource holding times. We are currently working on a comparison of the different approaches in terms of resource utilisation and implementation details.

5 Control systems: hard or soft?

5.1 Generalities on control systems

A feedback control system is given by the interconnection of a plant (a "physical" system to be controlled) and of a digital controller.

The plant is typically described by a differential equation:

$$\begin{aligned}\dot{x}(t) &= f(x(t), u(t), t)\\ y(t) &= g(x(t), u(t)),\end{aligned}$$

$t \in \mathbb{R}$ represents time, $x(t) \in \mathbb{R}^n$ is a vector of state variables, $y(t) \in \mathbb{R}^m$ is a vector of output variables and $u(t) \in \mathbb{R}^p$ is a set of input variables whereby the system evolution can be controlled. For the purposes of this paper, we will restrict our attention to linear and time invariant plants, for which the differential equation above can be specialised as

$$\begin{aligned}\dot{x}(t) &= A_c x(t) + B_c u(t)\\ y(t) &= C_c x(t) + D_c u(t),\end{aligned} \tag{1}$$

where A, B, C and D are matrices of suitable size. Typical tasks of a control system are to steer the system state to a desired equilibrium x_{eq} (point stabilisation) or along a specified trajectory $y_{ref}(t)$ (tracking). Instrumental to this goal is the solution of a particular sub-problem, which is the stabilisation of the origin of the state space (i.e., driving the system state $x(t)$ to 0). This problem is called *stabilisation*, and in this paper will receive a special attention.

The stabilisation problem amounts to finding a command value $u(t)$ such that the resulting *closed loop* system is asymptotically stable meaning that: 1) for all ϵ we can find a δ such that if the initial state $x(0)$ is within a distance δ of the origin, then $x(t)$ will always be within a distance ϵ for all t, 2) $\lim_{t\to\infty} x(t) = 0$. In plain words, if the system evolution starts close to the origin, it will always remain close to the origin, and it will eventually converge to the origin itself.

Different design methods can be found that achieve asymptotic stability and that possibly fulfil additional requirement [24]. Generally speaking, such methods produce a controller, which is itself a dynamic system that receives as input the output $y(t)$ of the plant and produces as output the input $u(t)$ for the plant

$$\begin{aligned} \dot{z}(t) &= E_c z(t) + F_c y(t) \\ u(t) &= G_c z(t) \end{aligned} \tag{2}$$

The vector z is compounded by the state variables of the controller and it can have a different size according to the type of controller chosen. For instance, for LQG controller z has the same size of the state vector x of the plant. The LQG controller has the important property: if the system is acted on by a noise term (i.e., in adddition to u we have another uncontrollable input w that fluctuates stochastically), the controller minimises a cost function like $\int_0^\infty E\{x^T Qx + u^t Rudt\}$, which accounts for the evolution of the state variables and for the control action.

The digital implementation of a controller like this amounts to developing a program that cyclically: 1) takes a sample of the output y of the plant, 2) updates the state of the controller producing an approximation of the solution of the differential Equation 2, 3) computes a command value u that is applied to the plant and held constant until the next sample is ready. The name of the game is to find a technological scheme such that the behaviour of the "ideal" control in Equation 2 is closely approximated and its relevant properties (first and foremost stability) preserved. The classic results on digital control [23] show how to achieve the result under the assumptions that the loop is executed with a fixed periodicity T and that computation time can be neglected. In practical terms, this is done by introducing a discrete-time model that describes the evolution of plant and controller across a sampling period $[kT, (k+1)T]$, under the typical assumption that control values are held constant throughout:

$$\begin{aligned} x((k+1)T) &= Ax(kt) + Bu(kt) \\ y(kT) &= Cx(kT) + Du(kt) \\ \\ z((k+1)T) &= Ez(kT) + Fy(kT) \\ u(kT) &= G_c z(kT) \end{aligned} \tag{3}$$

Computation delays do not pose serious issues as long as they are fixed. In this case the designer can easily account for the delays by extending the model of the plant with additional state variables and compensate for the dynamics of these additional variables in her/his controller.

5.2 Hard Real–Time implementation of controllers on multiprogrammed systems

Delay compensation cannot be done easily if the computation time of the task has a wide variability. Things become even trickier when the processor is shared with other tasks. In this scenario, the task can suffer from scheduling interference introduced by preempting tasks which adds to the variability of the introduced delays. A pragmatic way to solve this problem is by using the so-called time-triggered model of computation [25]. The idea is that sampling and actuations are *forced* to take place at specific points in time, regardless of when each job of the task starts or ends. For instance, suppose that samples are collected periodically with period T. *Exactly* at time kT with $k \in \mathbb{N}$, a new sample is collected [1] and a new job J_k is activated. The job is assigned a relative deadline D to complete its computation. When the job finishes, the result is retained in an intermediate buffer and is delivered to the actuators only at time $kT + D$. The evident advantage of this idea is that jobs are allowed to have a variable response time *but* the sensor-to-actuator delay is always D. The only requirement is that the deadline D is enforced for each and every job and this can be done using the standard methods of hard real-time scheduling theory.

This idea is therefore based on a clear separation of concerns between the control designer (who compensates for fixed delays) and the system designer (who uses real-time scheduling to secure the timely execution of the jobs). An important problem with this "clean" approach comes when the execution time of the control task changes by a significicnt amount on a job-by-job basis. This is the exactly the case for a new generation of control application that rely on data collected from visual sensors, LIDARS and RADARS. The processing time required to extract the meaningful information is heavily dependent on the environment (e.g., extracting relevant features from a clean image is obviously less demanding than from an images cluttered with all sort of artefacts). For applications like these, guaranteeing every single deadline under all possible conditions understandably requires a significant over-provisioning of system resources. On the other hand, several studies [12, 33] reveal that even in control systems a "controlled" fluctuation in the delays introduced by the feedback loop can be tolerated without evident degradations in control performance. What is needed to translate this simple observation into a credible alternative to hard real-time design is a rigorous methodology with certifiable results.

5.3 Making the case for soft real–time

Any methodology for "certifiable" design of real-time embedded controllers rests on three pillars:

1. A clear understanding of the system level properties that need to be guaranteed,

2. A model for the temporal behaviour of the computing platform that can be treated in control design,

3. A platform design based on a model of computation and a scheduling solution that allows to control the temporal behaviour of the tasks for an assigned choice of design parameters.

For classic "hard real-time" methods the system level property that needs to be preserved is usually related to system asymptotic stability, the temporal

[1]This can be done with an appropriate hardware which ensures a small sampling jitter.

behaviour of the task is captured by the (T, D) pair, where T is the sampling time and D is the sensor/to actuator delay and the platform design is based on the combination of time-triggered model of computation and fixed priority scheduling. Any alternative paradigm has to propose a credible alternative to each of these three elements.

Generally speaking, if we give up the strict hard real-time execution guarantees, we are implicitly injecting a stochastic component related to the behaviour of the controller. The property required to the closed loop system will have a stochastic flavour in its turn. There are different possibilities. The first one is *almost sure stability*, which essentially means that all the realisations but a set of null measure will converge to 0: $\mathrm{Prob}\{\lim_{t\to\infty} |x(t)| = 0\} = 1$. A different possibility is the so-called *second moment stability*, meaning that the variance of the state will converge to 0: $\lim_{t\to\infty} E\left\{x^T(t)Qx(t)\right\} = 0$. The intuition is that the state will be stocastically contained in a ball of shrinking size. We will adopt this second notion in the discussion below.

As for the platform design, the use of the CBS scheduler has the remarkable property that a choice of the scheduling parameters is immediately translated into the probability distribution of having different delays in the computation [32, 34], which is utmost useful for control design. For the model of computation, different choices are possible with different results, as discussed next.

5.3.1 A first possibility: job dropout

The simplest possible choice for a model of computation is to adopt a standard Time Triggered model that acquires sensor input exactly at the start of a sampling period (e.g., kT) and releases the output exactly at the end (e.g., $(k + 1)T$) with the addition that if the job executes beyond kT it is outright cancelled and the past input is held constant throughout the next job (from $(k + 1)T$ to $(k + 2)T$). In this case the evolution of the total state $\hat{x} = \begin{smallmatrix} x \\ z \end{smallmatrix}$, comprised of the state of the plant (x) and of the controller z, is given by:

$$\hat{x}((k+1)T) = \begin{cases} A_c\hat{x}(kT) & \text{if the } k^{th} \text{ job finished within } (k+1)T \\ A_o\hat{x}(kT) & \text{if it does not} \end{cases} \tag{4}$$

for an appropriate expression of the A_c and A_o matrices (see [22]).

With the simple Model of Computation proposed here, there is not any "carry-on" execution across to ajacent jobs. Therefore the stochastic process ruling the transition between the two modes (deadline respected and deadline missed) is memoryless. Thanks to the adoption of the Resource Reservation algorithm, the probability μ of dropping a job (with the system evolving according to the "open loop" matrix A_o) can easily be found as $\mathrm{Prob}\left\{\left\lceil \frac{c_j}{Q} \right\rceil R > T\right\rceil$ where Q is the reservation budget, R is the reservation period, T is the task period and c_j is the (stochastic) computation time of the j^{th} job. For a given probability distribution for c_j, it is straightforward to find this probability as a function of Q and R. Further, being the transition between the two modes ruled by a

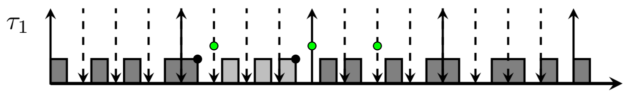


Figure 4: Example schedule for soft real–time control task

memoryless process, the second moment stability is expressed by the relatively simple requirement that the matrix

$$\tilde{A} = (1 - \mu)A_c \otimes A_c + \mu A_o \otimes A_o \tag{5}$$

has all eigenvalues with modulus smaller than 1, where $\otimes$ denotes the Kronecker product.

In summary, by using CBS with a simple model of computation where delayed jobs are simply dropped, it is easy to establish a link between the scheduling parameters and the probability μ of dropping a job, and also to find if the resulting system is second moment stable. The price to pay is that the outright cancellation of a delayed job can be a very crude choice and system stability can require unnecessarily high values of the bandwidth.

5.3.2 A Better Alternative

The problem with the approach above is that the system could possibly benefit more from a fresher control output (although generated with a little delay) than from an abrupt cancellation. On the other hand, the simplification of constraining the possible delays to one (or at least) a few possible values is too important to be easily dismissed.

One possibility before us is to use the combination of a real-time tasking model (where job activations are buffered until the previous jobs finish), with a time-triggered model of computation where the control input is collected with a fixed periodicity (release points) and the output is released only upon specified instants, which is convenient to set equal to the end of the reservation periods. An example is shown in Figure 4. The dashed lines denote the acceptable release points for the output, and the green circle represents the moment a new fresh output is taken. In this example, Job 1 is late. On the expiration of its period, the input for Job 2 is collected, but Job 1 keeps going. When Job 1 finishes its output is released upon the next possible point (green circle) and Job 2 can start. As regards Job 3, we can see that the job is late beyond an acceptable threshold and is cancelled.

As discussed in out previous work [20], it is possible to model the evolution of the delays as a Markov Chain, whose particular structure simplifies the computation of the steady state distributions [32, 34]. The resulting closed loop system is a so called Makrov Jump Linear System (MJLP), and its second moment stability can be analysed with (rather complex) mathematical tools.

To make a long story short, allowing for delays makes loosen the unnecessary restrictions of the Model of Computation where late jobs are cancelled. This can

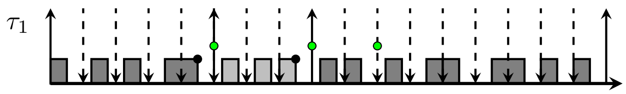


Figure 5: Example schedule the Continuous Stream model

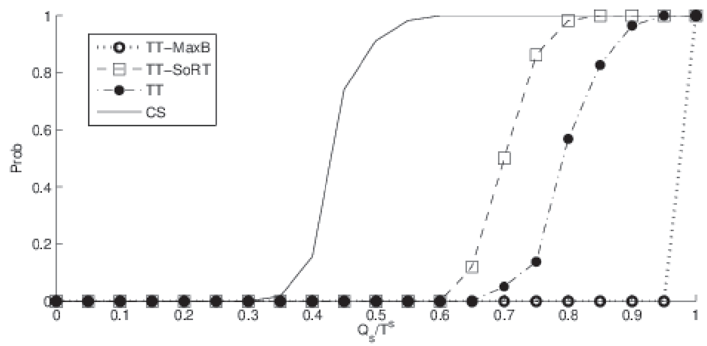


Figure 6: Percentage of systems stabilised with different model of comptuation vs bandwidth used

bring to important savings of bandwidth. However, the ensemble task/scheduler becomes a system with memory (a Markov Chain), which is certainly more difficult to analyse.

5.3.3 An even better alternative

As surprising as it can seem, we can allow for delays and yet have a model of computation as easy to analyse as time-triggered with job dropout. To do so we have to completely give up the idea of periodic sampling and adopt an event-triggered sampling instead. However, contrary to other event-triggered alternatives proposed in the literature the event we look does not emanate from the system we control but from the controller itself: a new sample is acquired and a new job is released only when the previous job terminates (to be precise on the earliest release point available after the job finishes. This model is called Continuous Stream and is described in [21]. An example schedule is shown in Figure 5 As is it is possible to see, no new activation is triggered for Job 2 nor any sample is collected before Job 1 finishes. The same applies for Job 3, which is cancelled because its execution exceeds a maximum delay threshold.

The fact that a task starts only after the previous one finishes cancels any carry on execution across two adjacent jobs and the ensemble scheduler/task becomes memoryless. In other words, we never have a job in the wait queue of a task that is still in execution. The control theoretical consequence of this fact is that we can set up the stability problem by requiring that a matrix $\tilde{A}$, as defined in Equation 5, has all eigenvalues in the unit circle. This apparently trivial mathematical issues has profound and far reaching implication from the practical point of view. Consider Figure 6 taken from [21]. We considered 60 randomly generated dynamical systems. For each of them we synthesised an LQG controller with the same characteristics. The controller was supposedly implemented by a real-time task having worst case utilisation 1 (i.e., maximum computation time equal to the theoretical period used for the synthesis of the controller). The computation time of the task was a randmom variable distributed with a beta distribution. The plot reports the bandwidth used against the number of systems that could be stabilised with various models of computation. The line TT refers to the hard real time time triggered model, TT-MaxB refers to a time-triggered model with job drop-out (see Section 5.3.1), TT-SoRT refers to soft real-time system with delays (Section 5.3.2) and finally line CS refers to the continuous stream (Section 5.3.3). As we can see the continuous stream stabilises most of the systems with 40% of the bandwidth, with a substantial saving over the bandwidth required by other models of computation.

6 Conclusions

In this paper we presented an overview of the state of the art in soft real-time system. We have presented models for dealing with uncertainty in execution times and arrival times of tasks, and in particular we have discussed the resource reservation framework and the Constant Bandwidth Server algorithm for providing temporal isolation. We have analysed the typical requirements of soft real-time tasks, and we have presented a meaningful application to control system design.

In particular, the latter is important because control systems are often considered as hard real-time, and subject to very conservative design methodologies. We have shown how it is possible to improve their performance by using soft real-time techniques without renouncing to certification of the properties.

For further information on soft real-time scheduling techniques, the interested reader can refer to [10]. Programming tools and techniques for soft real-time scheduling with `SCHED_DEADLINE` can be found in `https://github.com/scheduler-tools/rt-app` and `http://retis.sssup.it/rts-like/index.html`.

References

[1] Luca Abeni and Giorgio Buttazzo. "Integrating multimedia applications in hard real-time systems". In: *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*. IEEE. 1998, pp. 4–13.

[2] Luca Abeni and Giorgio Buttazzo. "QoS guarantee using probabilistic deadlines". In: *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*. IEEE. 1999, pp. 242–249.

[3] Luca Abeni, Nicola Manica, and Luigi Palopoli. "Efficient and robust probabilistic guarantees for real-time tasks". In: *Journal of Systems and Software* 85.5 (2012), pp. 1147–1156.

[4] Luca Abeni et al. "Analysis of a reservation-based feedback scheduler". In: *Proc. 23rd IEEE Real-Time Systems Symp. RTSS 2002*. IEEE Computer Society, 2002, pp. 71–80. DOI: 10.1109/REAL.2002.1181563.

[5] Luca Abeni et al. "QoS management through adaptive reservations". In: *Real-Time Systems* 29.2 (2005), pp. 131–155. ISSN: 0922-6443.

[6] Sanjoy Baruah et al. "Generalized multiframe tasks". In: *Real-Time Systems* 17.1 (1999), pp. 5–22.

[7] Sanjoy K Baruah. "Feasibility analysis of recurring branching tasks". In: *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*. IEEE. 1998, pp. 138–145.

[8] Moris Behnam et al. "SIRAP: a synchronization protocol for hierarchical resource sharingin real-time open systems". In: *Proceedings of the 7th ACM & IEEE international conference on Embedded software*. ACM. 2007, pp. 279–288.

[9] Marko Bertogna, Nathan Fisher, and Sanjoy K. Baruah. "Resource-Sharing Servers for Open Environments". In: *IEEE Transations on Industrial Informatics* 5.3 (2009), pp. 202–219. DOI: 10.1109/TII.2009.2026051.

[10] Giorgio Buttazzo et al. *Soft Real-Time Systems*. Springer, 2005.

[11] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. "Capacity sharing for overrun control". In: *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*. IEEE. 2000, pp. 295–304.

[12] A. Cervin et al. "The jitter margin and its application in the design of real-time control systems". In: *Proceedings of the IEEE International Conference on Real-Time and Embedded Computing Systems and Applications*. Gothenburg, Sweden. 2004.

[13] Houssine Chetto, Maryline Silly, and T Bouchentouf. "Dynamic scheduling of real-time tasks under precedence constraints". In: *Real-Time Systems* 2.3 (1990), pp. 181–194.

[14] Tommaso Cucinotta et al. "A Real-time Service-Oriented Architecture for Industrial Automation". In: *Industrial Informatics, IEEE Transactions on* 5.3 (2009), pp. 267–277. ISSN: 1551-3203.

[15] Tommaso Cucinotta et al. "A robust mechanism for adaptive scheduling of multimedia applications". In: *ACM Transactions on Embedded Computing Systems (TECS)* 10.4 (2011), p. 46.

[16] Tommaso Cucinotta et al. "Adaptive real-time scheduling for legacy multimedia applications". In: *ACM Transactions on Embedded Computing Systems (TECS)* 11.4 (2012), p. 86.

[17] Tommaso Cucinotta et al. "Adaptive reservations in a Linux environment". In: *Real-Time and Embedded Technology and Applications Symposium, IEEE* (2004). DOI: 10.1109/RTTAS.2004.1317269.

[18] Tommaso Cucinotta et al. “On the Integration of Application Level and Resource Level QoS Control for Real-time Applications”. In: *IEEE Transactions on Industrial Informatics* 5.4 (2010), pp. 479–491. DOI: 10.1109/TII.2010.2072962.

[19] Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. “Analysis and Implementation of the Multiprocessor Bandwidth Inheritance Protocol”. In: *Real-Time Systems* 48.6 (2012). DOI: 10.1007/s11241-012-9162-0.

[20] Daniele Fontanelli, Luca Greco, and Luigi Palopoli. “Soft Real–Time Scheduling for Embedded Control Systems”. In: *Automatica* 49 (2013), pp. 2330–2338. DOI: 10.1016/j.automatica.2013.04.036.

[21] Daniele Fontanelli, Luigi Palopoli, and Luca Abeni. “The Continuous Stream Model of Computation for Real–Time Control”. In: *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE. Vancouver, Canada, 2013. DOI: 10.1109/RTSS.2013.23.

[22] Daniele Fontantelli, Luigi Palopoli, and Luca Greco. “Optimal CPU allocation to a set of control tasks with soft real–time execution constraints”. In: *Proceedings of the 16th international conference on Hybrid systems: computation and control*. ACM. 2013, pp. 233–242.

[23] Gene F Franklin, J David Powell, and Michael L Workman. *Digital control of dynamic systems*. Vol. 3. Addison-wesley Menlo Park, 1998.

[24] Graham Clifford Goodwin, Stefan F Graebe, and Mario E Salgado. *Control system design*. Vol. 240. Prentice Hall New Jersey, 2001.

[25] Hermann Kopetz and Günther Bauer. “The time-triggered architecture”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 112–126.

[26] Juri Lelli et al. “An efficient and scalable implementation of global EDF in Linux”. In: *Proceedings of the international workshop on operating systems platforms for embedded real-time applications (OSPERT)*. 2011.

[27] Caixue Lin, Scott Brandt, et al. “Improving soft real-time performance through better slack reclaiming”. In: *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*. IEEE. 2005, 12–pp.

[28] Giuseppe Lipari and Sanjoy K. Baruah. “Greedy reclamation of unused bandwidth in constant-bandwidth servers”. In: *Proc. 12th Euromicro Conf. Real-Time Systems Euromicro RTS 2000*. IEEE. IEEE Computer Society, 2000, pp. 193–200. DOI: 10.1109/EMRTS.2000.854007.

[29] Giuseppe Lipari, Gerardo Lamastra, and Luca Abeni. “Task Synchronisation in Reservation-Based Real-Time Systems”. In: *IEEE Transactions on Computers* 53.12 (2004), pp. 1591–1601. DOI: 10.1109/TC.2004.120.

[30] Luca Marzario et al. “IRIS: A new reclaiming algorithm for server-based real-time systems”. In: *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symp RTAS 2004*. IEEE. Published by the IEEE Computer Society, 2004, pp. 211–218. DOI: 10.1109/RTTAS.2004.1317266.

[31] Luís Nogueira and Luís Miguel Pinho. "Capacity sharing and stealing in dynamic server-based real-time systems". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE. 2007, pp. 1–8.

[32] L. Palopoli et al. "An Analytical Solution for Probabilistic Guarantees of Reservation Based Soft Real–Time Systems". In: *Parallel and Distributed Systems, IEEE Transactions on* PP.99 (2015), pp. 1–1. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2416732.

[33] L. Palopoli et al. "Real-time control system analysis: an integrated approach". In: *Proceedings of the IEEE Real-Time Systems Symposium*. 2000. DOI: 10.1109/REAL.2000.896003.

[34] Luigi Palopoli et al. "An analytical bound for probabilistic deadlines". In: *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE. 2012, pp. 179–188.

[35] Luigi Palopoli et al. "AQuoSA - adaptive quality of service architecture". In: *Software: Practice and Experience* 39.1 (2009), pp. 1–31. ISSN: 1097-024X.

[36] Luigi Palopoli et al. "Weighted feedback reclaiming for multimedia applications". In: *Proceedings of the 6th IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, ESTIMedia*. Ed. by Petru Eles and Andy D. Pimentel. IEEE, 2008, pp. 121–126. ISBN: 978-1-4244-2612-6. DOI: 10.1109/ESTMED.2008.4697009. URL: http://dx.doi.org/10.1109/ESTMED.2008.4697009.

[37] Rodolfo Pellizzoni and Marco Caccamo. "M-cash: A real-time resource reclaiming algorithm for multiprocessor platforms". In: *Real-Time Systems* 40.1 (2008), pp. 117–147.

[38] Ragunathan Rajkumar et al. "Resource kernels: A resource-centric approach to real-time and multimedia systems". In: *Photonics West'98 Electronic Imaging*. International Society for Optics and Photonics. 1997, pp. 150–164.

[39] Claudio Scordino and Giuseppe Lipari. "A Resource Reservation Algorithm for Power-Aware Scheduling of Periodic and Aperiodic Real-Time Tasks". In: *IEEE Transactions on Computers* 55.12 (2006), pp. 1509–1522. DOI: 10.1109/TC.2006.190.

[40] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. "Priority inheritance protocols: An approach to real-time synchronization". In: *Computers, IEEE Transactions on* 39.9 (1990), pp. 1175–1185.

[41] Martin Stigge et al. "The digraph real-time task model". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE. 2011, pp. 71–80.

[42] Steve Vestal. "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance". In: *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. IEEE. 2007, pp. 239–243.