Experimental evaluation of slack management in real-time control systems: coordinated vs. self-triggered approach ⋆

Manel Velasco [a], Pau Martí [a,∗], Josep M. Fuertes [a], Camilo Lozoya [a], Scott A. Brandt [b]

[a] *Automatic Control Department, Technical University of Catalonia, Pau Gargallo 5, 08028 Barcelona, Spain*

[b] *Computer Science Department, University of California at Santa Cruz, 1156 High Street, Santa Cruz, CA 95064, USA*

Abstract

Effective slack management, i.e. management of unused computing resources, for real-time control tasks mandates to redistribute the available resources between controllers as a function of the state of the controlled plants. Slack can be allocated to control tasks to alter their rate of progress via e.g., the controllers' period, in order to adapt their behavior to changes in the computing platform and in the environment. This paper presents an experimental evaluation of two representative slack redistribution policies for multitasking real-time control systems: "coordinated" vs. "self-triggered". In the coordinated policy a resource manager is responsible for modifying each control task progress. Alternatively, in the self-triggered policy, each control task decides its progress. The demands that each policy poses to the computing platform are analyzed and different operating system architectures providing flexibility and adaptivity are discussed. A proof-of-concept implementation including the real-time control of three double integrator plants in the form of electronic circuits is presented, and a complete performance analysis is reported.

Key words: Real-time control systems, Slack management, Performance evaluation

1. Introduction

The use of computer controlled systems has increased dramatically in our daily life. Processors and microntrollers are in most of the devices we use every day, such as mobile phones, cars, dishwashers, etc. Due to cost constraints, many of these devices that run control applications are designed under space, weight, and energy constraints, i.e., with limited resources [1].

For control applications, this trend demands holistic approaches able to efficiently use the computing resources, or alternatively, able to achieve the best performance within the available resources. Holistic approaches often require the integration of control systems and real-time systems theory [2]. Within this scenario, this paper focuses on experimental evaluation of control and real-time co-design approaches that efficiently balance control performance and resource utilization when a set of tasks controlling physical plants is concurrently executed on a processor.

For control tasks, the task period is given by the sampling period used in the controller design stage. It has to balance the desired control performance and the feasible computational demand [3]. The sampling period can be selected from a range of

⋆ This work has been partially supported by ARTIST2 NoE IST-2004-004527, by ArtistDesign NoE IST-2008-214373, and by C3DE CICYT DPI2007-61527.

∗ Corresponding author. Tel/Fax: +34 938967798 / +34 938967700

Email address: pau.marti@upc.edu (Pau Martí).

values. Short periods allow quick reactions when the controlled system is affected by perturbations (which is positive from a control point of view), but increase the processor's load (which is negative from a resource utilization point of view). Long periods decrease the processor's load but may give poor control performance.

This situation calls for models that can dynamically accommodate different values for the control task period according to changes in the computing platform such as workload variations, and according to changes in the controlled plants such as perturbations' arrivals. Recent research [4–15] has theoretically shown that dynamic resource allocation improves control performance and/or minimizes resource utilization for a set of concurrently executing control tasks. In particular, many of them mandate to redistribute the available slack (unreserved resources or reserved but unused resources [16]) as a function of the state of the controlled plants.

As indicated in [17], among these results, two main tendencies can be identified: feedback scheduling and scheduling of event-driven control systems. The former primarily looks at the problem of optimizing control performance while the latter focuses on minimizing resource utilization. This paper evaluates two approaches, one from each of these tendencies.

The first approach [6] (further extended in [14]), belonging to the class of feedback scheduling methods, presented an optimal slack redistribution policy for control tasks. A resource manager collects the state of all plants and allocates resources via the controllers' periods while guaranteeing tasks timing constraints. Henceforth, this approach is refereed as *coordinated*. This name reflects the fact that the resource manager acts as a coordinator and it is responsible of determining each task progress.

The second approach [5], belonging to the class of event-based methods, presented a task model where each task instance decides whether to consume slack whenever available considering the state of its controlled plant. Henceforth, this approach is refereed as *self-triggered*. This name reflects the fact that the model allows each control task to adaptively trigger itself.

1.1. *Paper contributions and structure*

This paper presents an experimental evaluation of these two alternative approaches to slack management in real-time control systems. The paper starts by reviewing the key theoretical aspects of the approaches under evaluation. Then, it discusses which operating system architecture suits the demands of the coordinated and self-triggered approaches. The main decision variables required for successful implementation of both policies are identified, and their operation described.

Afterward, the experimental set-up is introduced. A description of the computing platform including the controlled plants and details of the implementation of these policies is given. Finally, an extensive comparative performance evaluation is presented. The evaluation focuses not only on control performance but also on resource utilization. Experimental data is also used to explain the impact of some key strategies adopted in the implementation of the two policies.

The rest of this paper is structured as follows. Section 2 summarizes the theoretical aspects of both policies. Section 3 discusses operating system support issues and the implementation strategy. Section 4 describes the experimental setup and the implementation details of each solution. Section 5 presents the experimental results. Section 6 summarizes the lessons learned. Finally, Section 7 concludes the paper.

2. Theoretical aspects

Consider a real-time system with n control tasks, each one controlling a plant, to be executed on a single processor. Each plant i can be described by the linear continuous-time state-space form [1]

$$\dot{\boldsymbol{x}}_i(t) = A_i \boldsymbol{x}_i(t) + B_i u_i(t) \quad t \in \mathbb{R}^+ \tag{1}$$

$$y_i(t) = C_i \boldsymbol{x}_i(t), \tag{2}$$

called *state* and *output* equations respectively, where matrices A_i, B_i and C_i are of the appropriated dimensions, $u_i(t)$ and $y_i(t)$ are the *input* and *output* of the plant, and vector $\boldsymbol{x}_i(t) = [x_i^1(t), \ldots, x_i^n(t)]$ is the state of the system at time t; its elements are called *state-variables*. For each plant, we define the norm of its state variables as the plant error

$$e_i(t) = |\boldsymbol{x}_i(t)|. \tag{3}$$

In terms of timing constraints, each control task is characterized by its period h_i (corresponding to

[1] Henceforth, the subscript i will specify a control loop, identifying either the controller or the controlled plant.

the sampling period [2]) and its worst-case execution time c_i, which corresponds to the sequential execution of sampling, control algorithm computation, and actuation. Each task deadline is assumed to be equal to the task period. However, having deadlines different than the task period would not alter the approach and results here reported.

For the set of n tasks, if deadlines are equal to periods, the task set processor utilization factor is

$$U = \sum_{i=1}^{n} \frac{c_i}{h_i}, \quad (4)$$

which is the fraction of processor time spent in the execution of the task set [18]. Note that $U \geq 0$, and that $U = 1$ denotes that the processor is fully utilized. Therefore, U provides a measure of computational load.

The *rate* or partial utilization factor of each task

$$r_i = \frac{c_i}{h_i}, \quad (5)$$

is the processor share (resource requirement) that each control task requires for a given period. Since the worst-case execution time of each control task is constant, any variation in task rate implies a corresponding variation in task period (and vice-versa). The two slack redistribution policies consider that a minimum rate $r_{i,min}$ is guaranteed to each control task, which is given by the longest task period $h_{i,max}$ and causes the lowest processing demand. The static allocation permits guaranteeing that control performance specifications are fulfilled.

Given the static allocation and given that controllers will provide better control performance when allocated more processor time, if slack is available U_s, the problem solved by the coordinated and the self-triggered approach is to decide for each control task how the rate should be increased (i.e., how the period should be shortened),

$$r_i = r_{i,min} + \Delta r_i \quad (6)$$

such that the overall control performance is improved subject to

$$\sum_{i=1}^{n} \Delta r_i \leq U_s. \quad (7)$$

[2] Note that in real-time systems notation, task period is usually denoted by P or T. Here the control systems notation for period is used.

Constraint (7) specifies that the additional resources Δr_i that will be given to each control task must be less or equal than the available slack U_s, which is considered to be the amount of unreserved processor time.

Summarizing, the resource management, either coordinated or self-triggered, has to obey the following specifications:

- For each control task, the bigger the error (3), the higher the rate r_i (or the shorter the sampling period h_i) to be allocated.
- For the set of control tasks, a given constraint on the utilization factor must be kept (7).

Note that these two specifications may conflict because the first attempts to have locally higher resource allocations while the second restricts the overall (global) resource utilization.

2.1. *Coordinated approach*

In the coordinated approach [6], each control task τ_i is characterized by its rate r_i, its linear benefit function $p_i(r_i) = \alpha_i r_i + \beta_i$ that reflects the assumption that controllers will provide better control performance when given more resources, and its controlled system error e_i.

Linear benefit functions are common in feedback scheduling approaches, e.g. [4]. As explained in [6] other type of benefit functions different than linear can be easily incorporated into the problem formulation without compromising the feasibility of the solution. In this case however, the solution to slack redistribution would be different than the one explained next. Note also that for feedback scheduling approaches, in [14] it was shown that these type of benefit functions together with the formulation of the slack redistribution problem as a performance maximization optimal problem would provide similar processor allocations than those achieved by cost minimization optimal problems formulated using quadratic cost functions.

In addition, it has been shown for example in [9] that using quadratic cost functions in feedback scheduling approaches leads to optimization problems that do not have explicit analytical solutions, and therefore, their implementation needs to be done using approximated solutions. In summary, although having linear benefit functions can be a simplifying assumption, if the plant under control increases performance linearly with the rate, the application of the policy presented in [6] is a good

choice.

For a given set of n control tasks, $\tau_1, \ldots, \tau_n$, the coordinated slack redistribution was formulated as a constrained optimization problem

$$\text{maximize} \quad \sum_{i=1}^{n} w_i e_i p_i(r_i) \tag{8}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \Delta r_i \le U_s \tag{9}$$
$$\Delta r_i \ge 0$$

where the solution are the rate increments Δr_i, $i = 1, \ldots, n$, and weights w_i in (8) can be used to permit appropriate comparisons between control loops.

The solution to (8)-(9) states that all the available slack U_s must be assigned to the control task with maximum $w_i e_i \alpha_i$. If all of the functions p_i and all the weights w_i are the same (i.e. the controlled plants are equal and of equal importance), all of the available slack must be assigned to the control task with the largest error e_i.

The coordinated approach tackles the slack redistribution problem considering that a coordinator, i.e. resource manager, knows the state of all controlled plants and the available slack, and then it implements the optimal solution. In addition, the application of the optimal policy requires the implementation of controllers capable of running with different sampling frequencies given different resource allocations. To do so, controllers are designed for the class of linear systems (1)-(2) using classic design procedures. In particular, let

$$\boldsymbol{x}_{n+1} = \Phi(h)\boldsymbol{x}_n + \Gamma(h)u_n \tag{10}$$

be the discretization of the system equation (1) [19]. The input is given by

$$u_n = -L(h)x_n \tag{11}$$

where $L(h)$ is a parametric standard state feedback control gain on the sampling period, that is, the control law depends on h. For each controller, a range of sampling periods $h = [h_{min}, \ldots, h_{max}]$ [3] is specified for which the closed loop requirements are met. The controller, implemented within a task, is allowed to execute with a run-time period that belongs to the specified range, adapting the gain accordingly. See [6] and its extension [14], and references therein for further details on the optimization, controller design and stability analysis.

[3] Note that the i-subscript is omitted to simplify the notation.

2.2. *Self-triggered approach*

The self-triggered approach tackles the problem considering that no central entity coordinating the resource allocation exists. With this assumption, the problem to be solved was treated as a decentralized management of the available slack among all control tasks [5]. Although the analysis in this paper focuses on uniprocessor systems, the self-triggered approach gains interest on multiprocessor systems, e.g., multi-core platforms, and more important, in networked control systems. In this case, when sensors, controllers and actuators are implemented in different nodes, the run-time optimization procedures of the feedback scheduling approaches can not be performed straightforward because the information required such as plants' current states is physically distributed. Therefore, different approaches to slack redistribution are required. And event-based control approaches, such as those based on self-triggered controllers, seems to perfectly fit for these decentralized architectures. However, their evaluation in terms of control performance and resource utilization has not been experimentally analyzed. This further motivates the comparison of the self-triggered approach versus the coordinated approach, even in a uniprocessor system.

A control approach to resource allocation at the task level capable of ensuring global resource utilization for all participating tasks was the basis for the solution in [5]. To do so, the available slack U_s was assumed to be known for each control task.

The key idea of this approach was to extend the discrete state space form of each plant (10) by imposing the desired slack management dynamics

$$\begin{bmatrix} \boldsymbol{x}_{n+1} \\ h_{n+1} \end{bmatrix} = \begin{bmatrix} \Phi(h) \cdot \boldsymbol{x}_n \\ \Upsilon(x_n, U_s) \end{bmatrix} + \begin{bmatrix} \Gamma(h) \\ 0 \end{bmatrix} u_n,$$

in terms of a new state variable: the desired sampling period h_{n+1} for the next task instance execution. In general, the desired dynamics are a function of the plant state and the available slack. The dynamics for h_{n+1} in [5] were heuristically specified as

$$h_{n+1} = (h_{max} - h_{min})\, e^{-K|\boldsymbol{x}_n|} + h_{min} \tag{12}$$

in such a way that it mathematically behaves as required by the problem specifications. If there is no

error ($|\boldsymbol{x}_n| = 0$), then $h_{n+1} = h_{max}$. And if the error increases, the sampling period decreases (and vice versa). By being a function of the exponential of the norm of the original state variables $\boldsymbol{x}_n$, (12) ensures positive values for the sampling period as well as smooth transitions between successive values. It also takes into account the available slack U_s. This is achieved by defining the shortest possible sampling period h_{min} that can be assigned to a control loop as

$$h_{min} = \frac{c}{U_s + r} \tag{13}$$

where r is the current task rate as defined in (5). Finally, h_{max} and K are the longest possible period given by the static allocation, and the *criticalness*, both to be assigned for each control task. The criticalness determines how quick a control loop will increase or decrease its period according to its error. Higher values for K will imply more abrupt changes in the sampling periods.

The model (12) is nonlinear. Note that this type of non-linear models is common for self-triggered control approaches [13], [15], as indicated in [20]. Although being non-linear, these type of models present the advantage that the plant dynamics and the period dynamics can be treated separately (note also that the input u_n does not directly affect the second state variable). Therefore, since the plant dynamics were defined as linear, the control input can be given by a parameterized standard linear controller (11), equal to the case of the coordinated approach (the same type of control design and stability analysis would apply). Having similar controllers for both approaches allows easier control performance evaluations as well as more fair resource utilization analysis.

3. Architecture

The theoretical analysis of the coordinated and self-triggered approaches to slack redistribution demands a flexible real-time operating system support. In addition, the implementation architecture has to take into account that the main variables that have to be considered for solving the slack redistribution problem originate from two different domains. Slack has to be redistributed according to the plant state. The state is an information that belongs to the application or user-level domain. However, the available slack is an information that belongs to the operating system domain.

3.1. *Flexible operating system support*

Effective slack redistribution requires that all of the controllers be capable of running with different sampling frequencies given different resource allocations. Each controller can be considered a flexible real-time process with flexible period choices. Dynamic resource allocation for controllers can be achieved by any existing real-time operating system or kernel supporting scheduling frameworks or scheduling algorithms which permits dynamic task period adjustment at run time and guarantees no deadline miss during the adjustment.

Therefore, the required real-time system support for implementing the slack redistribution policies should enforce timeliness with a certain degree of flexibility, trading off predictability in the performance and efficiency in the resource utilization, as also demanded by other type of modern control applications [21]. The standard approach for achieving these goals is the reflective architecture for real-time systems [22].

Reflection is commonly understood as a mechanism by which a program becomes "self-aware", checks its progress and can change itself or its behavior. This is achieved by allowing applications to access kernel data structures using application program interfaces (APIs) to obtain and modify information about the current system state. Examples include the Shark kernel [23], Flexible Real-Time Linux [24], Marte OS [25], RBED [26], Erika [27] or the minimal real-time kernel presented in [28]. A complementary view of reflection is when the kernel structures contain application data, which is then used by the kernel to alter the progress of each task.

Both approaches to slack management demand a reflective architecture capable of 1) providing the required flexibility in terms of accommodating task rate changes, and 2) facilitating communication mechanisms between kernel and applications' spaces for passing the required information to accomplish slack management.

3.2. *Implementation strategy*

The slack redistribution in the self-triggered approach is done, by definition at the user level, because each task rate of progress is decided by the task itself. To do so, each task needs to know the available slack U_s, which must be made accessible by the underlying real-time system, as well as, each

plant state vector, $\boldsymbol{x}_i$. With both informations, the new rate of progress, in the form of the next sampling period h_i, is calculated using (12). Each newly calculated h_i is passed into the real-time system, which sets the new rate for each task.

The slack redistribution in the coordinated approach can be implemented using to complementary strategies: at the kernel level or at the user level. The first strategy, which was already used in [6], is to specify that each task rate of progress is decided by the real-time system. To do so, the kernel needs to know all plants states $\boldsymbol{x}_i$ and the available slack, U_s. Therefore, control tasks need to pass the plants states into the kernel. With this information, the kernel calculates the slack redistribution Δr_i, which corresponds to the solution (8)-(9). Having all Δr_i, the new tasks rates can be computed and set in the real-time system. In addition, each new task rate r_i is passed back to each task, in order to allow each control task to correctly calculate control actions. The second strategy, used in several feedback scheduling approaches, e.g. [4], [7] or [9], is to use a high priority periodic task, namely feedback scheduler, to perform the slack redistribution. To do so, this task needs to know all plants states $\boldsymbol{x}_i$ and the available slack U_s, that must be made accessible by the real-time system. With this information, it calculates the slack redistribution as before, and sets the new periods in the real-time system.

For comparative purposes in the performance evaluation, the second strategy for the implementation of the coordinated approach is chosen. Therefore, both policies will be implemented in the user level space. This will facilitate the overhead analysis, and it will remove possible misleading interpretations of the presented performance results that may arise if one strategy was implemented at the user level and the other at the kernel level.

4. Experimental set-up

A proof-of-concept implementation for evaluating the two slack redistribution policies is presented. Three equal unstable plants in the form of double integrator electronic circuits are controlled by three control tasks concurrently executing on the Erika real-time kernel and scheduled under the Earliest Deadline First (EDF, [29]) scheduling algorithm. The EDF scheduling policy has been configured with a *tick* of $1ms$. The executing platform is the full Flex board [30] equipped with a dsPIC micro-processor.

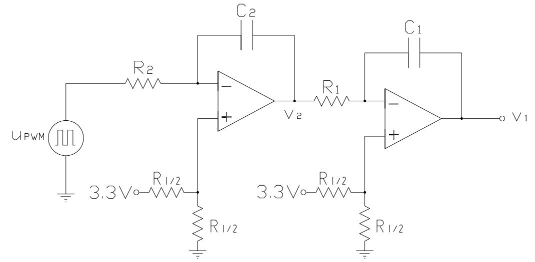

Fig. 1. Electronic double integrator circuit

Table 1
Electronic components nominal and validated values

Component	Nominal value	Validated value
$R_{1/2}$	1 kΩ	1 kΩ
R_1	100 kΩ	100 kΩ
R_2	100 kΩ	100 kΩ
C_1	470 nF	420 nF
C_2	470 nF	420 nF

4.1. *Plant details*

The electronic double integrator is illustrated in Figure 1. Note that in the integrator configuration, the operational amplifiers require positive and negative input voltages. Otherwise, they will quickly saturate. However, since the circuit is powered by the dsPIC, and thus no negative voltages are available, the 0V voltage (V_{ss}) in the non-inverting input has been shifted from GND to half of the value of V_{cc} (3.3V) by using a voltage divider $R_{1/2}$. Therefore, the operational amplifier differential input voltage can take positives or negatives values. The nominal electronic components are shown in Table 1.

An integrator implemented by an operational amplifier is modeled by

$$V_{\text{out}} = \int_0^t -\frac{V_{\text{in}}}{RC}\,\mathrm{d}t + V_{\text{initial}} \qquad (14)$$

where V_{initial} is the output voltage of the integrator at time $t = 0$, and V_{in}, V_{out} are the input and output voltages of the integrator, respectively.

From (14) and the scheme shown in Figure 1, the double integrator plant dynamics can be modeled by

$$\begin{bmatrix} \dot{v}_1 \\ \dot{v}_2 \end{bmatrix} = \begin{bmatrix} 0 & \frac{-1}{R_1 C_1} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{-1}{R_2 C_2} \end{bmatrix} u$$
$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}. \qquad (15)$$

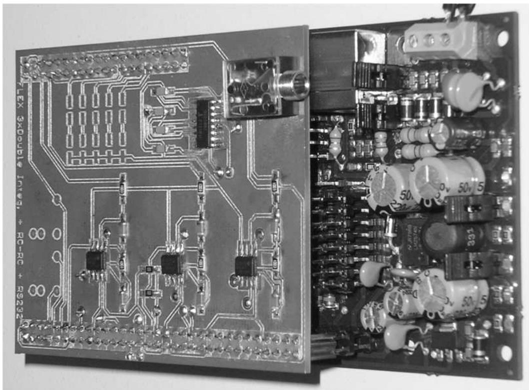

Fig. 2. Experimental setup

Knowing the existing tolerances in the electronics components (5% for resistors and 25% for capacitors), the mathematical model used for controller design that best matches the circuit dynamics is given by (15) with the validated values listed in Table 1.

Note that the plant is unstable because the eigenvalues of the system matrix are $\lambda_{1,2} = 0$. Each controller objective would be to have the circuit output voltage V_1 tracking random set point changes.

The three double integrators have been implemented in a daughter board plugged on top of the flex board, as illustrated in Figure 2. To debug and extract information from the micro-controller to a standard computer, a RS232 link has been established. This link is managed from the computer side using Matlab©.

4.2. *Code implementation details*

In this section we describe the implementation of both slack redistribution policies in Erika. The exchange of reflective information between kernel and control tasks is achieved by means of a system call and by accessing shared memory.

In Erika, the clock is generated by the periodic interrupts of a timer, that is accumulated in a *counter* at each execution of the interrupt handling routine. Each periodic task is attached to the counter using *alarms*, defining its first activation time and the interval (period) between consecutive activation times. When the alarm fires, the task is activated at the next execution of the interrupt handling routine (from idle becomes ready), and executed according to EDF. The implementation of the slack redistribution policies uses the system call *set_rel_alarm* for reconfiguring the alarms associated to the control tasks when new periods have to be applied.

Periodic controller task
{
  x_i^1 := **read_input**()
  $\hat{x}_i$:= **observer**(x_i^1, t_i^a)
  u_i := **calculate_output**($\hat{x}_i, h_i$)
}

Fig. 3. Pseudo-code for a standard periodic control task

4.2.1. *Standard controller*

The implementation of control algorithms using real-time periodic tasks can generate job executions prone to violate the required periodicity of the sampling and actuation operations, problem known as sampling and latency jitter [3]. To overcome this problem, the research literature offers several solutions, e.g., the Control Server model [31] or the One-Shot Task model [32]. In the implementation, the second solution has been adopted, although the application of the first one would have produced the same results. In short, this model accommodates non-periodic sampling (that is generated by scheduling interferences if sampling is executed at the beginning of each job execution) and then enforces actuation at synchronized time instants. After each sample, the state at the actuation times can be reconstructed from the sample using an advanced observer (predictor). Note that the same advanced observer is used for observing the second state variable V_2. Therefore, the one shot task model does not add computational overhead compared to an implementation using any other approach because the observer for V_2 is still required.

Figure 3 shows the pseudo-code for a standard real-time periodic controller, which is activated at each sampling period h_i. It samples the first state variable x_i^1, observes the state (which is the implementation of the one shot task model) $\hat{x}_i$ at the actuation instant t_i^a, and computes the control input u_i. The application of the u_i to the plant (actuation) is performed by the kernel in the interrupt handling routine at each t_i^a relative to the job release time. In particular, in the implementation, $t_i^a = h_i$, that is, actuation occurs at the next job release time. The actuation could have been also implemented in a separated periodic task or using a separate hardware interrupt.

4.2.2. *Coordinated implementation*

The implementation of the coordinated approach uses a periodic task for each controller and the feedback scheduling task for the computing the slack redistribution.

```
Coordinated controller task
{
    h_i^next := read_shared_memory()
    x_i := read_input()
    x̂_i := observer(x_i^1, t_i^a)
    u_i := calculate_output(x̂_i, h_i^next)
    write_shared_memory(x̂_i)
}

Feedback scheduler task
{
    U_s := read_shared_memory()
    for (each_control_task)
    {
        x_i := read_shared_memory()
        if (w_i e_i α_i is maximum) Δr_i := U_s
        else Δr_i := 0
        h_i^next = c_i / (r_{i,min} + Δr_i)
        write_shared_memory(h_i^next)
        set_rel_alarm(h_i^next)
    }
}
```

Fig. 4. Pseudo-code for the two tasks coordinated approach

Figure 4 shows the pseudo-codes for a control task and for the feedback scheduler. The only difference between the code of the control task in the coordinated approach with respect to the standard one (Figure 3) is the accesses to shared memory for obtaining the sampling period that applies, or for writing the state. The main job of the feedback scheduler is to compute the sampling period for each control task according to the optimal policy and reseting the control tasks periods using the system call *set_rel_alarm*. Accesses to shared memory are also required for obtaining the available slack, the plants states and for writing the computed sampling periods. It is out of the scope of this work to derive methods for slack computation at the kernel level (see [16] and references therein for further information).

For achieving an efficient implementation of the optimal policy in terms of processor utilization, when all the controlled plants are in equilibrium, i.e., $|x_n| = 0$, the three control tasks are set to execute with their longest period. In terms of implementation, and considering the process and measurement noise, this has been achieved by specifying the following threshold: if the controlled plant state fulfils that $|x_n| < 0.05V$, then the plant is considered to be in the equilibrium.

4.2.3. *Self-triggered implementation*

The implementation of the self-triggered approach only requires coding the control tasks because they are in charge of performing the slack redistribution. Figure 5 illustrates the pseudo-code for a self-triggered control task. Apart from accessing shared memory, the main differences with respect to the code of a control task of the standard approach (or the coordinated approach) is the computation of the next sampling period, and the reconfiguration of its period using the system call *set_rel_alarm*. Note that these two operations are similar to the main operations performed by the feedback scheduler. However, the feedback scheduler, at each job execution, performs them as many times as control tasks are in the system. As in the coordinated policy, for implementation effectiveness and due to the noise, the computation of the next sampling period is only performed when $|x_n| \geq 0.05V$. Otherwise, the period is set to the maximum. Therefore, the same threshold has been specified for detecting the plant in equilibrium and forcing then the lower execution rate for the control task.

```
Self-triggered controller task
{
    U_s := read_shared_memory()
    x_i^1 := read_input()
    x̂_i := observer(x_i^1, t_i^a)
    h_i^next := Υ(x̂_i, U_s)
    u_i := calculate_output(x̂_i, h_i^next)
    set_rel_alarm(h_i^next)
}
```

Fig. 5. Pseudo-code for the self-triggered approach

4.3. *Controller design*

Each controller implements the same parametric control law obtained by optimal control techniques, but parameterized on the sampling period.

The controller gain L corresponds to the discrete Linear Quadratic Regulator for (15) with the validated components' values, which minimizes a discrete cost function equivalent to the continuous cost function

$$J = \int_0^\infty (x^T Q x + u^T R u) dt \tag{16}$$

with Q being the identity and $R = 10$, for the different sampling period choices. The equivalent continuous closed loop poles are $p_{1,2} = -10.1885 \pm 8.6869i$, which determine that the feasible sampling periods should be less than $h = 70$ms [19].

For the observer design, a Kalman filter was designed taking into account the following noise covariances $Q_n = E(w \cdot w^T) = 1 \cdot 10^{-10} BB^T$ and $R_n = E(v \cdot v^T) = 5 \cdot 10^{-5}$, extracted from the elec-

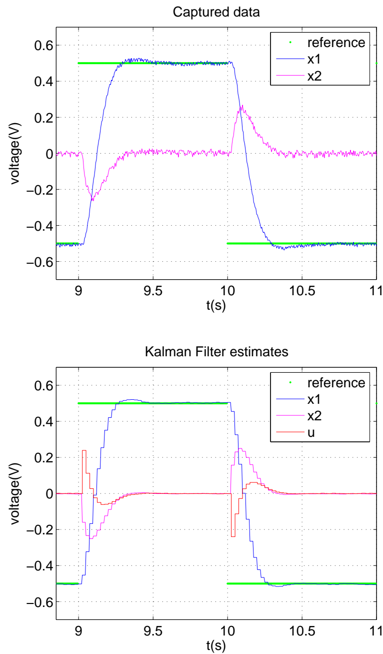

Fig. 6. Experimental data for observer design

tronic circuit, where w and v are the plant noise and the measurement noise, respectively.

In particular, for a sampling period $h = 25$ms, Figure 6 shows the response of a single double integrator circuit controlled by a real-time task using the *standard* implementation strategy with the Kalman observer. The top sub-figure shows the plant response by plotting the reference signal and the two state variables. The bottom sub-figure shows the corresponding Kalman filter state variable estimates given by the observer, as well as the control signal u. It can be concluded that the Kalman filter has the ability of removing the inherent noise.

5. Performance evaluation

This section presents the main results obtained from all the experiments. The two slack redistribution approaches have been tested under different scenarios, and the summarized results presented next are general and not limited to specific cases.

5.1. *Workload generation*

In the following experiments, the three control tasks controlling each double integrator circuit implemented the same optimal controller described in Section 4.3 parameterized on the sampling period choices. A uniform workload was used to simplify the performance analysis and comparison. For the same reason, a constant slack availability (U_s) is considered in the kernel. This permits to fairly evaluate the dynamic slack policies. To provide a direct comparison with traditional control system implementations, a baseline policy, *static*, was also implemented. In the static policy all controllers always share the available resources equally and no dynamic redistribution is used. The static policy implements the "standard" controller (described in section 4.2.1) and is used to examine the overall performance benefit of adaptive slack redistribution allocation.

5.1.1. *Perturbations*

For each of the policies, static, coordinated and self-triggered, the three controllers track randomly generated perturbations in the form of $1V$ set-point changes. They are generated with different average intervals in order to capture all the possible scenarios. Specifically, the average intervals are 0.75s, 1s, 1.5s, 3s, and 6s, and during each interval three set-point changes occur. This means that within each perturbation interval, the three control tasks are subject to a set-point change that occurs at random time instants but shifted one third in average.

For example, looking at the perturbation interval of 1s, during the first interval, the first control task receives a set-point change around 0.3s, the second task receives the set-point change around 0.6s, and the third around 0.9s. Therefore, having a short perturbation interval means that the set-point changes affecting the three plants, although shifted, are close enough that all the plants are in transient during the interval. And having a long perturbation interval means that only one plant is in transient at a time because each plant settles before any other plant receives a set-point change. In summary, the three plants can be continuously perturbed or almost never perturbed.

5.1.2. *Period choices*

The choice of allowed sampling periods for each slack redistribution method is a key point for providing a fair comparison between them. They have been designed taking into account that in the worst case scenario, at the perturbation arrival, the processor utilization should be the same. This demands knowing worst case execution times. Table 2 shows the main time consuming operations for each method, which have been measured using an

Table 2
Worst case execution times

	Operations	Static	Coord.	Self
1 Controller	read_input	28μs	28μs	28μs
	observer	116μs	116μs	116μs
	calculate_output	48μs	48μs	48μs
	compute e_i			3μs
	compute $\Upsilon()$			61μs
	set_rel_alarm			5μs
	other	5μs	5μs	5μs
Total		197μs	197μs	266μs
Feedback scheduler	compute max e_i		14μs	
	3 set_ref_alarm		15μs	
Total			29μs	

oscilloscope plugged to the board.

For each control task of the coordinated policy, the shortest and longest sampling periods are $h_{i,min}^{coord} =$ 25ms and $h_{i,max}^{coord} =$ 50ms respectively. In this case, the worst scenario during 50ms demands executing each control task one time, plus another execution of the control task with highest error, plus two executions of the feedback scheduler task. Note that the period of the feedback scheduler should be equal to the shortest period of any of the control tasks running in the system, which in this case is $h^{fs} =$ 25ms. Therefore, using the numbers of Table 2, the utilization of the coordinated policy during 50ms and considering the worst case scenario is

$$U_{coord} = \frac{4control + 2feed.sch.}{50ms} = \frac{846\mu s}{50ms} \approx 1.7\%.$$

Note that U_{coord} denotes processor usage for those tasks involved in the coordinated policy. But in the implementation other tasks for extracting data were also executing during the experiments for the coordinated policy, as well as for the static and self-triggered.

For the static method, periods are computed considering that during $50ms$ the utilization should be equal to the one of the coordinated, that is

$$h_i^{static} = \frac{3control}{U_{cent}} = \frac{591\mu s}{0.017} \approx 35ms.$$

Finally, for the self-triggered approach, the maximum sampling period is obtained by considering that in the worst scenario four controllers execute, which should also have the same resource usage than the coordinated policy, that is

Table 3
Sampling periods in the experiments

	Sampling period	Value
static	h_i^{static}	$35ms$
coordinated	$h_{i,max}^{coord}$	$50ms$
	$h_{i,min}^{coord}$	$25ms$
	h_i^{fs}	$25ms$
self-triggered	$h_{i,max}^{self}$	$63ms$
	$h_{i,min}^{self}$	$31ms$

$$h_{i,max}^{self} = \frac{4control}{U_{cent}} = \frac{1064\mu s}{0.017} \approx 63ms.$$

And the minimum period is set to be half of the maximum, as in the coordinated.

As a summary, Table 3 shows the sampling periods used in the experiments. The choice of periods for the coordinated and the self-triggered policies are not the same. The difference in periods is due to the high computational cost of the computation of the next sampling instant in each job of a self-triggered controller.

5.2. *Performance analysis*

The metrics that are evaluated are aggregated control performance of the three control loops and processor utilization. Specifically, for each control loop, control performance is evaluated using the discrete cost function

$$J^d = \sum_{k=1}^{experiment_time} (x_k^T Q x_k + u_k^T R u_k)$$

where matrices Q and R are set as before and where each k-state and $k-$control signal is extracted from the board every 5ms. The k-subscript rather than the n-subscript is adopted to note that this data was periodically extracted. Then, for each policy, control performance is evaluated by looking at the total cumulative cost of the three loops.

Subsection 5.2.1 compares the coordinated and self-triggered approaches versus the static approach in terms of control performance, and subsection 5.2.2 focuses on processor utilization. Subsection 5.2.3 summarizes the experimental results and subsection 5.2.4 discusses the overhead analysis. In the following, the self-triggered slack redistribution policy, the criticalness parameter has been set to $K =$ 4. This choice is further explained in section 5.3.

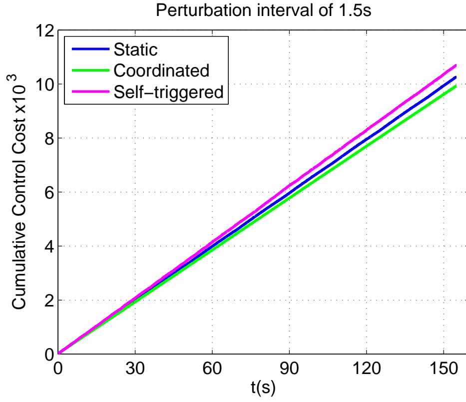

Fig. 7. Cumulative cost for the three policies

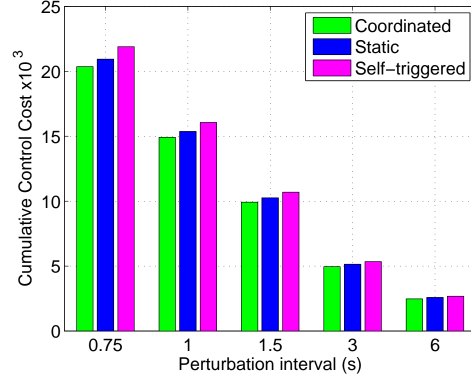

Fig. 8. Cumulative control cost histogram for the three policies and for all the perturbation intervals

5.2.1. *Control performance analysis*

Figure 7 shows the control performance in terms of cumulative cost of the three control tasks, running for 150s with perturbation interval of 1.5s. The lower the curve, the better the performance. The figure shows that the coordinated slack redistribution policy improves overall control systems performance compared to the static and self-triggered policies. However, the self-triggered slack redistribution is not able to improve control performance with respect to the static.

Figure 8 gives a complementary view of the control performance analysis. It shows the cumulative control cost of the coordinated, self-triggered and static policies, for different perturbation intervals, during execution runs of 150s. For each perturbation interval, the lower the bar, the better the policy in terms for control performance, i.e., the lower the control cost. For the three policies, longer perturbation intervals derive in lower costs because less set-point changes are applied.

In Figure 8 it can be seen that the coordinated policy always achieves better control performance than the other two policies regardless of the perturbation interval. However, by looking at the self-triggered policy, it can be seen that it always perform worse than the static policy.

As outlined in section 5.1, the self-triggered policy applies in general longer sampling periods than the coordinated, and therefore, its performance can not outperform the one achieved by the coordinated policy. And the self-triggered policy can not outperform the the static policy due to a similar reason. In most of the job executions of self-triggered controllers, the sampling period that applies is longer than the period of the static controllers.

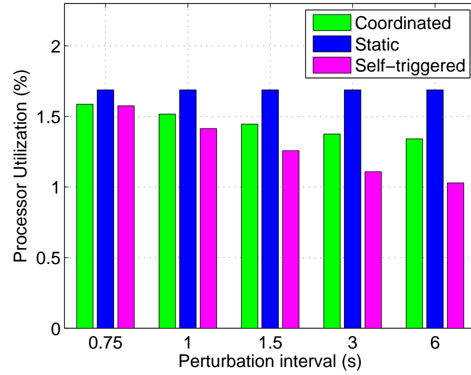

Fig. 9. Processor usage histogram for the three policies and all the perturbation intervals

5.2.2. *Resource utilization analysis*

Figure 9 shows the measured total processor usage of all tasks for the coordinated, self-triggered and static policies, for different perturbation intervals, during execution runs of 150s. The first conclusion that can be extracted is that both the coordinated and the self-triggered slack redistribution approaches always require less resources than the static policy. As the perturbation intervals increase, both policies consume less processor time. This is due to the fact that during more time intervals all the controlled plants are in equilibrium, and therefore the execution frequency of the controllers is set to the maximum period which is longer than the one used by the static controllers.

5.2.3. *Summary of the experimental results*

Figure 10 gives a complementary view of the experimental results. It shows the previous control performance and processor usage analysis for the co-

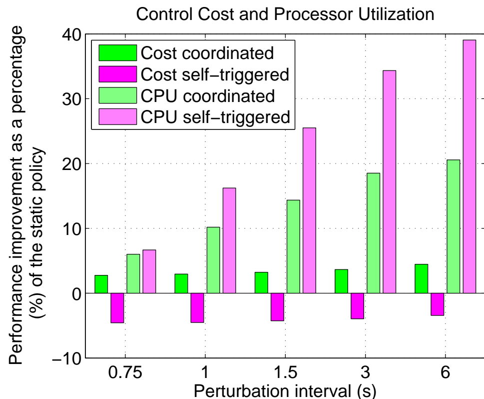

Fig. 10. Control performance and processor usage improvement as a percentage (%) of the static policy

ordinated and self-triggered policies with respect to the static, for different perturbation intervals, during execution runs of 150s. In terms of control performance, bars above zero mean that the control performance has been improved while bars below zero means that control performance degradation occurs, always with respect to the static policy. In terms of processor usage, bars above zero mean that processor time has been saved.

Looking at the relative improvements (or degradations), the first conclusion that can be drawn is that for this experiment, the processor usage improvements are more noticeable than the control performance improvements.

Second, the coordinated slack redistribution policy is able to improve both control performance and consumed processor time. Although difficult to appreciate, as the perturbation interval increases, the relative control performance improvement increases. This is because the perturbations are less overlapped, and the coordinated slack redistribution policy can perform its job more effectively.

Third, although the self-triggered slack redistribution policy always performs worse than the static in terms of control performance, it is the best for reducing processor usage. In addition, as the perturbation interval increases, the self-triggered reduces the control performance degradation, but more important, it is able to save, in percentage, more resources than the optimal, about two times more.

5.2.4. *Overhead analysis*

The processor utilization analysis presented in Section 5.2.2 implicitly incorporate the overhead analysis introduced by each slack redistribution policy. That is, in Figure 9, the processor time spent in the the coordinated policy includes the execution of the three control tasks as well as the execution of the feedback scheduler task. The later is the one in charge of performing the slack redistribution. Similarly, the processor time spent for the self triggered policy only includes the execution of the three control tasks. But they are the ones in charge of performing the slack redistribution.

For the execution of the three policies, at the kernel level, no specific tasks have to be performed apart from the standard dispatching of tasks according to EDF. It is worth noting than since for long perturbation intervals the two slack redistribution policies execute in average controllers with longer periods than the static, fewer context switches will occur. However, for short perturbation intervals, this property does not hold.

In addition, the sampling period settings shown in Table 3 and the time measures shown in Table 2 also provide some measures of overhead and determine control performance. First of all, the overhead of the coordinated policy is lower than the overhead of the self-triggered approach. Looking at Table 2, in the worst case scenario, during 50ms, the coordinated policy uses 58μs for the slack redistribution (two executions of the feedback scheduling task) while the self-triggered uses 276μs (time spent for the four controllers in slack redistribution operations).

Second, the overhead has a direct influence on control performance. For example, if the code of the feedback scheduler task would have been more complex, for a given $h_{i,max}^{coord}$, the coordinated utilization would have been higher, and then the h_i^{static} would have been shorter. In other words, the control performance improvement of the coordinated with respect the static would have been lower. Alternatively, by looking at the self-triggered policy, if the computation of the next sampling period, $\Upsilon(\cdot)$, would have used a simpler expression different than the exponential operation (remember (12)), the resulting periods for the self-triggered controllers, taking into account the static one, would have been shorter. In other words, the self-triggered policy could have achieved better control performance numbers.

5.3. *Key points*

In all the previous results, the criticalness parameter in the self-triggered policy was $K = 4$. Figure 11 shows for different values of K the relative control

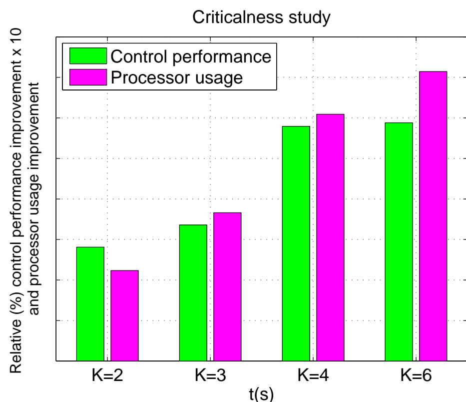

Fig. 11. Criticalness (K) study: control performance improvement and processor time savings relative to the case $K = 1$

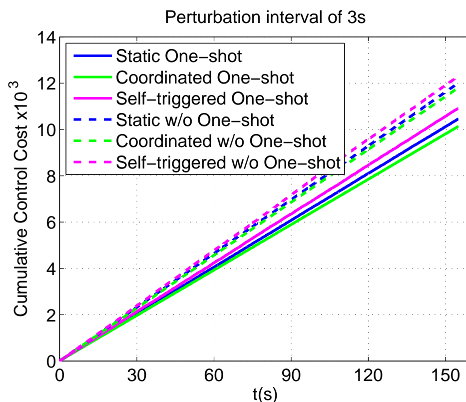

Fig. 12. Jitter evaluation

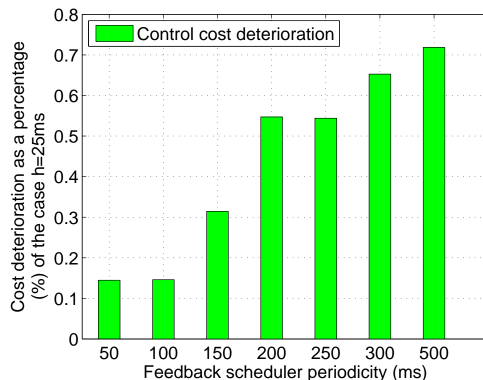

Fig. 13. Deterioration of the cumulative control cost as a function of the period of the feedback scheduler task.

performance improvement as well as the processor time savings relative to the case of $K = 1$. Note that in the figure, due to different orders of magnitude, the control cost bars has been multiplied by 10. This means that in percentage, the processor time savings are much more important than the cost improvements when K varies.

The set of evaluated values for K depend on the transient dynamics of the plant response. Small values of K produce slow changes in the sampling periods, that is, non-aggressive slack redistribution. If the sampling period changes take more time than the plant transient, they will not affect on control performance. On the contrary, higher values of K specify faster period changes. If the plant transient is short, higher values for K will provide in general better results. At it can be observed in the figure, more aggressive slack redistribution leads to better control performance (higher bars) but greater processor consumption (higher bars). That is, shorter periods during more time will provide better control performance but also increase the processor utilization. At some point, from $K = 6$, the performance achieved does not improve enough compare to the increase in processor time.

Theoretically, for higher values of K, the self-triggered approach tends to behave like the coordinated policy. The coordinated allocates the available slack in one execution (aggressively), while the self-triggered allocates the same available slack in few job executions. As a consequence, the higher the criticalness parameter, the faster will be the allocation of slack in the self-triggered. This also explains why the previous performance analysis has been performed with a high value for the criticalness parameter for the self-triggered policy. Since the coordinated is aggressive, the self-triggered has been also set to be aggressive.

In all the previous experiments, all the control tasks were implementing the one shot task model for avoiding the degradation problems caused by scheduling jitters. Figure 12 shows, for a given perturbation interval of 3s, the cumulative cost of the three policies when control tasks execute implementing or not the one-shot task model. For a controlled jitter of 5ms, the one-shot reduces the control cost for about 15% for all policies. The application of the one-shot task model ensures that the achieved control performance will be the same regardless of the tasks' deadlines [32].

The self-triggered policy is able to save more resources than the coordinated. An strategy for saving more resources in the coordinated policy is to execute the feedback scheduling task less frequently. This will influence control performance. The effect of the periodicity of the feedback scheduler with re-

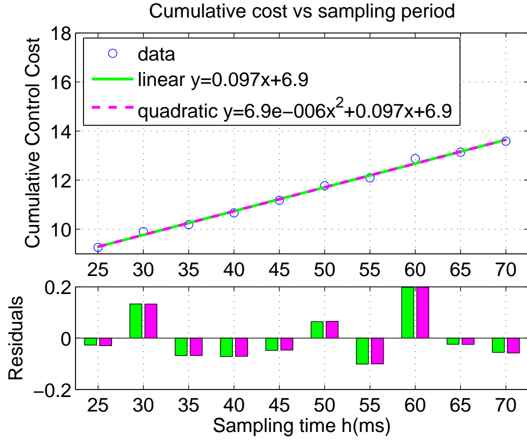

Fig. 14. Statistical analysis of the linear performance benefit.

spect to control performance is shown in Figure 13, per control performance degradation is with respect to the case of a feedback scheduler with a period of 25ms. As it can be seen, as the period of the feedback scheduler increases, the control performance degradation also increases, conclusion that was already drawn in a simulation study of other feedback scheduling approach [7].

Finally, it is important to note that for the presented evaluation, the assumption of linear benefit functions p_i for the coordinated approach is correct and no significant differences in performance would have been obtained by using for example quadratic benefit functions. This conclusion can be drawn from the following data fitting analysis. In Figure 14 we show in the top sub-figure the experimental numbers of control performance measured using (17) of a standard controller as a function of some of the sampling period choices (from $h = 25$ms to 70ms in steps of 5ms) considered in all the policies (represented by small circles). These numbers have been fit by a linear and a quadratic polynomial. The resulting fitting curves have been also plotted in the top sub-figure. Note that both curves overlap, meaning that both fittings are good choices. In fact, in the fit equation for the quadratic approximation, the second order term can be considered negligible. For completeness of the analysis, the bottom sub-figure shows the residuals analysis as a bar plot. Obviously, they are almost equal. Therefore, for the range of sampling periods and considering the controlled plants, the use of linear benefit function is appropriated.

6. Discussion

The lesson learned from the experimental evaluation of the coordinated and self-triggered approach to slack redistribution in real-time control systems can be summarized as follows:

- Both approaches require a tight collaboration between control tasks and real-time kernel. Slack redistribution is always based on two decision variables, controlled plant states and available slack. Since the first variable belongs to the applications space, and the second variable belongs to the kernel space, passing mechanisms have to be provided, such as specific system calls or shared memory. This demands a flexible real-time system: a reflective architecture has been shown to be the key for successfully implementing both approaches.
- The implementation details of both approaches show that the coordinated approach could demand more modifications in the kernel if the calculations of the slack redistribution were implemented in the kernel. However, this is not the case when the computations are performed at the user level by a feedback scheduler task. In this case, the coordinated and the self-triggered approach perform the slack redistribution at the task level, thus demanding small support at the kernel.
- In terms of control performance, the coordinated gives better results. This was already expected in the sense that it implements an optimal policy to slack redistribution while the self-triggered implements an heuristic policy. However, the theory could have failed in the implementation due to the overhead of the slack redistribution. But the paper has shown that even taking into account this overhead, the coordinated still performs the best.
- In terms of resource utilization, both approaches are capable of saving processor time, savings that increase when perturbations occur infrequently. Therefore, compared to the traditional approach to real-time control systems implementation, resources are not wasted, they are reclaimed and used when they are needed. In addition, the self-triggered has the potential of saving more processor time. Therefore, for highly resource-constrained systems, the self-triggered approach can be tailored so that resource utilization is minimized while control performance is still acceptable, fulfilling less strict control specifications. Hence, the self-triggered approach appears

to be a good candidate for slack redistribution in networked control systems.

- In terms of overhead, it has been shown from the implementation that the introduced overhead of the slack redistribution computations does not impair achieving good control performance and/or minimizing processor time. However, the overhead introduced by the self-triggered policy is higher than the one introduced by the coordinated. And this overhead prevents the self-triggered policy to achieve better control performance numbers. Therefore, lighter methods for computing the next sampling interval are required.

7. Conclusions

An experimental evaluation of two policies to slack management for real-time control systems has been presented. Since redistributing unused resources alters the rate of progress of each control task, both policies demand support from a flexible real-time operating system to permit changing tasks' rates. In addition, a reflective architecture has been identified as a suitable solution to permit changing tasks periods as a function of two decision variables: the available slack and the state of the plants.

The implementation of both policies shows which mechanisms are used between tasks and kernel to exchange the reflective information. The main computations for the slack redistribution are performed in the user space. In terms of performance, the coordinated approach provides higher benefits, as expected. And in terms of resource utilization, both policies are shown to be capable of saving resources. Specifically, the self-triggered approach can be tuned to save more resources if the computing platform is severely resource-constrained. This suggests that for such systems, event-based executions can be a solid approach to minimize resource consumption. Future work will focus on further analysis of this latest observation.

References

[1] Buttazzo, G., 2006. Research trends in real-time computing for embedded systems. SIGBED Rev. 3, 3 July, 1-10.

[2] Årzén, K.-E., Cervin, A., 2005. Control and Embedded Computing: Survey of Research Directions", 16th IFAC World Congress, July 2005.

[3] Årzén, K.-E., Cervin, A., Eker, J., Sha, L., 2000 An introduction to control and scheduling co-design. 39th IEEE Conference on Decision and Control, 2000.

[4] Cervin, A., Eker, J., Bernhardsson, B., Årzén, K.-E., 2002. Feedback-feedforward scheduling of control tasks. Real-Time Systems, 23:25–53.

[5] Velasco, M., Martí, P., Fuertes, J.M., 2003. The self-triggered task model for real-time control systems. WIP session of 24th IEEE Real-Time Systems Symposium.

[6] Martí, P., Lin, C., Brandt, S., Velasco, M., Fuertes, J.M., 2004. Optimal State Feedback Based Resource Allocation for Resource-Constrained Control Tasks, 25th IEEE Real-Time Systems Symposium, December.

[7] Henriksson, D., Cervin, A., 2005. Optimal On-line Sampling Period Assignment for Real-Time Control Tasks Based on Plant State Information. 44th IEEE Conference on Decision and Control and European Control Conference ECC05

[8] Ben Gaid, M., Cela, A., Hamam, Y., Ionete, C., 2006. Optimal Scheduling of Control Tasks with State Feedback Resource Allocation, 2006 American Control Conference, June 2006.

[9] Castañé, R., Martí, P., Velasco, M., and Cervin, A., 2006. Resource Management for Control Tasks Based on the Transient Dynamics of Closed-Loop Systems. 18th Euromicro Conference on Real-Time Systems, July.

[10] Tabuada, P., Wang, X., 2006. Preliminary results on state-triggered scheduling of stabilizing control tasks. 45th IEEE Conference on Decision and Control.

[11] Lemmon, M., Chantem, T., Hu, X., Zyskowski, M., 2007. On Self-Triggered Full Information H-infinity Controllers. Hybrid Systems: Computation and Control, April.

[12] Henningsson, T., Johannesson, E., and Cervin, A., 2008. Sporadic Event-Based Control of First-Order Linear Stochastic Systems. Automatica, 44(11), pp. 2890–2895, November.

[13] Anta, A., Tabuada, P., 2008. Space-Time Scaling Laws for Self-triggered Control. 47th Conference on Decision and Control, Dec.

[14] Martí, P., Lin, C., Brandt, S., Velasco, M., Fuertes, J.M., 2009. Draco: Efficient Resource Management for Resource-Constrained Control Tasks. IEEE Transactions on Computers, Vol. 58, N. 1, Jan.

[15] Wang, X., Lemmon, M., 2009, Self-Triggered Feedback Control Systems with Finite-Gain $\mathcal{L}_2$ Stability. IEEE Transactions on Automatic Control, Vol. 53, N. 3, pp 452-467, March.

[16] Lin, C., Brandt, S.A., 2005. Improving Soft Real-Time Performance Through Better Slack Reclaiming, 26th IEEE Real-Time Systems Symposium, pp. 3–14, December.

[17] Lozoya, C., Velasco, M., Martí, P., 2007. A 10-Year Taxonomy on Prior Work on Sampling Period Selection for Resource-Constrained Real-Time Control Systems. Work-in-progress Session, 19th Euromicro Conference on Real-Time Systems, Pisa, Italy, July.

[18] Buttazzo, G., 2005, "Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications", Second Edition, Springer.

[19] Åström, K.J., & Wittenmark, B., 1997. Computer–Controlled Systems. Third Edition. Prentice–Hall.

[20] Martí, P., Velasco, M., Bini, E., 2009. The Optimal Boundary and Regulator Design Problem for Event-Driven Controllers. 12th International Conference on Hybrid Systems: Computation and Control, San Francisco, CA, USA, April.

[21] Stankovic, J. A. 1996. Strategic directions in real-time and embedded systems. ACM Comput. Surv. 28, 4, Dec.

[22] Stankovic, J., Ramamritham, K., 1995. A Reflective Architecture for Real-Time Operating Systems. Chapter in Advances in Real-Time Systems, Prentice Hall, 23-38.

[23] Gai, P., Abeni, L., Giorgi, M., Buttazzo, G., 2001. A New Kernel Approach for Modular Real-Time Systems Development. 13th Euromicro Conference on Real-Time Systems.

[24] Terrasa, A., García-Fornes, A., J. Botti, V., 2002. Flexible Real-Time Linux: A Flexible Hard Real-Time Environment. Real-Time Systems, 22, 1-2 (Jan.), 151-173.

[25] Rivas, M. A. and Harbour, M. G., 2002. POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In Proceedings of the 14th Euromicro Conference on Real-Time Systems.

[26] Brandt, S. A., Banachowski, S., Lin, C., Bisson, T., 2003. Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes. 24th IEEE international Real-Time Systems Symposium.

[27] Erika Enterprise, Evidence Srl., `http://www.evidence.eu.com/content/view/27/254/`

[28] Marau, R., Leite, P., Velasco, M., Martí, P., Almeida, L., Pedreiras, P., Fuertes, J.M., 2008. Performing Flexible Control on Low Cost Microcontrollers using a Minimal Real-Time Kernel. IEEE Transactions on Industrial Informatics, Vol. 4, N. 2, pp. 125-133, May.

[29] Liu, C.L., & Layland, J.W., 1973. Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment. Journal of the ACM, 20(1), 40–61.

[30] Flex full base board, Evidence Srl, `http://www.evidence.eu.com/content/view/154/207/`

[31] Cervin, A., Eker, J. 2005. Control-Scheduling Codesign of Real-Time Systems: The Control Server Approach. Journal of Embedded Computing, vol.1, n.2, pp.209-224.

[32] Lozoya, C., Velasco, M., Martí, P., 2008. The one-shot task model for robust real-time embedded control systems. IEEE Transactions on Industrial Informatics, vol. 4, n. 3, pp 164–174, Aug.