**ADVERTIMENT**. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

**ADVERTENCIA**. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

**WARNING**. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Programa de Doctorat:

AUTOMÀTICA, ROBÒTICA I VISIÓ

Tesi Doctoral

# RESOURCE AND PERFORMANCE TRADE-OFFS IN REAL-TIME EMBEDDED CONTROL SYSTEMS

**Rafael Camilo Lozoya Gámez**

Directors:
Pau Martí Colom and Manel Velasco García

Abril de 2011

To Irlanda, Camilo and Betty

# Abstract

The use of computer controlled systems has increased dramatically in our daily life. Microprocessors are embedded in most of the daily-used devices, such as mobile phones, cars, dishwashers, etc. Due to cost constraints, many of these devices that run control applications are designed under processing power, space, weight, and energy constraints, i.e., with limited resources. Moreover, the embedded control systems market demands new capabilities to these devices or improvements in the existing ones without increasing the resource demands.

Enabling devices with real-time technology is a promising step toward achieving cost-effective embedded control systems. Recent results of real-time systems theory provide methods and policies for an efficient use of the computational resources. At the same time, control systems theory is starting to offer controllers with varying computational load. By combining both disciplines, it is theoretically feasible to design resource-constrained embedded control systems capable of trading-off control performance and resource utilization.

This thesis focuses on the practical feasibility of this new generation of embedded control systems. To this extend, two main issues are addressed: 1) the effective implementation of control loops using real-time technology and 2) the evaluation of resource/performance-aware policies that can be applied to a set of control loops that concurrently execute on a microprocessor.

A control task generally consists of three main activities: input data acquisition (sampling), control algorithm computation, and output signal transmission (actuation). The timing of the input and output actions is critical to the performance of the controller. The implementation of these operations can be conducted within the real-time task body or using hardware functions (or dedicated high priority tasks). The former introduces considerable amounts of jitters while the latter forces delays. This thesis shows that, by combining both approaches, control loops can be implemented in such a way that the problems caused by jitters and delays are removed.

The effective implementation of simple control algorithms does not guarantee the feasibility of implementing resource/performance-aware policies. Conversely to the initial problem targeted by these policies, that is, to minimize or keep resource requirements to meet the tight cost constraints related with mass production and strong industrial competition, research advances seem to require sophisticated procedures that may impair a cost-effective implementation. This thesis presents an evaluation framework that permits to assess the potential benefits offered by the theory as well as the pay-off in terms of complexity and overhead.

# Acknowledgments

This thesis would not have been possible without the support of many people.

First I like to express my gratitude to my thesis supervisors, Pau Martí and Manel Velasco. Thanks Pau, your encouragement, guidance and patience have been greatly valuable for me, personally and professionally. Thanks Manel, I deeply appreciate your support and knowledge that have been abundantly helpful for my thesis research. Working with both of you has been a wonderful experience.

I owe my deepest gratitude to Josep M. Fuertes for his leadership and guidance during my research activities, also for letting me be part of the DCS research group. I would like to show my gratitude to Vicenç Puig whose invaluable assistance help me to find a way in the Ph.D. program.

I would like to thanks my professors from the Automatic Control Department (ESAII) at the UPC, for broadening my horizons. I also thanks my fellow colleagues that support me in so many ways during the development of this work, specially to Antonio Camacho, Joseé Yépez and Julio Romero. Also thanks to Mercè Cabané for her always helpful assistance.

I like to acknowledge the CONACyT, Funcación Carolina and ITESM for their financial support during my Ph.D. studies. I would also convey my thanks to Gerardo Silveyra, Alberto Araujo, Joaquín Guerra and Rodolfo Castelló, for their trust in me and also for giving me the opportunity to become a member of the ITESM faculty.

Not forgetting to my friends Rene Luna and Julia Jasso whose hospitality make those days in Barcelona enjoyable. To Jorge Rodas who encourage me to continue studying.

Finally I wish to express my gratitude to my parents for their encourage and support. To my wife, my son and my daughter for their understanding, caring and endless love during this journey.

# Contents

**1 Introduction** **1**

1.1 Background . . . 1

1.2 State of the art . . . 3

1.3 Summary of contributions and thesis overview . . . 10

**2 One-shot task model** **13**

2.1 Introduction . . . 13

2.2 Problem set-up . . . 13

2.3 One-shot task model . . . 14

2.4 Simulation and experiments . . . 20

**3 One-shot task model extended to noisy measurements** **31**

3.1 Introduction . . . 31

3.2 Problem set-up . . . 31

3.3 One-shot and noisy measurements . . . 33

3.4 Simulation and experiments . . . 35

**4 Taxonomy on resource/performance-aware policies** **43**

4.1 Introduction . . . 43

4.2 Problem set-up . . . 43

4.3 Taxonomy . . . 48

4.4 Selected methods for performance evaluation . . . 52

**5 Performance evaluation framework** **55**
5.1 Introduction . . . 55
5.2 Problem set-up . . . 55
5.3 Evaluation framework . . . 59
5.4 Implementation and evaluation of selected methods . . . 70

**6 Performance evaluation: a detailed experience** **83**
6.1 Introduction . . . 83
6.2 Problem set-up . . . 83
6.3 A FBS and EDC implementation . . . 87
6.4 Results . . . 94

**7 Conclusions** **101**
7.1 One-shot task model . . . 101
7.2 Performance evaluation framework . . . 102
7.3 Future work and open problems . . . 102

**References** **103**

**A Continuous and discrete cost function** **111**

**B Framework simulation source code** **113**
B.1 Main program . . . 113
B.2 Initialization modules . . . 114
B.3 Controllers and optimization algorithms . . . 115

**C Framework experiment source code** **117**
C.1 File: `setup.c` . . . 117
C.2 File: `config.oil` . . . 121
C.3 File: `code.c` . . . 122

# List of Figures

1.1 Control timing demands. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.2 Hard real-time periodic task. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 Embedded control systems executing multiple control loops. . . . . . . . . . . . . . 10

2.1 Naif task model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.2 One-sample task model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3 Timing analysis of controllers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.4 Controller pseudo-code. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.5 One-shot task model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.6 One-shot task execution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.7 One-shot task vs. one-sample task. . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.8 Electronic double integrator circuit. . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.9 Simulation model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.10 EDF schedule timing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.11 Detailed view of the operation of several control strategies. . . . . . . . . . . . . . 23
2.12 One-shot performance evaluation in front of existing solutions (simulation). . . . . 24
2.13 Naif task pseudo-code. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.14 One-sample task pseudo-code. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.15 One-shot task pseudo-code. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.16 One-shot performance evaluation in front of existing solutions (experiment). . . . . 28

3.1 Kalman filter design approaches . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.2 Simulation model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.3 Simulation system response . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

3.4 Kalman filter implementation using the standard controller . . . . . . . . . . . . . 38

3.5 Kalman filter implementation using one-shot controller . . . . . . . . . . . . . . . . 39

3.6 Removing noise with the Kalman filter . . . . . . . . . . . . . . . . . . . . . . . . . 40

3.7 Controllers response with jitters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

3.8 Kalman gain evolution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

3.9 Accumulated execution time. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

5.1 Full Flex board with a dsPIC33 microcontroller. . . . . . . . . . . . . . . . . . . . 59

5.2 Framework functional modules. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

5.3 Framework multitasking single processor configuration. . . . . . . . . . . . . . . . . 60

5.4 Framework resource manager module. . . . . . . . . . . . . . . . . . . . . . . . . . 61

5.5 Framework task controller module. . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

5.6 Simulink/TrueTime model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

5.7 Processor kernel model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

5.8 Simulation framework flow diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

5.9 A microcontroller task controlling one double integrator plant. . . . . . . . . . . . 66

5.10 Main program pseudo-code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

5.11 Periodic controller pseudo-code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

5.12 On-line optimization pseudo-code . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

5.13 Event controller pseudo-code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

5.14 Electronic double integrator circuit . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

5.15 Model validation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

5.16 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

5.17 Off-line FBS plant response and activation times. . . . . . . . . . . . . . . . . . . . 75

5.18 On-line FBS-Inst. plant response and activation times. . . . . . . . . . . . . . . . . 76

5.19 On-line FBS-FH plant response and activation times. . . . . . . . . . . . . . . . . . 77

5.20 Heuristic self-triggered plant response and activation times. . . . . . . . . . . . . . 78

5.21 Self-triggered plant response and activation times. . . . . . . . . . . . . . . . . . . 79

5.22 Optimal self-triggered plant response and activation times. . . . . . . . . . . . . . . 80

6.1 Pseudo-code for a standard periodic control task . . . . . . . . . . . . . . . . . . . . 88
6.2 Pseudo-code for the two tasks coordinated approach . . . . . . . . . . . . . . . . . . 89
6.3 Pseudo-code for the self-triggered approach . . . . . . . . . . . . . . . . . . . . . . . 89
6.4 Experimental data for observer design . . . . . . . . . . . . . . . . . . . . . . . . . . 90
6.5 Cumulative cost for the three policies . . . . . . . . . . . . . . . . . . . . . . . . . . 92
6.6 Cumulative control cost histogram for the three policies and for all the perturbation intervals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
6.7 Processor usage histogram for the three policies and all the perturbation intervals . 94
6.8 Control performance and processor usage improvement as a percentage (%) of the static policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
6.9 Criticalness ($K$) study: control performance improvement and processor time savings relative to the case $K = 1$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
6.10 Jitter evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
6.11 Deterioration of the cumulative control cost as a function of the period of the feedback scheduler task. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
6.12 Statistical analysis of the linear performance benefit. . . . . . . . . . . . . . . . . . 99

# List of Tables

2.1 Task set parameters in milliseconds. . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.1 Simulation control performance. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.2 Experimental control performance. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

4.1 Taxonomy of resource management approaches. . . . . . . . . . . . . . . . . . . . . . 50

4.2 Selected methods of FBS and EDC showing key distinctive features. . . . . . . . . . 52

5.1 FBS evaluation parameters and platform . . . . . . . . . . . . . . . . . . . . . . . . . 56

5.2 EDC evaluation parameters and platform . . . . . . . . . . . . . . . . . . . . . . . . 57

5.3 Electronic components nominal values . . . . . . . . . . . . . . . . . . . . . . . . . . 69

5.4 Simulation tasks sampling periods (seconds) . . . . . . . . . . . . . . . . . . . . . . . 72

5.5 Control performance and resource utilization simulation results . . . . . . . . . . . 73

5.6 Experimental tasks sampling periods (seconds) . . . . . . . . . . . . . . . . . . . . . 74

5.7 Control performance and resource utilization experimental results . . . . . . . . . . 81

6.1 Worst case execution times . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

6.2 Sampling periods in the experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

# Chapter 1

# Introduction

## 1.1 Background

Embedded devices are being widely used in many areas, playing a key role in our society. An embedded system is a special-purpose computer system designed to perform dedicated control activities, interacting with the environment. Often the user of the device is not even aware that a computer is present. Embedded control systems are found on portable devices (e.g. digital watches, mobile phones, credit cards), daily-used machines (e.g. dishwashers, automobiles, domestic TV's), medical instruments (e.g. heart pacers, patient monitors), and large stationary installations (e.g. traffic lights controllers, factory automation systems).

Currently 98% of computing devices in the world are embedded systems. Most of mobile phones contains 5-10 processors and a typical car has 60 processors or more. Conservative estimations indicate that a total of 16 billion of embedded systems will be available by the end of the year 2010, and a forecast of over 40 billion of available devices worldwide by 2020. In the next five years, the share of the value of embedded electronics components in the value of the final product is expected to reach significant percentages (more than 40% in average) [ART10].

Embedded systems market demands devices with more and better functionalities at lower prices. In addition, embedded devices are designed under space, weight and energy constraints, imposed by cost restrictions. As a consequence, embedded applications typically run on small processing units with limited memory and computational power. This increases embedded control systems complexity from both the control and computer science perspectives. As a consequence researchers in the computer and control fields are becoming increasingly aware of the need for an integrated scientific and technological perspective on the role that computers play in control systems and that control can play in computer systems [SÅ03].

However, by tradition, the design of embedded control systems has been based on the separation-of-concerns principle [ÅC05]. This principle is based on the fact that the most common analysis, design and implementation approach for embedded control systems is the periodic execution of control algorithms. This periodic data abstraction is advantageous from the design standpoint. The control community has focused on the pure control design without having to worry about which resource needs the controller is placing on the execution platform and how the control system eventually is implemented. The computer science community has focused on development of computational methods and models, without any need to understand what impact the final implementation has on the stability and performance of the plants under control.

While this so-called separation-of-concerns has proven advantageous from a designer's perspective, it does not necessarily lead to cost-effective implementations. By separating the concerns of

the control engineer from the computer science engineer, each designer is forced to adopt a conservative viewpoint that may lead to unnecessary over-provisioning in the system implementation and hence to higher system costs and sub-optimal control performance.

To overcome this limitation, there has been a recent interest in developing co-design frameworks where the concerns of computer science and control systems engineers are treated in a unified manner. In particular, the real-time systems community and the control community have been dealing with the co-design problem in the last years [SAÅ+04]. One of the first statements of the co-design problem was given in [SLSS96]. Since that time, a number of other co-design approaches have been suggested. A list of such methods can be found in the introduction to control and scheduling co-design given in [ÅCES00].

Hence, enabling embedded devices with real-time technology is a promising step toward achieving cost-effective embedded control systems. In fact, nowadays simple embedded control systems often contain a multi-tasking real-time kernel [CHL+03]. For example, controllers are often implemented as one or several periodic tasks. Often the microprocessor also contains tasks for other functions (e.g., communication and user interfaces). Therefore, the kernel uses multiprogramming to multiplex the execution of the various tasks. In this case, the central process unit (CPU) time can be viewed as shared resource for which the tasks compete.

However, the simple approach based on the assumption that controllers can be modelled and implemented as periodic real-time tasks with fixed periods and scheduled with standard scheduling policies such as fixed priority (FP) or earliest deadline first (EDF) [LL73] does not guarantee efficient resource usage and outstanding control performance. First, the selection of fixed rates of execution is not an easy task: low rates imply low resource utilization but also imply low control performance (and viceversa). Second, embedded control systems usually operate in dynamic environments where application demands, computational workload and resource availability experience changes during execution time. Therefore, the enforcement of a fixed rate can be inappropriate.

Overcoming these limitations demands flexible and adaptable scheduling policies and controller designs capable to make an efficient use of the computational resources [But06]. It is desirable to achieve more dynamic system architectures where the control applications and the implementation platform negotiate on-line for access to shared resources, such as CPU time. To this extend, two new trends for the analysis and design of embedded control systems can be identified in the literature [LVM07]. The first one, often referred as "*feedback scheduling*" (FBS), is based on applying efficient sampling period selection techniques that account for processor load and plants dynamics in such a way that the aggregated control performance delivered by the set of control loops is improved. The second trend is based on applying feedback "*event-driven control*" (EDC) techniques in order to minimize controllers resource demands while still guaranteeing stability and acceptable control performance.

These resource/performance-aware approaches focus on theory, whereas practical aspects are often omitted. Moreover, these theoretical advances demand flexible real-time kernel support as well as more complex controller mechanism and designs, thus requiring *a priori* more sophisticated and expensive software/hardware solutions. However, as argued before, new trends for embedded control systems demand low cost solutions.

This thesis provides insight into these conflicting demands which may impair the implementation feasibility of this new generation of embedded control systems. In particular, by focusing in the evaluation of the diverse resource/performance-aware policies that can be applied to a set of control loops that concurrently execute on a microprocessor, two main problems are identified:

- the effective implementation of control loops using real-time technology.
- the definition and development of an evaluation framework capable of including the wide variety of resource/performance-aware policies.

## 1.2 State of the art

The design of embedded control systems is essentially a co-design process, since decisions made in the real-time design affect the control design and viceversa. Recent research on embedded control systems has focused on two main domains. The first one is related to effective implementation techniques of control algorithms on real-time platforms, and the second one refers to the resource/performance-aware policies for multitasking control systems in order to maximize control performance and/or minimize resource utilization. In this document both domains are discussed. The state of the art presented below does not try to be an exhaustive one, but instead it provides representative tendencies on embedded control systems in order to raise important research challenges.

### 1.2.1 Implementation of control algorithms

The common approach to computer controlled systems design has two steps. The first step is to obtain a discrete-time model of the plant. The second step is to design a discrete-time control law for the discrete-time plant model. The design approach mandates to periodically sample and actuate.

The key aspect of the design procedure is that the discrete-time model of the plant describes the behavior of the analog plant at the sampling instants. Moreover, the actuation instants are defined in terms of the sampling instants. Therefore, the time reference and synchronism is given by the sampling instants (as illustrated in Figure 1.1). Once a sample is taken, the control signal is computed assuming that the next sample will occur after $h$ time units (i.e., one sampling period), and assuming that the actuation will occur after $\tau$ time units (i.e., one time delay). Both assumptions refer to future *known* events: next sampling instant or subsequent actuation instant.

After the controller design stage, the control law is implemented by means of a control algorithm. Although real-time computing is about meeting timing constraints [But97], it is not straightforward to meet the periodic control demands with available real-time technology. The hard real-time periodic task is the baseline computational abstraction for implementing control algorithms. In a periodic task, shown in Figure 1.2, consecutive releases times mark the task period, and jobs execute within each release time and relative deadline. The relative deadline can be assumed to be less or equal than the period.

The control task model identified in [ÅCES00] as the common practice implementation of control loops in real-time control systems is refereed as "*naif*" task model. The naif task model assumes control algorithms implemented as hard real-time periodic tasks, with task period equal to the sampling period. Sampling (input) and actuation (output) operations are specified to occur at the beginning and at the end of each job execution. The deadline specification is a key aspect in the naif task model. For one task executing in isolation, the timing of the control task execution corresponds to the expected timing (Figure 1.1) if the deadline is set equal to the time delay. However, in a multitasking system, this tight specification of the deadline impairs task set schedulability in the general case [MFVF01]. Relaxing this specification by setting the deadline greater than the time delay introduces sampling and latency jitters in control job executions. From a control-theoretic perspective, it is useful to distinguish between sampling jitter (variation in the input instant or sampling) and input/output jitter (variation in the delay from input or sampling to output or actuation). If jitters occur, the next sampling and/or actuation will be performed at times different than the expected ones. The synchronism given by the sampling instants is lost. Therefore, the introduced time uncertainty violates the mandated periodicity, and control performance degradation occurs [WNT95].

Different solutions have been proposed to the jitter problem, these can be roughly divided into control-based solutions and real-time based solutions.

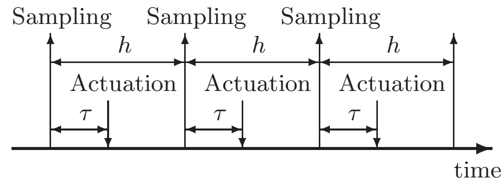



Figure 1.1: Control timing demands.

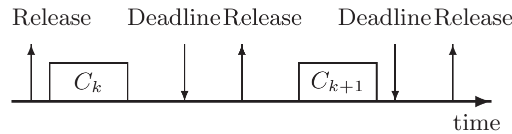



Figure 1.2: Hard real-time periodic task.

#### 1.2.1.1 Control-based solutions

The problem of dealing with scheduling-introduced jitters has been treated from a more pure control perspective. In these approaches, the common trend is to accept the presence of the jitters and modify control parameters in order to compensate the control actuation.

The jitter problem has been analyzed considering control systems with irregular sampling intervals. An initial solution is proposed by [AS90], where each discrete control action is evaluated according to the time interval since the last sample, allowing the controller parameters to be directly updated each time the sampling period changes. Later in [AC99], an advanced observer (predictor) is proposed based on the intrinsic properties of sampled data systems to reduce the effect of variable time-delays. However these control design techniques are not integrated within existing scheduling theory, and implementation details are not considered.

Other solutions consider the use of time-stamps and linear compensators to reduce the degrading effect of jitters. In [Lin02] a method to compensate for time-varying random delays in digital control systems is presented. The idea is to add a time-delay aware compensator to the original controller, to improve stability and performance of the control loop. In this model the actuator sends time-stamps to the controller, in this way the controller may use knowledge of when former control signals actually appeared at the plant. The compensator measures the difference between the actual control signal and the desired control signal from the controller, to modify the data sent to the actuator. In [Boj05] is shown that the effects of jitter can be modelled by approximations for the plant dynamics, and additive noise constructed by the modulation of plant signals with the jitter. These approximations give useful insights for digital controller design.

In [FSR04] a solution is proposed by solving the problem for a continuous-time system with uncertain but bounded time-varying delay in the control input. Linear matrix inequalities (LMI) conditions are derived for stabilization of systems, the derived conditions are conservative in order to guarantee stabilization for all sampling intervals not greater than a maximum sampling period. In a similar way, in [SB09] an approach to the analysis and synthesis of state-feedback controllers with timing jitter is presented where LMI methods are used to derive a Lyapunov function that establishes an upper bound on performance degradation due to the timing jitter. The LMI methods can be used to synthesize a constant state-feedback controller that minimizes the performance bound, for a given level of timing jitter. Other works have discussed the use of an input delay approach for non-uniform sampling [Mir07], in which the sample-and-hold circuit is embedded into an analog system with a time-varying input delay. However latency jitters are not covered by this approach.

#### 1.2.1.2 Real-time based solutions

Alternatives solutions to minimize the likelihood of jitters can be found in the real-time systems literature. A seminal work is presented by [LL73] where sampling and actuation are performed periodically by hardware interrupts at the release time and deadline respectively, introducing one-sample delay in all control loops closed over the computer. This type of task model is referred as "*one-sample*" task model. By using this model, jitters are removed but the model imposes an artificial long time delay in the closed-loop system, which introduces an unnecessary although predictable control performance degradation.

In order to reduce the long latencies inherent to the one-sample task model, a scheduling algorithm is proposed by [BBGL99] which reduces the latencies by assigning shorter relative deadlines. In the same way a computational model based on Giotto [HHK01] is proposed by [CE03], where the primary goal of the model is to facilitate co-design of real-time systems with control systems. In particular the model reduces input-output latencies by dividing a task into a number of segments, each segment is scheduled individually using dynamic periods which equals the current segment length. A preliminary work based on this approach can be found in [Cer01].

A similar work can be found in the definition of a control task model formalized by [BRVC04] based on the initial proposal by [ACR+00]. The objective of this approach is to reduce the delay variance in the delivering of the control action computed by the different tasks. For that purpose, each task is partitioned into an initial part (sensing or acquisition part), a mandatory part (computing or control algorithm part) and a final part to deliver the control action (actuation or delivering part). Each part is scheduled as independent, then by assigning different priorities levels and deadlines to each task part, the delay variance can be reduced and an average delay value can be incorporated into the controller design. This control task can be implemented using either FP or EDF scheduling policy. These real-time based solutions suffer from three problems. First, the occurrence of jitters is not completely eliminated. Second, artificial input/output (I/O) latencies are still enforced. And when latencies are shortened by specifying shorter relative deadlines, the third problem arises. System schedulability is reduced because jobs are forced to execute within shorter time intervals. This solution [BRVC04] is refereed as "*split*" task model in the rest of this document.

Finally, the model proposed by [MFRF01] is based on the notion of compensations wherein controller parameters are adjusted at runtime for the presence of jitters. This model is identified as the "*switching*" control task model. This approach uses control theory to calculate the adjusted controller parameters to compensate for the timing variations. Therefore the control law used in this task model requires to be calculated each time the task is executed since the sampling period may be constantly varying due to jitters. This model differs from the split model because it accepts the presence of jitter and compensate for it at runtime, instead of trying to reduce its variance. In the switching task model, parameter adjustments are conducted on-line if run-time overheads are acceptable, if not an alternative implementation is also proposed via table look-ups at runtime. In any case, additional computation or additional memory are required for implementation. The main drawback of this approach is the computation overhead or additional memory requirements due the switching of the controller gains, as well as possible chattering problems that may occur.

#### 1.2.1.3 Thesis approach

From a design point of view, there is a fundamental trade-off between delay and jitter. Performing input and output action in the real-time task body produces jitter in the general case. Using hardware functions or dedicated, high-priority input and output tasks, it is possible to exchange jitter for delay. But both delay and jitter degrade the control performance, and it cannot be said which is worse in general.

The approach considered in this thesis combines input performed in the control task body and actuation performed with hardware functions. The control algorithm using this implementation technique is based on prediction techniques [AC99]. Integrating both techniques, a novel control task model named "*one-shot*" is developed. It is shown that it permits to remove the endemic problems for embedded control systems that jitters and delays represent.

### 1.2.2 Feasibility of resource/performance-aware policies

The widespread use of embedded systems and their design challenges due to resource, cost and timing constraints has triggered novel research on control and real-time strategies in the analysis and design of embedded control systems with constrained resources. Two emerging disciplines, identified as feedback scheduling and event-driven control systems are surveyed next. In addition, the implementation feasibility study of these policies demands adequate simulation an experimentation tools.

A brief review of some representative simulation tools and real-time platforms used for research purposes in the embedded control systems area is also presented. In order to limit the scope of this review, the focus has been placed on recent tools and platforms suitable for real-time and control analysis and design.

#### 1.2.2.1 Resource/performance-aware policies

Feedback scheduling refers to the problem of sampling period selection for real-time control tasks that compete for limited computer resources such as processor time. Its goal is to optimize the aggregated control performance achieved by all tasks by using efficiently the scarce resources. The standard FBS architecture includes a resource manager element which dictates how task periods are assigned to each control task in order to optimize the overall control performance. The allocation of resources is commonly formulated as a constrained optimization problem where the objective function relates control performance and resource utilization, the later usually in terms of the sampling periods (task periods) or frequencies. The optimization variables usually are the set of sampling periods to be assigned to all control tasks. The optimization problem is constrained by two key aspects. The set of optimal sampling periods must guarantee closed loop stability and task set schedulability. Stability is either guaranteed by the formulation of the optimization problem, or it is not explicitly imposed in the formulation but analyzed after solving the optimization problem. Task set schedulability is often imposed by resource utilization tests. A few methods, instead of providing optimal sampling periods, provide job sequences. That is, the outcome of the optimization problem is an optimal sequence of jobs for each control task to be executed periodically. Examples of feedback scheduling results found in the literature are [SLSS96], [SLS98], [ZZ99], [EHÅ00], [RS00], [HCAÅ02], [PPSV+02], [CEBÅ02], [CLS03], [MLB+04], [PPBSV05], [HC05], [CMV+06], [GCHI06], [MLB+09], [BC08], [SCEP09], [SEPC09], [GcH09], or [CVMC10]. The large amount of contributions indicates that feedback scheduling is a theoretically mature discipline for processor-based systems. The main drawback of these approaches, identified in several of the listed papers, refers to whether the solution of the optimization problem is feasible at run-time, since implementability is not demonstrated.

In event-driven control systems, event conditions determine the occurrence of discrete events that trigger control updates. The event condition is often imposed in the problem formulation to restrict the desired system dynamics, but it can also be intrinsic to the nature of the control setup, such as the measurement method. The execution of event-driven controllers aims at minimizing resource utilization while ensuring stability or bounding the inter-sampling dynamics. This is achieved by executing controllers without periodic requirements: controllers jobs are only executed when needed. Event-driven controllers adapt the real-time system task period directly

in response to the application performance [Årz99]. In this way the real-time system is only used when it is essential for the system performance. Since the system state is always changing, this approach generates an aperiodic sequence of control task invocations. In general, the hope is that the average rate of this aperiodic tasks will be much lower than the rate of a comparable periodic task. There is, in fact, ample experimental evidence to support the assertion that event-triggered feedback improves overall control system performance while reducing the real-time system use of computational resources [AB02] [HSB08]. Examples of results of event-driven control systems in processor-based platforms are [Årz99], [HGvZ+99], [AB02], [VMF03], [TW06], [Mis06], [Tab07], [LCH+07], [JHC07], [SNR07], [AT08a], [AT08b], [WL08a], [WL08b], [HSB08], [HJC08], [WL09a], [WL09b], [MVB09], [MAT09], [MT09], [VMB09a], [AT09], [AT10], Unfortunately, although work on event-driven control started to appear in the 50's [Ell59], the discipline still lacks a mature system theory. Moreover, rarely implementation issues are discussed.

#### 1.2.2.2 Simulation tools

The real-time research community has developed a number of prototypical tools for schedule simulation, timing analysis and schedule generation, such as STRESS [ABRW94] and DRTSS [SL96]. Meanwhile, the control community have used mathematical software for simulations such as Matlab/Simulink[1] created by MathWorks since 1984 or Scilab/Scicos[2] created by the INRIA (Institut National de Recherche en Informatique et Automatique) and the ENPC (Ècole Nationale des Ponts et Chaussées) since 1990.

Recently the following computation tools have been developed for research purposes for the co-analysis and co-simulation of embedded control systems.

Ptolemy II [HLL+03] is the third generation of software produced within the Ptolemy project at the University of California at Berkeley. Ptolemy II supports heterogeneous, hierarchical modelling, simulation, and design of concurrent systems, especially embedded systems. The focus is on complex systems mixing various technologies and operations. Ptolemy is component-based and models are constructed by connecting a set of components and have them interact under the model of computation. In Ptolemy the real-time control system simulation is just one part of a larger framework.

The Aida toolset [REKT04] integrates the design and performance analysis of control systems with embedded real-time system design. The toolset enables specification and analysis of real-time implementations of control applications. Control system designs are imported to a real-time system-modelling domain in which the functionality is distributed on a target computer system. Once the real-time design is complete, the response times and release jitter of the processes and their contained functions can be analyzed and the system information exported back to the control domain. Matlab/Simulink can be used in the control domain, since Aida includes an interface with Matlab/Simulink. Aida focuses more in the model-based design rather than in the co-simulation analysis.

RTSIM [PLLA02] is a tool that is aimed at simulating realtime embedded control systems. The main goal is to facilitate cosimulation of real-time controllers and controlled plants in order to evaluate the timing properties of the architecture in terms of control performance. The tool consists of a collection of libraries which allows the user to specify a set of plants, the functional controller behavior, the implementation architecture, and a mapping of functional behavior onto the architectural components. The simulation produces results related both to the realtime performance and the control performance. This includes the generation of execution traces, realtime statistics (delays and jitter), and control performance metrics such as time responses and quadratic costs.

[1] Mathlab/Simulink, `http://www.mathworks.com/`
[2] Scilab/Scicos, `http://www.scilab.org/`

Torsche [SKSH06] is a Matlab-based toolbox including scheduling algorithms, that are used for various applications such as high level synthesis of parallel algorithms or response time analysis of applications running under a fixed-priority operating system. Using the toolbox, one can obtain an optimal code of computing intensive control applications running on specific hardware architectures. The tool can also be used to investigate application performance prior to its implementation. These values can be used in the control system design process performed in Matlab/Simulink.

TrueTime [HCÅ02] is a MATLAB/Simulink-based toolbox that facilitates simulation of the temporal behavior of a multitasking realtime kernel executing control tasks. The tasks are controlling processes that are modelled as ordinary continuous time Simulink blocks. TrueTime also makes it possible to simulate models of standard medium access control (MAC) layer network protocols, and their influence on networked control loops. TrueTime allows the execution time of tasks and the transmission times of messages to be modelled as constant, random, or datadependent. Furthermore, TrueTime allows simulation of context switching and task synchronization using events or monitors.

Recent surveys on simulation tools for real-time and control systems co-design can be found in [Årz05] and [THÅ+06].

#### 1.2.2.3 Real-time platforms

Many of the resource/performance-aware policies require 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 that no deadline is missed during the adjustment.

Therefore, the required real-time system support for implementing these 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 [Sta96].

Nowadays, there are more than a hundred commercial products that can be categorized as real-time operating systems, from very small kernels to large multipurpose systems for complex real-time applications [But97]. The most important commercial products and suppliers for real-time operating systems are: VxWorks (Wind River), OSE (OSE Systems), Windows CE (Microsoft), QNX (Neutrino), Integrity (Green Hills), RTLinux (University of New Mexico) and Linux/RK (TimeSys). However, these kernels are based on fixed priority scheduling, hence only rate monotonic (RM) scheduling and its derivatives can easily be implemented.

The rest of this section only focuses on recent real-time kernels developed for research purposes, since this generation of new operating systems include flexible features that allow the analysis and implementation of novel FBS and EDC algorithms. These features include the ability to treat tasks with explicit timing constraints, such periods and deadlines, and the possibility to characterize tasks with additional parameters, which are used to analyze the dynamic performance of the system. Moreover, they have to provide 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. Alternatively, the flexibility can be achieved when kernel structures contain application data, which can be then used by the kernel to alter the progress of each task. Examples of kernels providing this services include the Shark kernel [GAGB01], Marte OS [ARGH01], PaRTiKle [PMRC07] and Erika [Srl08a].

SHARK (Soft and Hard Real-time Kernel) [GAGB01] is a dynamic configurable research kernel architecture designed for supporting a simple implementation, integration and comparison of

scheduling algorithms. The kernel supports the development and testing of new scheduling algorithms, aperiodic servers and resource management protocols. SHARK is based on a generic kernel, which does not implement any particular scheduling algorithm neither a resource manager policy; the generic kernel provides the primitives to allow external modules to implement specific scheduling and resource manager algorithms. This kernel has been used for academic purposes and it is compliant with the POSlX (Portable Operating System Interface) 1003.13 PSE52 specifications [ISIEIS96].

MARTE (Minimal Real-Time OS for Embedded Applications) [ARGH01] is a real-time kernel for embedded applications that follows the Minimal Real-Time POSIX.13 subset [10096], providing both C and Ada language POSIX interfaces. The kernel has a low-level abstract interface for accessing the hardware that encapsulates operations for interrupt management, clock and timer management, and thread context switches. The applications planned for this kernel are industrial embedded systems, such as data acquisition systems and robot controllers, the targeted applications are mostly static, with the number of threads and system resources well known at compile time. This kernel has been mainly implemented on x86 platforms (Intel) and also on the MC68332 microcontroller (Freescale).

PaRTiKle [PMRC07] is a embedded real time aperating system designed to be POSIX compliant. PaRTiKle has been designed to support applications with real-time requirements, providing features such as full preemptability, minimal interrupt latencies, and all the necessary synchronization primitives, scheduling policies, and interrupt handling mechanisms needed for this type of applications. PaRTiKle supports Ada, C++ and Java applications. The PaRTiKle kernel has been designed and implemented as a set of hardware-independent subsystems and a set of drivers. So far, PaRTiKle supports execution environments based on the x86 platform, and it has recently been ported to LPC2000 microcontrollers (NXP Semiconductors).

ERIKA [CMC+04] is a small size, but fully functional kernel distributed by Evidence s.r.l.[3] and supports many features from the OSEK/VDX (Open Systems and the Corresponding Interfaces for Automotive Electronics / Vehicle Distributed eXecutive) standard [OSE]. ERIKA has been designed to be an effective educational and research platform for real-time programming in embedded systems. The kernel architecture consists of two main layers: the kernel layer and the hardware abstraction layer. The first layer contains a set of modules that implement task management and real-time scheduling policies. The hardware abstraction layer contains the hardware dependent code that manages context switches and interrupt handling. ERIKA currently supports Microchip dsPIC33 microcontrollers and Altera NIOS II processors.

Platforms and real-time kernels for testing embedded control systems can also be found in educational papers such as [ÅBW05] and [MVF+10]

#### 1.2.2.4 Thesis approach

Despite of the great variety of different resource/performance-aware policies that have recently proposed in the literature, no unified framework exist to assess their pros and cons, as well as, their implementation feasibility. Partial evaluations can be found in the literature. An evaluation of a feedback scheduling policy using a real-time kernel can be found in [MLV+08]. A first attempt to compare feedback scheduling methods can be found in [CA06], and a first attempt to analyze resource demands of a class of event-driven control systems is found in [VML08] and [VMB09b]. Schedulability issues of event-driven controllers were initially analyzed in [VMB08]. The work presented in [VMF+10] provides the first comparison between a feedback scheduling policy and an event-driven multitasking control system. A common evaluation framework is required to provide adequate services in order to fulfill the specifications obtained after the analysis of the different

[3] Evidence s.r.l., http://www.evidence.eu.com/

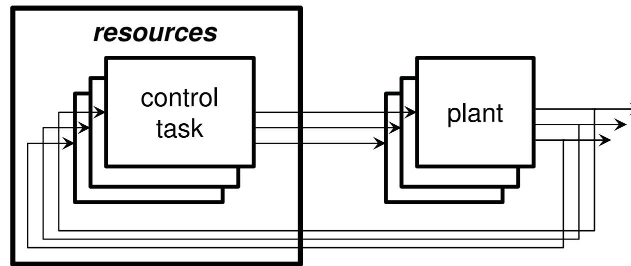



Figure 1.3: Embedded control systems executing multiple control loops.

existing methods. Furthermore the common framework must provide fair measurement metrics in order to evaluate control performance, resource utilization, and computation overhead.

This thesis presents an evaluation framework that permits to assess the potential benefits offered by each resource/performance-aware policy, as well as the pay-off in terms of complexity and overhead. This is achieved by classifying these policies into a taxonomy and defining the evaluation framework able to assess the diversity found in such policies. The framework includes a simulation part and an experimentation part, the first one is based on TrueTime while the later one is based on ERIKA.

## 1.3 Summary of contributions and thesis overview

The contribution of the thesis falls into the broad area of implementation and evaluation of embedded control systems. The embedded control system considered in this thesis is a micro-processor based system in which $n$-physical plants should be controlled by a computer with limited computational resources. The control is achieved by $n$-concurrent control tasks executing in the computer, each task being responsible for the sampling, control computation, and actuation in one loop. Only linear time-invariant systems are considered. The concurrency is facilitated by a real-time kernel enabled with EDF and FP scheduling policies. The overall situation is depicted in Figure 1.3.

In this context, the two main contributions of this thesis are:

1. Development of a novel theory and practice for effective implementation of control tasks running on top of a real-time kernel in such a way that endemic problems such as jitters and delays are eliminated, while easy integration in current scheduling policies is guaranteed.

2. Development of a common evaluation framework capable to assess 1) the implementation feasibility of key resource/performance-aware policies strategies for embedded control systems, and 2) their advantages and disadvantages in terms of control performance and resource utilization.

These contributions are described in this document as follows:

**Chapter 2** reviews existing solutions for effective implementation of control tasks, in terms of control tasks models. Then it presents the analysis, design and implementation of the one-shot task model as a novel task model for real-time control tasks. It is shown the feasibility and effectiveness of the proposed model compared with previous real-time and/or control based solutions already discussed in this chapter. The one-shot task model has already been introduced in the following papers:

[LMVF08] C. Lozoya, P. Martí, M. Velasco, and Josep M. Fuertes. Analysis and design of networked control loops with synchronization at the actuation instants. In *34th Annual Conference of the IEEE Industrial Electronics Society (IECON08)*, Orlando, Florida, US, November 2008.

[LVM08] C. Lozoya, M. Velasco, and P. Martí. The one-shot task model for robust real-time embedded control systems. *IEEE Transactions on Industrial Informatics*, 4(3), August 2008.

[MVF+07] P. Martí, M. Velasco, J.M. Fuertes, R. Villà, J. Yépez, and C. Lozoya. The one-shot task model for implementing real-time control tasks. In *II Congreso Español de Informática (CEDI2007)*, Zaragoza, Spain, Sep. 2007.

**Chapter 3** extends the one-shot task model to the case of noisy measurements. This is achieved by using Kalman techniques, which must take into account the non-periodicity of the sampling operations. Experimental results illustrate the effectiveness of the proposed solutions. Part of the results from this chapter has been published on:

[LRM+10] C. Lozoya, J. Romero, P. Martí, M. Velasco, and J. M. Fuertes. Embedding Kalman techniques in the one-shot task model when non-uniform samples are corrupted by noise. In *18th Mediterranean Conference on Control and Automation (MED2010)*, Marrakech, Morocco, June 2010.

Although not used with the one-shot task model, Kalman filter has been applied to estimate delays in a wireless network on:

[LMVF10] C. Lozoya, P. Martí, M. Velasco, and J. M. Fuertes. Study of a remote wireless path tracking control with delay estimation for an autonomous guided vehicle. *The International Journal of Advanced Manufacturing Technology*, pages 1–11, 2010. 10.1007/s00170-010-2736-x.

**Chapter 4** analyzes the characteristics of the resource/performance-aware polices used in the implementation of embedded control systems. Based the analysis, main features and current trends are identified, and a taxonomy is provided. Initial analysis on this topic have already been presented:

[LVM07] C. Lozoya, M. Velasco, and P. Martí. A 10-year taxonomy on prior work on sampling period selection for resource-constrained real-time control systems. In *Work in Progress 19th Euromicro Conference on Real-Time Systems (ECRTS 07)*, Pisa, Italy, July 2007.

[VML08] M. Velasco, P. Martí, and C. Lozoya. On the timing of discrete events in event-driven control systems. In *11th International Conference on Hybrid Systems: Computation and Control (HSCC08)*, St. Louis, MO, USA, April 2008.

**Chapter 5** analyzes how existing resource/performance-aware polices have been evaluated. It then describes the characteristics of the proposed performance evaluation framework. It also presents the application of the framework at evaluating different but representative resource management strategies. Initial performance/resource evaluation on different optimization policies were presented in the following publications:

[CMV+10] A. Camacho, P. Martí, M. Velasco, C. Lozoya, R. Villà, J. M. Fuertes, and E. Griful. Self-triggered networked control systems: an experimental case study. In *IEEE 2010 International Conference on Industrial Technology (ICIT2010)*, Valparaiso, Chile, March 2010.

[LMF10] C. Lozoya, P. Martí, and J. M. Fuertes. Minimizing control cost in resource-constrained control systems: from feedback scheduling to event-driven control. In *18th Mediterranean Conference on Control and Automation (MED2010)*, Marrakech, Morocco, June 2010.

[LMV06] C. Lozoya, P. Martí, and M. Velasco. Control performance evaluation of feedback scheduling of real-time control tasks. Technical Report ESAII-RR-07-16 Technical Report, Automatic Control Department, Technical University of Catalonia, Barcelona, Spain, 2006.

[LMVF08] C. Lozoya, P. Martí, M. Velasco, and J.M. Fuertes. Control performance evaluation of selected methods of feedback scheduling of real-time control tasks. In *17th World Congress of IFAC*, Seoul, Korea, July 2008.

[LMVF09] C. Lozoya, P. Martí, M. Velasco, and J.M. Fuertes. Simulation study on control performance and resource utilization for resource-constrained control systems. Technical Report ESAII-RR-09-01 Technical Report, Automatic Control Department, Technical University of Catalonia, Barcelona, Spain, 2009.

[YLMF09] J. Yépez, C. Lozoya, P. Martí, and J.M. Fuertes. Preliminary approach to Lyapunov sampling in CAN-based networked control systems. In *35th Annual Conference of the IEEE Industrial Electronics Society (IECON09)*, Porto, Portugal, November 2009.

**Chapter 6** presents a complete evaluation of two resource/performance-aware policies. The evaluation is carried out under the performance evaluation framework, but with a wider diversity of scenarios, which permit to extract new hidden conclusions. The analysis and results from this chapter have been published on:

[VMF+10] M. Velasco, P. Martí, J. M. Fuertes, C. Lozoya, and S. Brandt. Experimental evaluation of slack management in real-time control systems: Coordinated vs. self-triggered approach. *Journal of Systems Architecture*, 56(1), January 2010.

**Chapter 7** presents the conclusions of this thesis.

The research activities for this thesis are conducted as part of the research lines defined by the Distributed Control Systems (DCS)[4] group of the Automatic Control Department (ESAII)[5] at the Technical University of Catalonia (UPC)[6]. This thesis is built upon the results published in two previous DCS group thesis: [Mar02] and [Vel06].

[4]DCS, `http://dcs.upc.edu/`
[5]ESAII, `http://webesaii.upc.edu/`
[6]UPC, `http://www.upc.edu/`

# Chapter 2

# One-shot task model

## 2.1 Introduction

In a context where embedded control systems implemented in small micro-processors enabled with real-time technology, control laws are often designed according to discrete-time control systems theory and implemented as hard real-time periodic tasks. Standard discrete-time control theory mandates to periodically sample (input) and actuate (output). Depending on how input/output (I/O) operations are performed within the hard real-time periodic task, different control task models can be distinguished. However, existing task models present important drawbacks. They generate task executions prone to violate the periodic control demands, problem known as sampling and latency jitter. Or they impose synchronized I/O operations at each task job execution that produce a constant but artificially long I/O latency.

To overcome these limitations, in this chapter the "*one-shot*" task model is presented. The novel control task model is built upon control theoretical results that indicate that standard control laws can be implemented considering only periodic actuation. That is, the periodic sampling requirement can be relaxed. Taking advantage of this property, the one-shot task model permits to remove endemic problems for real-time control systems such as sampling and latency jitters while to minimizing the harmful effects that artificially imposed longer I/O latencies have on control performance. To corroborate its correctness and effectiveness, simulations and real experiments have been carried out. They show the benefits that can be obtained by the application of the one-shot task model in terms of operation and performance, compared to existing real-time and/or control based methods and models currently used a) for implementation of control algorithms using real-time technology or b) for minimizing the degrading effects that jitters have on control performance.

## 2.2 Problem set-up

The use of periodic task has been the common approach when control systems are implemented on a computing platform. Within this approach, two periodic control task models have been traditionally used in order to meet control timing demands with real-time technology:

- The first model, identified as the "*naif*" task model, represents the common practice implementation of control loops in real-time control systems [ÅCES00]. The naif model assumes the standard hard real-time periodic task model for control tasks where sampling (input)

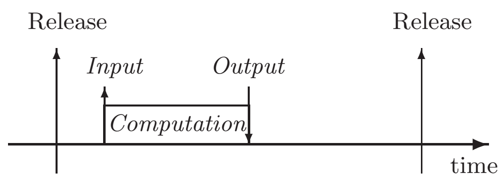


Figure 2.1: Naif task model.

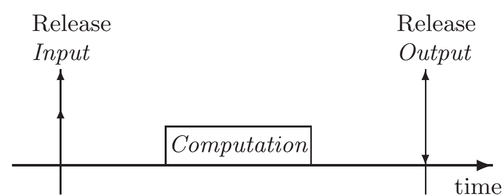


Figure 2.2: One-sample task model.

and actuation (output) occurs at the beginning and at the end of each job execution, as illustrated in Figure 2.1. This model assumes that deadline is equal to the period.

- In the second model, identified as the "*one-sample*" task model, input/output operations are performed periodically by hardware functions [LL73] as illustrated in Figure 2.2. In this model the deadline equals the sampling period and there is a constant input/output latency of one sampling period.

Both task models present important drawbacks: the one-sample task model imposes an artificial long time delay in the closed loop system of one-sample, which may introduce an unnecessary although predictable control performance degradation. The naif task model introduces sampling and input/output time variations known as jitters, which make the analysis and design of feedback control loops extremely difficult.

## 2.3 One-shot task model

The key property that permits to develop the one-shot task model states that control algorithms can be implemented considering only periodic actuation. One-shot task model has been proposed initially by [MVF+07] and extended by [LVM08]. Figure 2.3 graphically illustrates the theoretical approach. In the following subsections, this figure will be described in detail. The time reference for the three subfigures is the same.

### 2.3.1 Timing analysis of standard controllers

Consider the mathematical description of a system given by the $n$-order state-space model of a linear time-invariant discrete-time system with $m$ inputs and $p$ outputs and a sampling period of $h$, [ÅW97]

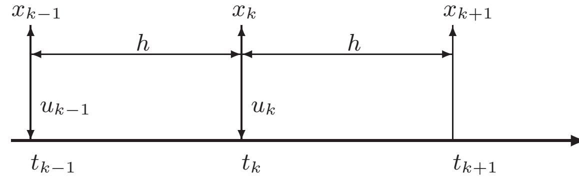
$$\begin{array}{rcl} x_{k+1} & = & \Phi(h)x_k + \Gamma(h)u_k \\ y_k & = & Cx_k, \end{array} \tag{2.1}$$

where $x_k \in \mathbb{R}^{n\times 1}$ is the plant state, $u_k \in \mathbb{R}^{m\times 1}$ and $y_k \in \mathbb{R}^{p\times 1}$ are the inputs and outputs of the plant, matrix $C \in \mathbb{R}^{p\times n}$ is the output matrix, and matrices $\Phi \in \mathbb{R}^{n\times n}$ and $\Gamma \in \mathbb{R}^{n\times m}$ are obtained using

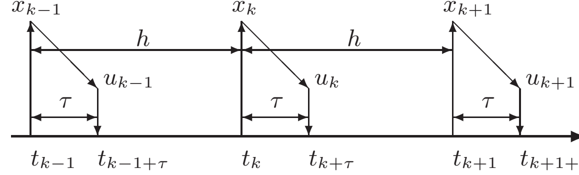
$$\Phi(t) = e^{At}, \quad \Gamma(t) = \int_0^t e^{As} B ds, \tag{2.2}$$

with $t = h$, where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$ are the system and input matrices of the continuous-time form

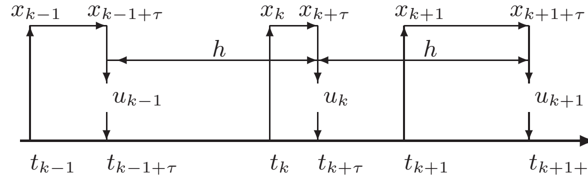
$$\begin{array}{rcl} \dot{x}(t) & = & Ax(t) + Bu(t) \\ y(t) & = & Cx(t). \end{array} \tag{2.3}$$



(a) standard controller



(b) standard controller considering a time delay



(c) controller with updated control signal

Figure 2.3: Timing analysis of controllers.

If the inputs $u_k$ are given by a control algorithm implemented as a periodic task, $k$ identifies the job within a sequence of jobs. For standard closed-loop operation of (2.1), the control signal $u_k$ is given by

$$u_k = Lx_k \quad \text{with} \;\; L \in \mathbb{R}^{m\times n}, \tag{2.4}$$

where $L$ is the state feedback gain obtained using standard control design methods from matrices $\Phi$ and $\Gamma$ such as pole placement or optimal control.

The application of (2.4) to the plant mandates computing the control signal with *zero* time, as illustrated in Subfigure 2.3(a) where the $k^{th}$ control signal $u_k$ is applied to the plant at the same time instant $t_k$ that the $k^{th}$ sample $x_k$ is taken[1], sample that is required to compute the control signal. This is physically impossible!, computing the control signal takes time. This theoretical model (2.1) can be augmented to cope with a time delay modelling an I/O latency that appears due to the computation of the control algorithm. The standard model that incorporates a time delay $\tau$, with $\tau \leq h$, is [ÅW97]

$$x_{k+1} = \Phi(h)x_k + \Phi(h-\tau)\Gamma(\tau)u_{k-1} + \Gamma(h-\tau)u_k. \tag{2.5}$$

Note that matrices $\Phi(\cdot)$ and $\Gamma(\cdot)$ in equation (2.5) slightly differs from conventional notation. The

[1]Sample is used to refer to the full state vector, regardless of whether it has been sampled or observed.

purpose of this notation is to explicitly indicate dependencies on the sampling period and delay, $h$ and $\tau$ respectively.

Model (2.5) has been taken as the underlying control model for constructing real-time task models for control algorithms. As illustrated in Subfigure 2.3(b), it is based on two synchronization points, the sampling and actuation instants, on a time reference given by the sampling instants. At time $t_k$ the $k^{th}$ sample ($x_k$) is taken and the computation of the control signal ($u_k$) can be started. At time $t_{k+\tau}$ the calculated control signal is applied to the plant. The sampling period $h$ is defined from $t_k$ to $t_{k+1}$, and the time delay $\tau$ from $t_k$ to $t_{k+\tau}$. Note that the standard subscript notation in the standard state-space model with time delay (2.5) may be misleading. In the model, the $k$-subscript identifies the job within a sequence of jobs, not the time events occur. For example, $u_k$ is the control signal of the $k$-job, although it is applied at time $t_{k+\tau}$.

For closed-loop operation of (2.5), the control signal will be

$$u_k = \begin{bmatrix} L_1 & L_2 \end{bmatrix} \begin{bmatrix} x_k \\ u_{k-1} \end{bmatrix} = L_1 x_k + L_2 u_{k-1}$$
$$\text{with} \quad L_1 \in \mathbb{R}^{m\times n}, \quad L_2 \in \mathbb{R}^{m\times m}, \tag{2.6}$$

where $[L_1\ L_2]$ is the state feedback gain obtained using standard design procedures as before. With this configuration, the control signal $u_k$ is held from $t_{k+\tau}$ to $t_{k+1+\tau}$.

**Remark 1.** *The standard controller designed to cope with a time delay (2.6) involves computing the control signal ($u_k$) to be applied to the plant at time $t_{k+\tau}$ using a sample ($x_k$) taken at $t_k$, $\tau$ time units before (represented by diagonal arrows in Subfigure 2.3(b) ).*

**Remark 2.** *The closed-loop model given by (2.5) and (2.6) holds the control signal $u_k$ from $t_{k+\tau}$ to $t_{k+1+\tau}$.*

### 2.3.2 Controller with updated control signal

Rather than applying standard controllers (2.4) or (2.6), the one-shot task model, as illustrated in Subfigure 2.3(c), proposes to apply a controller with updated control signal to account for the delay. Instead of computing the control signal with an state vector $x_k$ that becomes outdated at the time the control signal is applied, it is proposed to use an updated (estimated) state vector. Thus, the control signal $u_k$ is computed it terms of the estimated state at time $t_{k+\tau}$, labelled $x_{k+\tau}$. Moreover, $x_{k+\tau}$ can be computed using a sample $x_k$ taken at any time $t_k \in (t_{k-1+\tau}, \quad t_{k+\tau})$.

Therefore, the controller will first estimate the state

$$x_{k+\tau} = \Phi(\tau_k)x_k + \Gamma(\tau_k)u_{k-1}, \tag{2.7}$$

where matrices $\Phi(\cdot)$ and $\Gamma(\cdot)$ are given by (2.2) for $t = \tau_k$, with

$$\tau_k = t_{k+\tau} - t_k. \tag{2.8}$$

And second, the controller will compute the control signal

$$u_k = Lx_{k+\tau} \quad \text{with} \;\; L \in \mathbb{R}^{1\times n}, \tag{2.9}$$

where $L$ is the original controller gain (2.4) obtained using standard control design methods from matrices $\Phi(h)$ and $\Gamma(h)$.

**Remark 3.** *A controller using (2.7)-(2.9) relies on a the time reference given by the actuation instants. The time elapsed between consecutive actuation instants $t_{k+\tau}$ and $t_{k+1+\tau}$ is the sampling*

**Algorithm 1**: `Controller with updated control signal`

```
1 begin
2     x_k = read_input()
3     t_k = get_time()
4     t_{k+τ} = t_{k+τ} + h
5     τ_k = t_{k+τ} − t_k
6     x_{k+τ} = Φ(τ_k)x_k + Γ(τ_k)u_{k−1}
7     u_k = Lx_{k+τ}
8     u_{k−1} = u_k
9 end
```

Figure 2.4: Controller pseudo-code.

*period $h$. Moreover, no delay is present in the closed loop model. And samples are not required to be periodic because $\tau_k$ in (2.8) can vary at each closed-loop operation.*

The equivalence relations between the state space models in closed loop form when using standard controllers as in (2.4) or (2.6), or when applying the controller using updated control signals (2.7)-(2.9), are summarized next:

- For irregular sampling with $t_k \in (t_{k-1+\tau} \quad t_{k+\tau})$, standard controllers can not be applied. However, the proposed controller given by (2.7)-(2.9) can be applied.
- The control law in (2.9) has the same dimension than the control law in (2.4). This keeps the controller design problem simpler than the case of the standard model with time delay, where the controller gain in (2.6) has to also consider the previous control signal.
- The application of controller (2.6) is more general than (2.7)-(2.9). That is, a more complete set of dynamics can be achieved using standard controllers.

### 2.3.3 Controller design

The implementation of the proposed controller (2.7)-(2.9) requires executing a control algorithm that slightly differs from conventional controllers.

The pseudo-code of the controller with updated control signal is given in Figure 2.4. The controller first samples the plant and gets the current time, thus obtaining $x_k$ and $t_k$ (lines 2 and 3). Afterward, lines 4 and 5 are used to compute the time that will elapse from $t_k$ to the actuation instant, thus implementing (2.8). Recall that actuation instants are given by $h$ (remark 3). The initial value of $t_{k+\tau}$ is zero. Line 6 implements the state prediction at the actuation instant (2.7). Therefore, the current value of $\tau_k$, obtained in line 5, is used to compute the estimated vector using the sampled state and the previous control signal. Then, in line 7 the control signal is computed using the gain $L$ that has been obtained at the design stage. The last line of the pseudo-code is used to save the control signal value for the next execution.

The pseudo-code shown in Figure 2.4 should be executed *periodically*, meaning that it should meet the timing imposed by (2.7)-(2.9) and illustrated in Subfigure 2.3(c). It is important to stress that in the pseudo-code the control signal is not directly output to the plant because the real-time kernel will be in charge of enforcing its application at the actuation instants.

Looking at computational overhead, the state vector estimation (line 6) is the most significant modification compared to a standard control algorithm. However, it does not add significant overhead because it implies the same operations than standard observer based control designs.

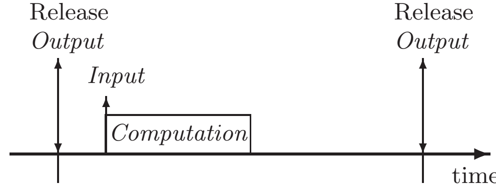



Figure 2.5: One-shot task model.

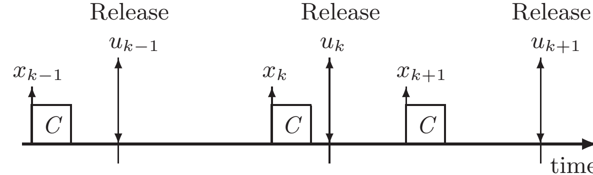



Figure 2.6: One-shot task execution.

### 2.3.4 Task model analysis and properties

Remark 3 provides the semantics to build a new task model for control tasks, the one-shot task model. The proposed controller (2.7)-(2.9) forces job executions to occur within two consecutive actuation instants, separated by fixed $h$ time units. Therefore, each job release takes place at each actuation instant, that is $r_k = kh$. And each job deadline is given by the next actuation instant, that is $d_k = r_k + h$. With these timing constraints, the one-shot task model matches the standard Liu and Layland [LL73] periodic task model with deadline equal to period, but with the following requisites:

- Sampling is performed at each job execution start time.
- Actuation has to be carried out by a synchronized output operation performed by the kernel at the release times.
- The control algorithm implements the pseudo-code of Figure 2.4.

Figure 2.5 illustrates the one-shot task model. Figure 2.6 shows the execution of a one-shot task in the scenario illustrated in Subfigure 2.3(c).

The one-shot task model has several appealing properties for controllers. Although some of them have been already stated, all of them are summarized next.

**Property 1.** *Compatible with standard scheduling: the one-shot task model does not demand any specific timing constraints other than the ones of the standard hard real-time periodic task model. Therefore, it can be applied within existing scheduling algorithms for periodic tasks.*

**Property 2.** *Improves schedulability: the only synchronized operation required by each one-shot task is the actuation. Therefore, the number of interrupts handlers for a real-time system executing multiple control loops is cut by half compared to those models using also synchronization operations for sampling. A simple consequence is that task set schedulability is improved (see [JS93] for an example of analysis of hardware interrupts in dynamic priority task systems).*

**Property 3.** *Absorbs scheduling jitters: the a priori known time reference for the task model is given by the actuation instants. Moreover, no delay is present in the model (remark 3). Therefore, latency jitter is not a concern. Sampling jitter problems also disappear because equation (2.7) absorbs the irregular sampling.*

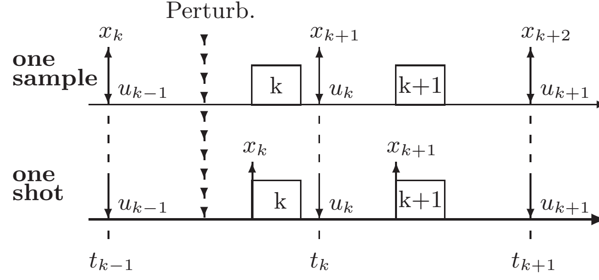



Figure 2.7: One-shot task vs. one-sample task.

**Property 4.** *Does not require switching controllers: compared to other solutions [MFRF01] where the gain is updated according to varying timing constraints, in the one-shot task model the gain is constant because constraints do not vary: sampling period is constant and no delay is present.*

**Property 5.** *Does not force long I/O latencies: the one-shot task model performs sampling at the beginning of each job execution, and actuation is performed at the next task release. Therefore, the I/O latency goes from each job start time to the next release. In the general case, this time interval will be less than one-sample which is the I/O latency forced by previous approaches [LL73].*

As a final observation, note that Property 3 provides isolation between control tasks, like in the approach presented by [CE03]. That is, although task instances may be subject to jitters, their operation is not affected by them. Therefore, the performance that will be achieved by tasks using the one-shot task model is similar to the performance that tasks will achieve if executing in isolated processors.

The last property has an immediate benefit in terms of responsiveness of controllers based on the one-shot task model. Short I/O latencies permit controllers to be more efficient when affected by perturbations.

Let us compare a controller that is implemented with a task based on the one-sample model (e.g., [LL73]) to one with a task based on the one-shot model. Both tasks have the same timing constraints (same period and deadline). An arbitrary schedule gives the sequence of two jobs executions ($k$ and $k+1$ job execution), as shown in Figure 2.7. The top part shows the one-sample task jobs and the bottom part shows the one-shot task jobs. Each $k$-job input and output operations are labelled by $x_k$ and $u_k$, and illustrated with upside and downside arrows, respectively. Times $t_k$ mark jobs release times.

In Figure 2.7 a line of down arrows marks the arrival of a perturbation, which occurs before the $k$-job execution. It is interesting to note that for each $k$-job, all perturbations arriving at times $t \in (t_{k-1}, t_{k,s}]$ will be detected and started to be corrected by the $k$-job in the one-shot task but not in the one-sample task ($t_{k,s}$ denotes the $k$-job execution start time). In the one-sample task, these perturbations will be detected and started to be corrected by the $(k+1)$-job. That is, corrective operations will be send out one sampling period later in the one-sample model.

This benefits the control performance achievable by the one-shot task because it reacts faster to perturbations. If perturbations arrive at times $t \in (t_{k,s}, t_k)$, both tasks will provide the same responsiveness. As a consequence, scheduling policies that favor jobs executions near to their deadlines, e.g. [OY98], making longer the interval $(t_{k-1}, t_{k,s})$ will improve control performance if using the one-shot task model for controllers.

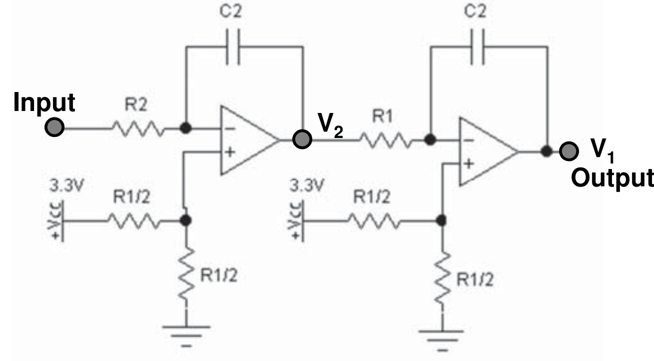



Figure 2.8: Electronic double integrator circuit.

## 2.4 Simulation and experiments

In this section, a set of simulations and experiments show the operation and performance of the "*one-shot*" task model compared with existing real-time and/or control based methods and models currently used in embedded control systems. This evaluation includes the "*naif*" task model (already discussed) and the "*one-sample*" task model as example of the application of formal methods to real-time implementation of control loops [HHK01]. In addition, the solution presented by [MFRF01], named "*switching*" task model, as example of control-based solution, and the model presented by [BRVC04], named "*split*" task model, as example of real-time based solution, are also evaluated.

### 2.4.1 Plant description

A second-order plant is used for the simulation and the control experiments. The plant is an electronic double integrator circuit illustrated in Figure 2.8 and it is defined by

$$\dot{x}(t) = \begin{bmatrix} 0 & -1/(R_1C_1) \\ 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ -1/(R_2C_2) \end{bmatrix} u(t), \tag{2.10}$$

considering that $R_1$ and $R_2$ resistors have a value of 100K$\Omega$, and $C_1$ and $C_2$ capacitors have a value of 470nF, the plant can be numerically described as

$$\dot{x}(t) = \begin{bmatrix} 0 & -21.2766 \\ 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 21.2766 \end{bmatrix} u(t), \tag{2.11}$$

As specified in (2.7), the state vector estimation requires to apply $\Phi(t)$ and $\Gamma(t)$ for each $t = \tau_k$. Therefore, these matrices have to be pre-computed in terms of $t$ as in

$$\Phi(t) = \begin{bmatrix} 1 & -h/21.2766 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad \Gamma(t) = \begin{bmatrix} h^2/(2*21.2766*21.2766) \\ -h/21.2766 \end{bmatrix}. \tag{2.12}$$

### 2.4.2 Simulation

The simulation model was implemented on a platform based on Matlab/Simulink using the True-Time toolbox, as shown on Figure 2.9. The processor is provided by the a TrueTime element which

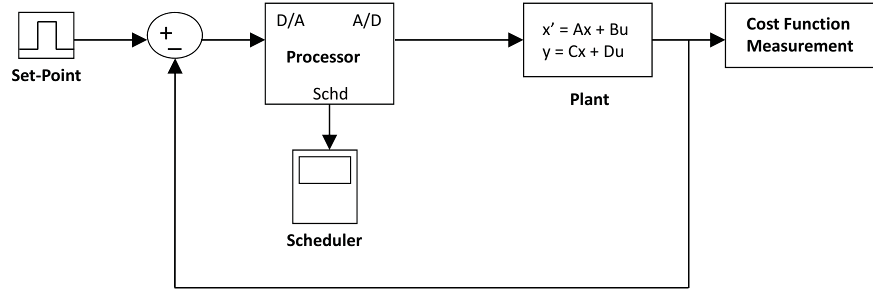



Figure 2.9: Simulation model.

Table 2.1: Task set parameters in milliseconds.

| | $h$ | D | C |
|---|---|---|---|
| $T_1$ | 30 | 30 | 10 |
| $T_2$ | 50 | 50 | 10 |

simulates a computer with a flexible real-time kernel executing user-defined tasks. The others elements are implemented by Matlab/Simulink blocks. The model of an electronic double integrator circuit is used as the controlled plant. The cost function element makes direct measurement from the plant. The plant control cost is obtained from the quadratic continuous function

$$J = \int_0^{t_f} \left( x^T(t) Q x(t) + u^T(t) R u(t) \right) dt \tag{2.13}$$

where $t_f$ marks the final time of the evaluation period, and the $Q$ and $R$ represents the cost weighting matrices with the following values:

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad , \quad R = 1,$$

#### 2.4.2.1 Simulation set-up

For the analysis, let us consider the task set given in Table 2.1, where $h$ is the task period, D is the relative deadline and C is the worst case execution time. Task $T_2$ is a control task while task $T_1$ is a non-control real-time periodic task. Under earliest deadline first (EDF [LL73]) scheduling, it is easy to observe that control task $T_2$ suffers sampling and latency jitter.

Figure 2.10 shows the task set schedule over the task set hyper-period under EDF, assuming that tasks execute on their worst case execution time. As it can be seen, the control task $T_2$ suffers sampling and latency jitters. For example, the first and fourth instances of $T_2$ starting at times 0ms and 150ms respectively, are preempted at times 5ms and 155ms by the first and the sixth instances of $T_1$, thus provoking latency jitters. In addition, the third instance of $T_2$ starts executing later than its expected release time (100ms), suffering a sampling jitter.

Within this scenario, the control task $T_2$ will implement a standard pole placement control law to track the square wave set-point. The state feedback controller places the continuous closed-loop poles at $-18.4261+10.6383i$ and $-18.4261-10.6383i$. The corresponding discrete closed-loop poles

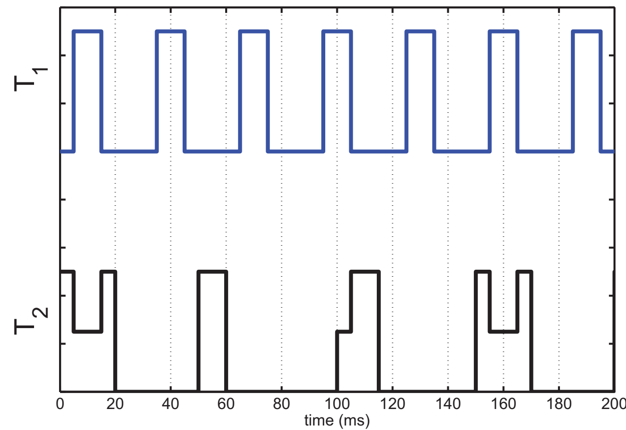



Figure 2.10: EDF schedule timing.

locations depend on the control task period (i.e., sampling period), that together with the task latency (i.e., time delay), will then determine the controller gain. Since the delay modelling the computation time has also been considered in some approaches, the third discrete-time closed-loop pole is set to 0 whenever required. Therefore, the specific timing used to design the controller is listed next, according to the six strategies under evaluation:

1. Naif task model: the sampling period and time delay for designing the control law are 50ms and 10ms.

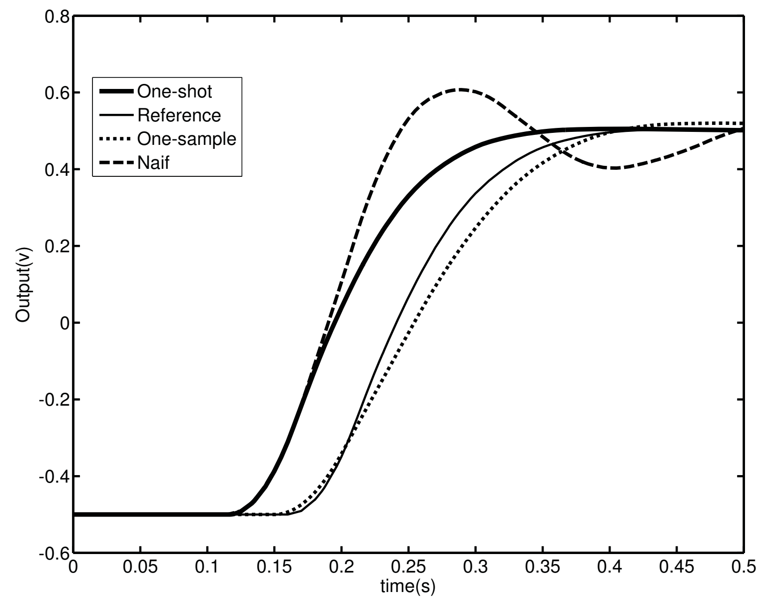
2. One-sample task model: the sampling period and time delay for designing the control law are 50ms and 50ms. In addition, the control task will execute sampling and actuation at a constant I/O latency of one period, 50ms.

3. One-shot task model: the sampling period and time delay for designing the control law are 50ms and 0ms. Recall remark 3 where it is noted that the sampling period is constant on the basis of equidistant actuation instants, and the time delay is zero since the control signal is computed using the updated state vector.

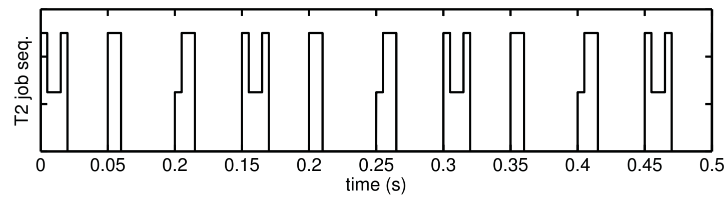
4. Switching controller gains: the control task will be applying one controller gain out of three different gains depending on the real sampling periods and latencies that can be derived from the EDF schedule, $\{(h, \tau)\} = \{(50, 10), (55, 10), (45, 20)\}$ms, assuming that tasks execute on their worst case execution time.

5. Split task model: the operation of the control algorithm is split into three sub-tasks, sampling, control computation and actuation, that are scheduled separately. For this case, the sampling period and time delay for designing the control law are 50ms and 10ms respectively.

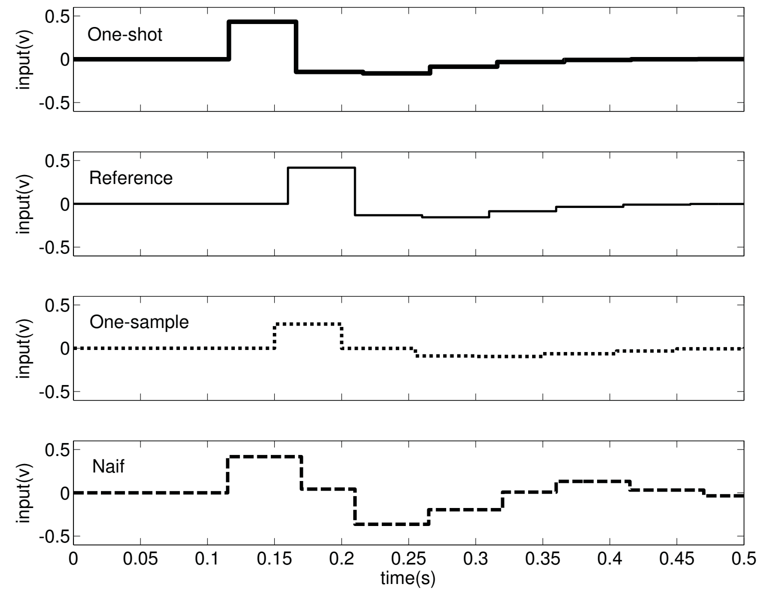
6. Reference model: this model has been designed only for comparative purposes, is a pure periodic controller, with period 50ms and no delay, thus the model is simulating an ideal execution in isolation. In this case the control task is not subject to jitters.




(a) plant outputs




(b) sequence of jobs




(c) control signals

Figure 2.11: Detailed view of the operation of several control strategies.

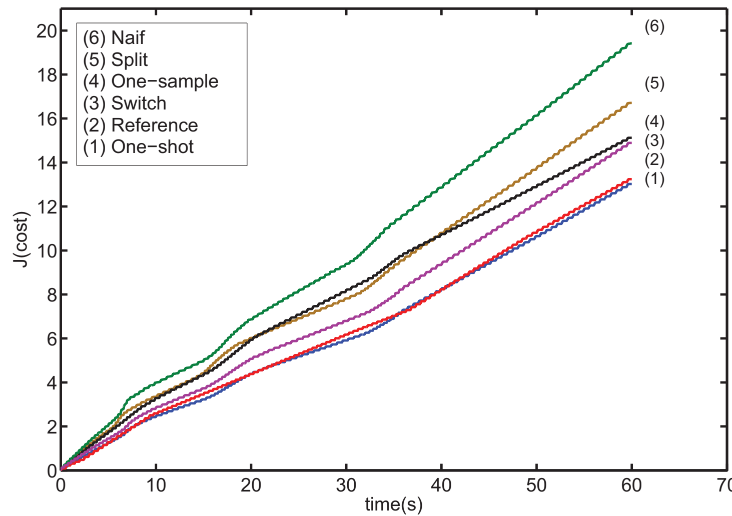



Figure 2.12: One-shot performance evaluation in front of existing solutions (simulation).

#### 2.4.2.2 Detailed performance analysis

This section gives simulation details of the operation of the one-shot task model in terms of plant outputs and control signals compared to other strategies. The simulation involves evaluating the tracking on a set-point change from −0.5V to +0.5V, occurring at time 100ms.

For the analyzed control strategies, Figure 2.11 shows the plant outputs (top) and the control signals (bottom) over 500ms. The middle subfigure shows the sequence of jobs of the control task $T_2$, where high level line means job in execution, and middle level line means job preempted due to the execution of jobs of $T_1$, and low level lines mean no job execution.

Since the reference model simulates an ideal execution, the sequence of jobs shown in the middle subfigure does not apply to this strategy. For the rest of strategies, the controller implemented in task $T_2$ uses the one-shot, the one-sample or the naif task model. Note that in this evaluation, the switching and the split task models are not assessed in order to simplify this detailed analysis. They are included in the summarized results given next.

As it can be seen in the top subfigure, the outputs of the reference controller and the controller using the one-shot task model are equal in terms of dynamics, but shifted in time. Taking into account that the set-point change occurs at time 100ms, the third job of the one-shot controller executes at time 105ms, and therefore it sees the new set-point and starts immediately correcting the tracking error. However, the third job of the reference controller does not see the new set-point since it executes exactly at 100ms, this controller starts correcting the error until the fourth job, i.e. at time 150ms.

Since the first tracking error is the same, both control signals are the same, but shifted in time, as illustrated in the bottom subfigure. In this case the one-shot controller is more reactive to the set-point change than the reference controller due to the specific phasing between samples and the set-point change. The one-shot benefits regarding the responsiveness to perturbation was already analyzed in Section 2.3.4.

The performance of the one-shot compared to the one-sample and naif task model is different in terms of plant output due to the difference in the control signals. The plant output of the

**Algorithm 2**: `void task_naif(void)`

```
1  begin
2      L: Controller gain
3      observer: Observer matrices
4      x_observed, x_observed_old, u_old : Intermediate variables
5      task_id=naif
6      task_initialization()
7      while(1)
8      begin
9          input = readInputPort()
10         observerComputation(x_observed, observer, x_observed_old, u_old, input)
11         u = calculateControlSignal(x_observed, reference, input, L)
12         writeOutputPort(u)
13         x_observed_old = x_observed
14         u_old = u
15         end_cycle()
16     end
17 end
```

Figure 2.13: Naif task pseudo-code.

one-sample gives different performance due to the introduced I/O latency of 50ms. As it can be seen in the bottom subfigure, the first correcting action, applied at time 150ms, is of different magnitude than the one-shot because it is based on an extended controller (2.6). In addition, subsequent control actions do not incorporate the state estimation used in the one-shot. The naif task model gives the worst dynamics because job executions are affected by jitters, which have not been accounted for in the controller.

All these effects imply that the desired performance of the tracking achieved by the controller using the one-sample and naif task model is not met. In addition, for all the evaluated task models, each set-point change produces an increment of cost if the tracking is evaluated using a standard quadratic cost function on the states and control inputs, as it is further analyzed next.

#### 2.4.2.3 Simulation results

Figure 2.12 shows the control performance achieved by the six strategies. The y-axis shows the cumulative error measured in terms of the quadratic cost function of the plant state and control signal, as defined by (2.13). Therefore, the lower the curve, the better the performance. The plant has been perturbed using the set-point changes from $-0.5$V to 0.5V and viceversa, occurring periodically each 0.5s over a total simulation time of 70s.

As it can be seen in Figure 2.12, the one-shot task model achieves the best performance. As expected, the naif task model achieves the worst performance because the I/O operations are subject to jitters. The other approaches, that follow different strategies to overcome the jitter problem, lie in the middle. The exact ordering in terms of performance will vary depending on the number of tasks, plants under control, and scheduling policy. The split task model strategy, although reducing the jitter variance, can not completely remove jitters. The one-sample task model approach eliminates jitters by construction at the expenses of forcing a one-sample delay at each job execution, which introduces performance degradation in the control loop operation. The switching task model strategy, although applying different gains according to the run-time jitters, also introduces some degradation due to the switching. Finally, looking at the one-shot task model,

**Algorithm 3**: `void task_onesample()`

```
1  begin
2      L: Controller gain
3      observer: Observer matrices
4      x_observed, x_observed_old, u_old : Intermediate variables
5      task_id=one_sample
6      task_initialization()
7      while(1)
8      begin
9          input = readKernel()
10         observerComputation(x_observed, observer, x_observed_old, u_old, input)
11         u = calculateControlSignal(x_observed, reference, input, L)
12         writeKernel(u)
13         x_observed_old = x_observed
14         u_old = u
15         end_cycle()
16     end
17 end
```

Figure 2.14: One-sample task pseudo-code.

since it eliminates jitters by construction without introducing a one sample delay, it is capable of achieving the best performance.

### 2.4.3 Control experiments

A control experiment involving the implementation of the one-shot task model in a real-time kernel is presented. The experiment shows the feasibility of implementing an embedded control application using the one-shot task model compared to existing approaches. The experimental platform consists in a Microchip dsPIC33 microcontroller[2] based system, running the Erika real-time kernel [Srl08a].

There are two tasks being executed in the dsPIC33 microcontroller. $T_1$ a non control tasks which executes every 5ms to obtain the updated state variables and sends the data to an external PC via RS-232 communication to compute the control cost. $T_2$ is a control task executed every 50ms which implements the different evaluated strategies. $T_1$ includes a 2ms delay to cause jitters on $T_2$. The plant control cost is calculated by transforming the continuous cost function specified by (2.13) into a discrete-time with an interval of 5ms. For further details on how to obtain the discrete control cost function refers to Appendix A.

The six strategies have been implemented in the experimental platform and their control performance are evaluated. However details for the switching task model and the split task model are omitted because their performance lie in the middle and because they are not as common as the others. The true comparison should be between the one-sample and the one-shot. In each model, the control task implements the pseudo-code given in Figure 2.4 and follows the specifications given in order to track the square wave. In addition to the controller gain, a deadbeat reduced observer for estimating the second state variable has been designed. This differs from the simulation described in Subsection 2.4.2 where it is assumed that all state variables are available for direct measurement.

[2]Microchip, `http://www.microchip.com/`

**Algorithm 4**: `void task_oneshot(void)`

```
1  begin
2      L: Controller gain
3      observer: Observer matrices
4      x_observed, x_observed_old, u_old : Intermediate variables
5      task_id=one_shot
6      task_initialization()
7      while(1)
8      begin
9          input = readInputPort()
10         current_time = get_time()
11         observerComputation(x_observed, observer, x_observed_old, u_old, input,
           current_time, next_job)
12         u = calculateControlSignal(x_observed, reference, input, current_time,
           next_job, L)
13         writeKernel(u)
14         x_observed_old = x_observed
15         u_old = u
16         end_cycle()
17     end
18 end
```

Figure 2.15: One-shot task pseudo-code.

#### 2.4.3.1 Task code implementation details

The pseudo-code of the tasks implementing the controller when naif, one-sample and one-shot task model are used, are described by Figures 2.13, 2.14 and 2.15 respectively. For all types of models, the pseudo-code starts by defining the main variables. Only the most significants are shown: gain ($L$) and observer matrices (*observer*), and the intermediate variables. In the start-up part the control tasks have to identify themselves right before the task initialization. Then, an infinite loop contains the main code, which slightly varies depending on the task model.

In the naif task model (Figure 2.13), which corresponds to the standard implementation, the main property is that sampling and actuation are performed at the beginning and end of each task job execution. First, the output variable is read from the input port and its value stored (*input*). With the value, the appropriated control signal $u$ is computed, taking into account the computation of the second state variable that is observed. Then, $u$ is written to the output port, state updates are performed, and the system call *end_cycle*() notifies the kernel the termination of the control task execution. Writing the control signal before updating the state is a code optimization that minimizes the I/O latency, as suggested by [Cer01]. The actual code of the task for all models also prevents saturation on the control signal. Here it has been omitted for the sake of clarity.

The main code in the infinite loop for the one-sample and one-shot task models vary with respect to the naif model. As it can be seen in Figure 2.14, the one-sample code does not access to I/O ports for reading the sample or writing the control signal. It obtains the sample from the kernel and it forwards the control signal to the kernel. The kernel is then in charge of reading samples and writing control signals at each one-sample job release time. Finally, it has to be pointed out that the observer computation and the control computation take into account the one sample delay of one period that this model requires.

As it can be seen in Figure 2.15, the one-shot code mixes features of the naif model and the one-sample model. The one-shot directly access the input port for reading the sample. However,

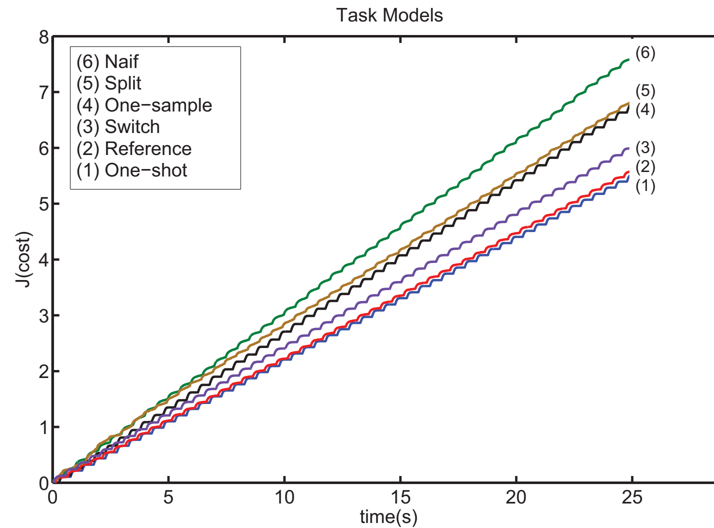



Figure 2.16: One-shot performance evaluation in front of existing solutions (experiment).

after computing the control signal, $u$ is forwarded to the kernel. An the kernel will apply it at the next job release time. For this model, the observer computation and the control signal computation require knowing the time at which the sample was taken as well as the time of the next job release. The former is read right after the task accesses the input port and stored in *current_time*, and the later belongs to the task control block structure, which is updated at the *next_job* variable.

In summary, as it can be seen in the three figures illustrating the three models, the codes are very similar. This confirms that implementing embedded control applications using the one-shot task model does not require many changes.

#### 2.4.3.2 Experimental results

The performance evaluation for the six strategies is measured in terms of the quadratic cumulative error using a discrete cost function. Figure 2.16 shows the performance results for the different strategies over a period of 25s. During this time the set-point is changing from −0.5V to 0.5V every 0.5s.

First, it can be observed that similar control performance results are obtained during the control experiments when compared with the simulation results. Also, it is interesting to note two other important facts. First, the one-sample approach curve lies between the naif and the one-shot. Therefore, this proves that the most current approach to avoid the jitter problem is outperformed by the approach presented in this chapter. But even more interesting is to compare the one-shot curve with respect the reference curve. The reference curve is the performance of the control task when it is executed in isolation, that is, without jitters. As it can be seen in the figure, the one-shot even outperforms this scenario. This is due to the fact that controllers implemented using the one-shot task model are more responsiveness to perturbations. The reference curve corresponds to the controller executing without jitters, that is, sampling at each release time and actuating right after the control algorithm computation (which is less than the period). Therefore, all perturbations affecting the circuit between the end of the control algorithm computation and the release time are detected at the next release time. However, in the one-shot, since samples are taken later than

the release time, more perturbations can be detected. Note that the real benefit of this property depends on the perturbation arrival times as well as on the workload. This property has been also observed during the simulation.

# Chapter 3

# One-shot task model extended to noisy measurements

## 3.1 Introduction

In this chapter the one-shot task model is extended to the case of noisy measurements. It is well known that systems are noise corrupted and that sensors in a control loop do not provide exact readings of desired quantities [May79]. In these cases, filtering is desirable since it removes the noise from signals while retaining the valuable information. The Kalman filter [Kal60] has been proved to be a useful tool for inferring the missing information from indirect and noisy measurements.

The one-shot task model includes a predictor that estimates, during each sample, the state variables values at the next actuation instance. If samples are noise-free and assuming that an adequate plant model is used, then the state prediction will be accurate and thus the control signal applied to the plant will be correct. However, if samples are corrupted by noise, then the prediction will fail and an incorrect control signal will be sent to the plant. The solution proposed in this work is to embed the Kalman filter into the one-shot task model when the system presents noisy measurements.

The Kalman filter combines all available measurement data, plus prior knowledge about the system and measuring devices, to produce an estimate of the desired variables in such a manner that the error is minimized statistically. The standard approach for the implementation of a discrete-time Kalman filter assumes strict periodic sampling and actuation. However, in the one-shot task model, the available measurements are not periodic. This poses the problem of adapting the standard Kalman filter to the case of irregular sampling, and decide when to apply the prediction and the correction phase. Two different strategies are presented, and their control performance and computation demand are analyzed through simulations and real experiments.

The application of Kalman techniques for systems with diverse type of non-periodic sampling can be found in the literature, such as for multi-rate control systems, e.g. [LSX08, PSCC08], or event-based control systems, e.g. [LM07, SNR07]. However, non of them applies to the problem tackled in this chapter.

## 3.2 Problem set-up

The discrete-time Kalman filter addresses the general problem of trying to estimate the system state of a discrete-time controlled plant with a sampling period $h$. Therefore, for the filter implementation

the model (2.1) can be enhanced by adding process and measurement noise ($w_k$ and $v_k$ respectively) as in

$$\begin{aligned} x_{k+1} &= \Phi(h)x_k + \Gamma(h)u_k + w_k \\ y_k &= Cx_k + v_k. \end{aligned} \tag{3.1}$$

The algorithm for implementing the Kalman filter is divided in two phases: time update (predictor) and measurement update (corrector). The predictor phase uses the previous estimation to produce the *a priori* estimation of the system state (equations (3.2) and (3.3)). In the corrector phase, measurement information from the system output is used to refine the prediction and obtain the *a posteriori* estimation (equations (3.4), (3.5) and (3.6)). The *a posteriori* estimation is used in the next predictor phase.

In the predictor phase, if it is considered that the estimation of the next system state is required as in (3.1), then in the predictor phase, the *a priori* estimation of the system state is

$$\hat{x}^-_{(k+1)} = \Phi(h)\hat{x}_{(k)} + \Gamma(h)u_{(k)} \tag{3.2}$$

where $\Phi(h)$ and $\Gamma(h)$ represent the system dynamics from (3.1), $\hat{x}_{(k)}$ defines the current *a posteriori* estimate of the process state, and $u_{(k)}$ represents the current input. The *a priori* estimation of the covariance error is

$$P^-_{(k+1)} = \Phi(h)P_{(k)}\Phi^T(h) + Q \tag{3.3}$$

where $P_{(k)}$ is the current *a posteriori* estimate of the covariance error, and $Q$ is the constant covariance value of the process noise.

In the corrector phase, the next Kalman gain value

$$K_{(k+1)} = \frac{CP^-_{(k+1)}}{CP^-_{(k+1)}C^T + R} \tag{3.4}$$

is obtained prior to the calculation of the *a posteriori* estimation, where $K_{(k+1)}$ is the Kalman gain, $C$ defines the constant measurement gain as in (3.1), and $R$ is the covariance value of the measurement noise. Then *a posteriori* estimation of the next state is

$$\hat{x}_{(k+1)} = \hat{x}^-_{(k+1)} + K_{(k+1)}(y_{(k)} - C\hat{x}^-_{(k+1)}) \tag{3.5}$$

where $y_{(k)}$ is the measured output of the system as in (3.1). The *a posteriori* estimation of the covariance error is

$$P_{(k+1)} = (I - CK_{(k+1)})P^-_{(k+1)} \tag{3.6}$$

where $I$ is the identity matrix. When using Kalman, the control signal for the closed loop operation is now calculated based on the current state estimation, as follows

$$u_k = L\hat{x}_k. \tag{3.7}$$

The implementation of a discrete-time Kalman filter is straightforward if strictly periodic sampling is ensured, note the dependency of equations (3.1),(3.2) and (3.3) on the sampling period $h$. However, integrating a Kalman filter with the one-shot task model raises some problems that require a detailed analysis. The Kalman filter algorithm has two phases which are prediction and correction. The correction must take place at the sampling instant, since it requires a process measurement in order to execute the correction, as shown in equations (3.5). However the one-shot

task model makes the synchronization at the actuation instants and the sampling periods may be irregular. Furthermore the one-shot task model uses a time difference (2.8) to estimate the state at the actuation instant (2.7), in addition to the estimations and predictions required by the Kalman filter algorithm.

## 3.3 One-shot and noisy measurements

By considering these aspects, two different approaches to embed a Kalman filter with the one-shot task model were identified. The first approach implements the Kalman correction just from sampling to actuation instants, while the second approach considers the complete sampling interval to implement the Kalman correction. For the rest of the chapter, the first approach is identified as the *half* Kalman filter and the second one as the *complete* Kalman filter.

### 3.3.1 Half Kalman filter

In this approach the Kalman filter is split into two parts. In the first one, from sampling ($t_{s,k}$) to actuation ($t_k$), only the predictor phase is used. In the second one, from actuation ($t_k$) to next sampling ($t_{s,k+1}$), the predictor and the corrector phases are executed, as illustrated in Figure 3.1(a). It is important to highlight that, during the first part, the corrector phase cannot be used since process measurements values, used for corrections, are only available at sampling instants and not at actuation instants.

Hence, if only predictor applies from sampling ($t_{s,k}$) to actuation ($t_k$), equations (3.2) and (3.3) transform to

$$\hat{x}_k^- = \Phi(\tau_k)\hat{x}_{s,k} + \Gamma(\tau_k)u_{k-1} \tag{3.8}$$

$$P_k^- = \Phi(\tau_k)P_{s,k}\Phi(\tau_k)^T + Q. \tag{3.9}$$

In the second part, from actuation ($t_k$) to next sampling ($t_{s,k+1}$), the Kalman predictor and corrector apply. First, the predictor from (3.2) and (3.3) is redefined as

$$\hat{x}_{s,k+1}^- = \Phi(h-\tau_{k+1})\hat{x}_k^- + \Gamma(h-\tau_{k+1})u_k \tag{3.10}$$

$$P_{s,k+1}^- = \Phi(h-\tau_{k+1})P_k^-\Phi(h-\tau_{k+1})^T + Q, \tag{3.11}$$

and then from (3.4), (3.5) and (3.6), the corrector phase is formulated in this strategy as

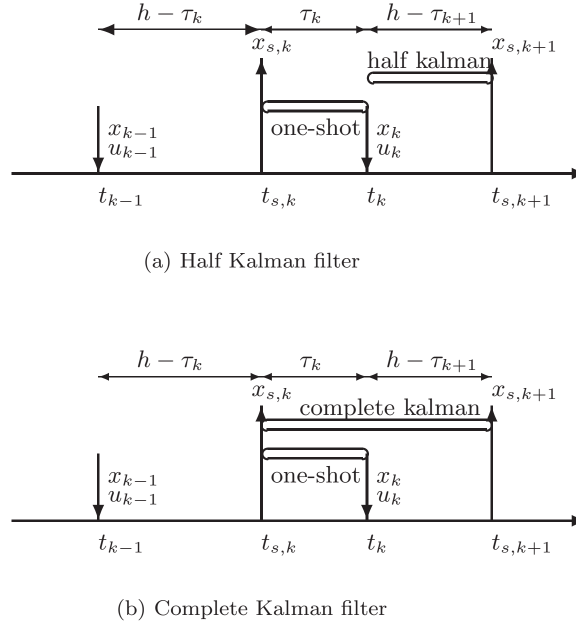
$$K_{s,k+1} = \frac{CP_{s,k+1}^-}{(CP_{s,k+1}^-C^T + R)} \tag{3.12}$$

$$\hat{x}_{s,k+1} = \hat{x}_{s,k+1}^- + K_{s,k+1}(y_{s,k+1} - C\hat{x}_{s,k+1}^-) \tag{3.13}$$

$$P_{s,k+1} = (I - CK_{s,k+1})P_{s,k+1}^-, \tag{3.14}$$

Then, the one-shot task model (2.7) and (2.9) can be implemented. Notice that the estimation of the state at the actuation instant has been already obtained in (3.8). Hence the control signal is calculated by

$$u_k = L\hat{x}_k^-. \tag{3.15}$$




(a) Half Kalman filter

(b) Complete Kalman filter

Figure 3.1: Kalman filter design approaches

### 3.3.2 Complete Kalman filter

This approach uses a Kalman filter to predict and correct from current sampling ($t_{s,k}$) to next sampling ($t_{s,k+1}$). In addition, the one-shot task model requires an estimation from sampling ($t_{s,k}$) to actuation ($t_k$), as illustrated in Figure 3.1(b).

If the complete sampling interval is considered, the Kalman *a priori* estimation can be obtained by substituting (3.8),(3.9) into (3.10),(3.11) respectively, then the following predictor phase equations are obtained

$$\begin{aligned}\hat{x}^-_{s,k+1} &= \Phi(h-\tau_{k+1}+\tau_k)\hat{x}_{s,k} \\ &\quad +\Phi(h-\tau_{k+1})\Gamma(\tau_k)u_{k-1} \\ &\quad +\Gamma(h-\tau_{k+1})u_k \qquad (3.16)\\ P^-_{s,k+1} &= \Phi(h-\tau_{k+1}+\tau_k)P_{s,k}\Phi(h-\tau_{k+1}+\tau_k)^T \\ &\quad +\Phi(h-\tau_{k+1})Q\Phi(h-\tau_{k+1})^T+Q. \qquad (3.17)\end{aligned}$$

Notice that $u$ is not constant over the complete sampling period. Hence, equation (3.16) considers $u_{k-1}$ and $u_k$. Also, the sampling period is not constant and it is equal to $h-\tau_{k+1}+\tau_k$.

From (3.4), (3.5) and (3.6), the corrector phase is formulated in this approach

$$K_{s,k+1} = \frac{CP^-_{s,k+1}}{(CP^-_{s,k+1}C^T+R)} \qquad (3.18)$$

$$\hat{x}_{s,k+1} = \hat{x}^-_{s,k+1}+K_{s,k+1}(y_{s,k+1}-C\hat{x}^-_{s,k+1}) \qquad (3.19)$$

$$P_{s,k+1} = (I-CK_{s,k+1})P^-_{s,k+1}. \qquad (3.20)$$

According to the one-shot task model, the control signal is calculated from the estimation of the state at the actuation instant, which is taken from the *a posteriori* state estimation at sampling instance. Therefore, equations (2.7) and (2.9) of the task model are redefined as

$$\hat{x}_k = \Phi(\tau_k)\hat{x}_{s,k} + \Gamma(\tau_k)u_{k-1} \quad (3.21)$$

$$u_k = L\hat{x}_k. \quad (3.22)$$

### 3.3.3 Discussion

At first sight both approaches are similar and it is expected that both will produce similar results. However, in a deeper analysis, there are some differences that may affect the computational demand of their implementation.

By using the half Kalman filter, the computation of the estimated state at the sampling instance ($\hat{x}^-_{s,k+1}$) becomes simpler compared with the complete Kalman approach, since the complete approach requires two different control values $u_{k-1}$ and $u_k$. On the other hand, the half Kalman filter requires to obtain $\hat{x}^-_k$ previous to $\hat{x}^-_{s,k+1}$, which may imply an additional operation. However $\hat{x}^-_k$ is required anyway by the one-shot model. In addition, the implementation of the complete approach requires calculating three different $\Phi$ values, i.e. $\Phi(\tau_k)$, $\Phi(h-\tau_{k+1})$ and $\Phi(h-\tau_{k+1}+\tau_k)$. However, in the half approach only two $\Phi$ values are required, i.e. $\Phi(h-\tau_k)$ and $\Phi(\tau_k)$. Hence,the half approach simplifies the implementation and it may reduce the computational demand.

## 3.4 Simulation and experiments

This section presents the simulation and experiments that shows the advantages and the feasibility of implementing the Kalman filter using the one-shot controller. First, the simulations show that Kalman filter and one-shot task model can work together and preserve their own benefits, i.e., remove noise and eliminate jitters degrading effects. Then, the experiments focus on demonstrating that the Kalman filter and one-shot controller can be successfully embedded into a microcontroller. Both simulations and experiments present detailed information regarding the control performance of the different implementations approaches.

### 3.4.1 Implementation approaches

Five different approaches has been implemented for simulation and experiments, in every case noisy measurements from the plant is considered. Two approaches implement the one-shot controller, while the other three uses a standard controller also identified as naif, as described in Subsection 2.2.

1. Kalman with standard controller (no jitters): the control task is executed in stand alone, so there is no jitters. This approach is used as a reference.
2. Half Kalman with one-shot controller: implements the half Kalman algorithm according to (3.8)-(3.15), jitters affect the execution of the control task.
3. Complete Kalman with one-shot controller: implements the complete Kalman algorithm based on (3.16)-(3.21), control task also suffers from jitters.
4. Kalman with standard controller: the standard discrete-time Kalman filter, which assumes periodical sampling (3.2)-(3.7), is embedded into the naif task model. There are jitters during the control task execution.
5. No Kalman with standard controller: in this approach there are jitters and the naif controller does not implement the Kalman algorithm.

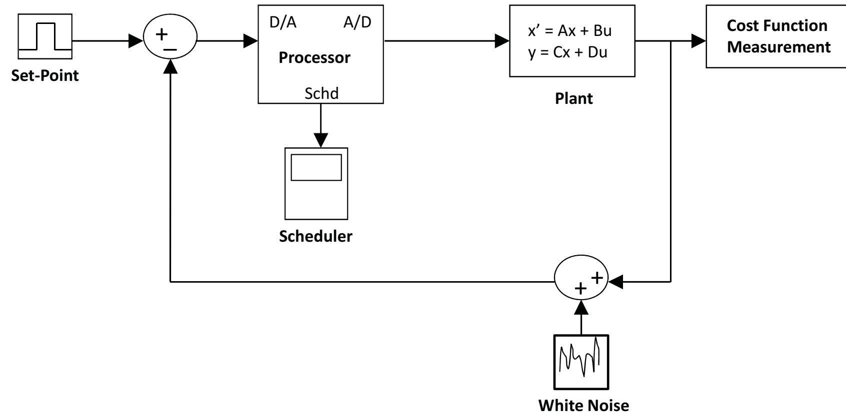



Figure 3.2: Simulation model.

### 3.4.2 Simulation

As in Subsection 2.4, simulation has been conducted using a Matlab/Simulink platform with the TrueTime toolbox. Figure 3.2 shows the simulation model which includes a *white noise* element to simulate noisy measurement from the plant, notice that cost function measurement are taken directly from the plant, so control cost values are not affected by noise. An electronic double integrator described in Subsection 2.4.1 is used as the controlled plant.

#### 3.4.2.1 Simulation set-up

The controller gain $L$ corresponds to the discrete linear quadratic regulator for (3.1), which minimizes a discrete cost function equivalent to the continuous cost function

$$J = \int_0^\infty \left(x^T(t)Qx(t) + u^T(t)Ru(t)\right) dt \tag{3.23}$$

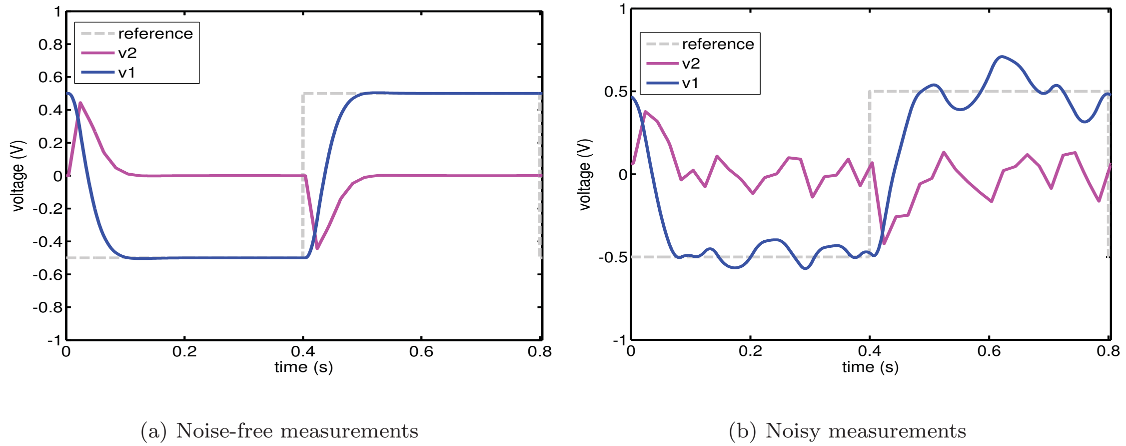
where the weighting matrices $Q$ and $R$ are the identity.

For simulation purposes a sampling period of $h = 50$ms is considered for the control task, with a task latency of 10ms. In order to simulate jitters, the control task shares the processor with a non-control task. Both tasks are scheduled using EDF policy. The non-control task has a sampling period of 20ms and a task delay of 10ms.

Since the present analysis is focused in noisy measurements, the simulation model only includes an element to simulate the measurement noise, but a plant noise element is not included. Therefore the Kalman filter was designed considering a very small plant noise covariance $Q_n = E(w \cdot w^T) = 1 \cdot 10^{-12}$, meanwhile for the measurement covariance, configured in the *white noise* element, a heuristical relatively large value was selected $R_n = E(v \cdot v^T) = 1 \cdot 10^{-3}$, where $w$ and $v$ are the plant noise and the measurement noise, respectively.

#### 3.4.2.2 Simulation results

Considering the standard naif task model, the system response is degraded considerably when noisy measurements affects the controller activities. Subfigure 3.3(a) shows the plant response without




(a) Noise-free measurements

(b) Noisy measurements

Figure 3.3: Simulation system response

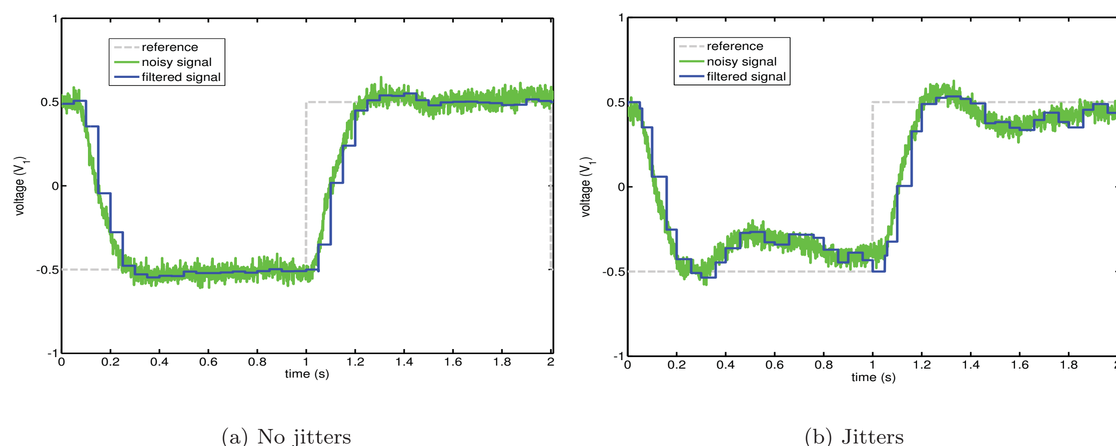
noise and Subfigure 3.3(b) shows the response when there are noisy measurements, $v_1$ and $v_2$ for both figures are obtained directly from the plant states.

To solve this problem Kalman filter can be implemented in the controller to remove the noise from signals while retaining the valuable information. Subfigure 3.4(a) illustrates illustrates that control performance can be improved when noisy signals are properly filtered, for the sake of clarity only $v_1$ is shown. However, if jitters are introduced due task scheduling, the system response is again deteriorated, as illustrated in Subfigure 3.4(b).

It has been demonstrated that the one-shot task model is capable to remove the negative effects caused by the scheduling jitters. Now, if Kalman filter is embedded into a one-shot task model, the degrading effects of noise and jitters can be removed. Figure 3.5 shows the $v_1$ transient response for the half Kalman and complete Kalman implementation in the one-shot task model when noise and jitters are introduced. Notice that no major difference is detected in the response of these two implementation. In addition, it can be observed that control performance of half Kalman and complete Kalman are considerably improved when compared with the standard controller with jitters, from Subfigure 3.4(b), and their response is similar to the one provided by the standard controller with no jitters, from Subfigure 3.4(a).

#### 3.4.2.3 Simulation performance evaluation

The different implementation approaches specified in Subsection 3.4.1, were evaluated in terms of control performance. For each approach, performance was measured with the continuous cost defined in (3.23). Table 3.1 presents the control performance results for a 40s simulation period. In general, these results indicate that half Kalman (B) and complete Kalman (C) implementation have better performance compared with the standard controller (D) and (E). Also half Kalman (B) and complete Kalman (C) performances are very close to the performance provided by the Kalman implementation with no jitters (A). Further details on performance evaluation can be found in the next subsection which includes a comparison between the simulation performance and the experimental performance of the different approaches.




(a) No jitters

(b) Jitters

Figure 3.4: Kalman filter implementation using the standard controller

Table 3.1: Simulation control performance.

| Implementation Approach | Control Performance |
| --- | --- |
| (A) Kalman with standard controller (no jitters) | 3.7695 |
| (B) Half Kalman with one-shot controller | 3.9022 |
| (C) Complete Kalman with one-shot controller | 3.9224 |
| (D) Kalman with standard controller | 4.9786 |
| (E) No Kalman with standard controller | 5.0316 |

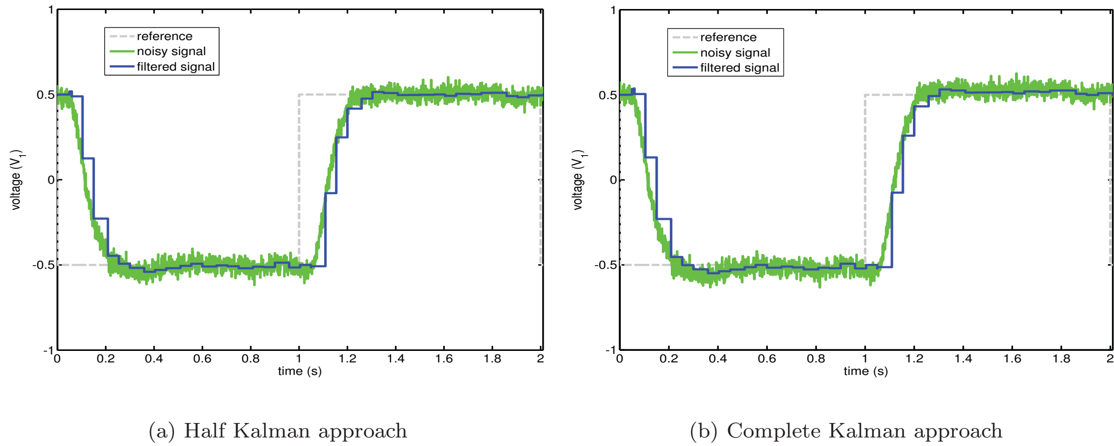
### 3.4.3 Control experiments

As in Subsection 2.4.3, control experiments have been conducted using a Microchip dsPIC33 microcontroller based system, running the Erika real-time kernel. An electronic double integrator circuit is used as the controlled plant which is described in Subsection 2.4.1.

#### 3.4.3.1 Experiment set-up

The controller gain $L$ has been designed using optimal control technique based on discrete cost function equivalent to the continuous cost function (3.23). Considering a sampling period of $h =$ 50ms, the optimal controller gain is $L = \begin{bmatrix} 0.4324 & -1.0255 \end{bmatrix}$. Jitters are generated by scheduling a non-control task in the dsPIC33 microcontroller. A non-control task executes every 5ms to obtain the updated state variables and sends the data to an external PC via RS-232 communication to compute the control cost. This task includes a 2ms delay in order to produce jitters on the control task. The control task is executed every 50ms and implements each one of the evaluated approaches specified in Subsection 3.4.1. Both tasks (non-control and control) are scheduled using EDF (earliest deadline first) policy.The plant control cost is calculated by transforming the continuous cost function (2.13) into a discrete-time with an interval of 5ms. For further details on how to obtain the discrete control cost function refers to Appendix A.

The Kalman filter was designed taking into account the noise covariances $Q_n = E(w \cdot w^T) =$ $2 \cdot 10^{-7}$ and $R_n = E(v \cdot v^T) = 8 \cdot 10^{-5}$ extracted from the electronic circuit of the experimental setup, where $w$ and $v$ are the plant noise and the measurement noise, respectively. Off-line sample measurements data, using the dsPIC33 and considering sampling periods of $h = 50$ms, were taken




(a) Half Kalman approach (b) Complete Kalman approach

Figure 3.5: Kalman filter implementation using one-shot controller

in order to determine the measurement noise covariance. Plant noise covariance was calculated from data obtained from direct plant measurements with calibrated instruments. In both cases, the data obtained corroborate the presence of white noise.

#### 3.4.3.2 Kalman algorithm implementation

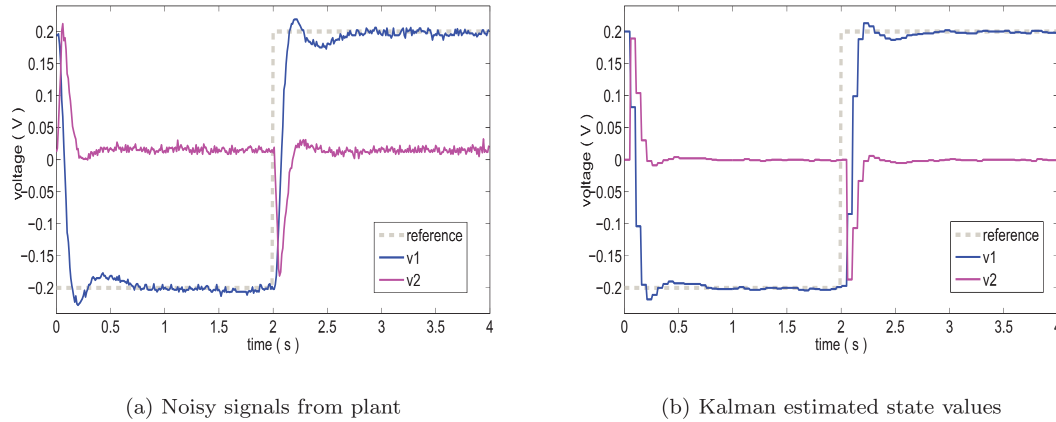
The implementation of the half Kalman algorithm and the complete Kalman algorithm into the dsPIC33 processor, requires to calculate $\Phi(\cdot)$ and $\Gamma(\cdot)$ as function of different time values according to (3.16) for complete Kalman and according to (3.8), (3.10) for half Kalman. These calculations represent the most time consuming operations for the processor, since equation (2.2) needs to be implemented in the dsPIC33. Then to reduce processor time the pre-computed matrices defined by (2.12) were implemented.

#### 3.4.3.3 Experimental results

Control experiments were conducted to validate the feasibility of integrating the Kalman filter with the one-shot task model and to evaluate the different implementation approaches. First, it is shown that the Kalman filter and the one-shot task model preserve their benefits when both are integrated in a control loop. Then, the five different implementation approaches, specified in Subsection 3.4.1 are evaluated in terms of control performance. Finally, the resource demand of the half and complete Kalman implementation are analyzed.

Experimental results shows that Kalman filter effectively removes the noise from the measured signals. Figure 3.6 compares the noisy captured data from the plant (top) with the estimated states obtained with the half Kalman filter (bottom). Similar results are found with the complete Kalman. Note that reference changes use small values ($-0.2$V to $0.2$V) to appreciate the noisy signal.

Now, lets consider the case where the controller task has the presence of random timing variations in the form of scheduling jitters. Jitters produce irregular sampling periods ranging from 0 to $20ms$. The system response of the half Kalman one-shot controller is compared with the standard naif controller in order to assess whether embedding the Kalman filter jeopardizes the benefits of the one-shot task model in removing the jitters effects. As it can be seen in Figure 3.7, the con-

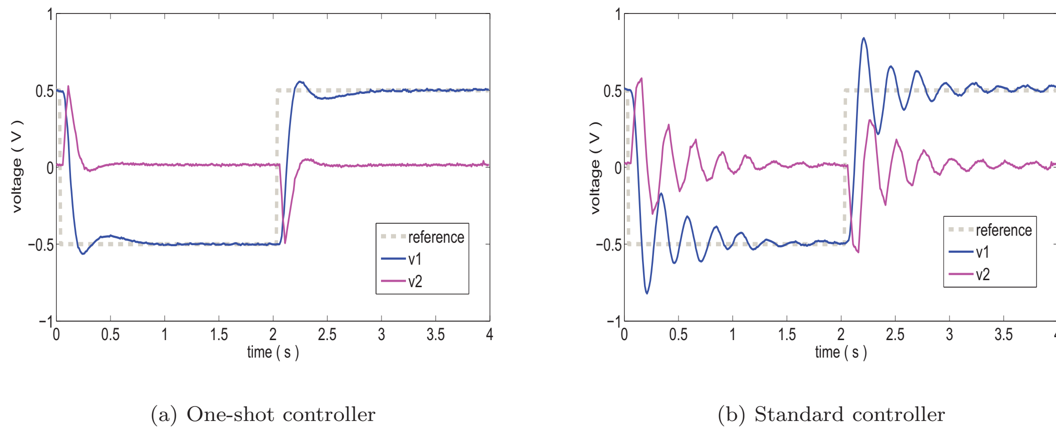



(a) Noisy signals from plant

(b) Kalman estimated state values

Figure 3.6: Removing noise with the Kalman filter

trol performance of the standard controller is considerably degraded while the one-shot controller achieves better performance.




(a) One-shot controller

(b) Standard controller

Figure 3.7: Controllers response with jitters.

The Kalman gain values during the previous experiment (with jitters) were obtained in order to certificate the correct implementation of the Kalman filter. Figure (3.8) shows the evolution of the first element of the Kalman gain for the half and complete approaches compared with the Kalman filter's gain using a standard controller with no jitters. It can be noticed that the values are similar despite of small variations for the half and complete approaches.

#### 3.4.3.4 Experimental performance evaluation

The different implementation approaches specified in Subsection 3.4.1, were evaluated in terms of control performance. For each approach, performance was measured with a discrete-time control

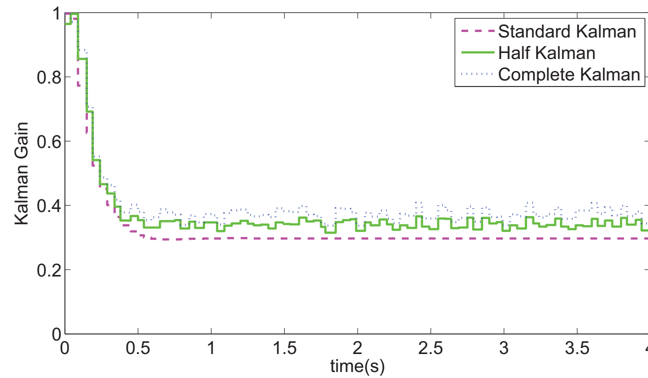



Figure 3.8: Kalman gain evolution.

Table 3.2: Experimental control performance.

| Implementation Approach | Control Performance |
|---|---|
| (A) Kalman with standard controller (no jitters) | 4.4517 |
| (B) Half Kalman with one-shot controller | 4.4523 |
| (C) Complete Kalman with one-shot controller | 4.4542 |
| (D) Kalman with standard controller | 5.0645 |
| (E) No Kalman with standard controller | 7.5359 |

cost equivalent to the continuous cost defined in (3.23).

A set of ten different experimental scenarios were elaborated in order to cover a wide variety of system conditions. Each scenario considers different jitters values, and different set-points (reference) amplitudes and frequencies. The same set of scenarios was applied to each approach, with the exception of the first approach (A) where no jitters were applied, since this implementation approach serves as a reference (ideal case) for the experimental evaluation. Average values of the ten scenarios in terms of cost (smaller values means better performance) are presented on Table 3.2 for a 40s evaluation period.

The results shows that the half Kalman (B) and the complete Kalman (C) implementations using the one-shot controller has no meaningful differences in their performance. And both approaches have practically the same performance as the ideal case (A), even when (B) and (C) includes jitters. Now, the Kalman filter implementation in the standard controller (D) has a worse performance compared with (B) and (C) as expected because jitters affects its performance. Finally, it is interesting to notice that if a standard controller is used without Kalman (E), the jitters degrading effect is greater than the one obtained with the use of Kalman (D).

It is important to highlight that experimental results (Table 3.2) and simulation results (Table 3.1) present the same trend regarding the control performance evaluation of the different approaches. However values in simulation are slightly but consistently lower than in experiments, even when noise covariance is higher in simulation. This may be caused in part, by the fact that simulation measurements for performance are obtained directly from the plant (noise-free) meanwhile experiments measurements are affected by noise.

Finally, from the control performance perspective the half Kalman and the complete Kalman are similar. Now, if the processor workload is taken in consideration, by measuring the task execution time of each algorithm, then it is obtained that the complete Kalman algorithm requires more time compared with the half approach. Figure 3.9 shows the accumulated execution time during a

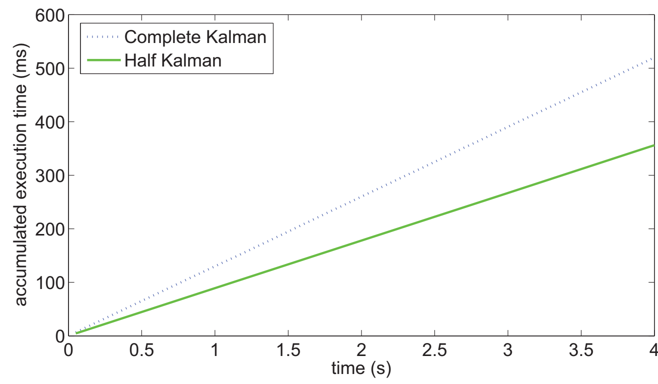



Figure 3.9: Accumulated execution time.

period of 4s for the half and complete Kalman implementations. As it can be noticed half Kalman spends less execution resulting in a simpler and less processor intensive implementation approach.

# Chapter 4

# Taxonomy on resource/performance-aware policies

## 4.1 Introduction

The execution rate of controllers determine control performance as well as resource utilization. Selecting sampling periods is not an easy task when resources are limited. The real-time and control communities have provided diverse theoretical results on both control and resource optimization for resource limited computing systems concurrently executing several controllers. Loosely speaking, most of these results suggest to efficiently select the controllers' sampling periods, such that controllers' execution rates are different from those provided by the standard periodic sampling approach [ÅW97].

As indicated in Chapter 1, within these results, two main disciplines can be identified: feedback scheduling (FBS), and event-driven control systems (EDC). This chapter reviews these results. The outcome is a taxonomy on resource/performance-aware policies for embedded control systems.

## 4.2 Problem set-up

This section reviews existing results for FBS and EDC, while identifying key features. The main goal is to be able to construct a taxonomy, that will permit a better definition of the evaluation framework presented in next chapter. For feedback scheduling approaches, the reviewed results are [SLSS96], [SLS98], [EHÅ00], [RS00], [HCAÅ02], [PPSV+02], [CEBÅ02], [CLS03], [MLB+04], [PPBSV05], [HC05], [CMV+06], [GCHI06], [MLB+09], [BC08], [SCEP09], [SEPC09], [GcH09], and [CVMC10]. For event-driven control, the reviewed results are [Årz99], [HGvZ+99], [ZZ99], [AB02], [VMF03], [TW06], [Mis06], [Tab07], [LCH+07], [JHC07], [SNR07], [AT08a], [AT08b], [WL08a], [WL08b], [HSB08], [HJC08], [WL09a], [WL09b], [MVB09], [MAT09], [MT09], [VMB09a], [AT09], and [AT10]. In addition, the survey also will focus on which type of evaluation was applied to each method, information that will be also used in next chapter.

### 4.2.1 Feedback scheduling

Initial research on feedback scheduling is found in Seto et *al.* [SLSS96], where an off-line algorithm was proposed to select tasks sampling periods. The performance of each controller was captured by a cost function that describes the relationship between the sampling rate and the quality of control. Assuming that the cost vs rate for each controller can be approximated by an exponentially decreasing function, the paper gives an optimization algorithm that assigns optimal sampling rates to the controllers, subject to an EDF CPU utilization constraint. This concept was extended to RM scheduling in [SLS98]. Although both approaches were described in detail, only simulations were provided for validation purposes. A further extension of this off-line approach to redundant controllers was presented in [CLS03], using the same simulated scenarios (temperature control, bubble control, and inverted pendulum) over the Simplex [SKSC98] software platform. Bini *et al.* [BC08] extended these type of off-line approaches considering also, apart from sampling periods, input-output delays. The approach was validated using simulated workloads and randomly generated delays.

An off-line optimal period assignment method for a set of state feedback controllers using the stability radius as a performance criterion was presented by Palopoli et *al.* in [PPSV+02] and [PPBSV05]. Evaluation was performed on simulated inverted pendulums using the Erika kernel.

Rehbinder et *al.* [RS00] presented an off-line approach based on cyclic executives. Therefore the solution to the optimization problem was expressed in terms of sequences of tasks instead of the sampling periods' values. The proposed approach uses the predicted error over a finite time horizon as the information required to define the optimal sequences of tasks. The approach was validated using simulations on inverted pendulums.

Recently, Samii *et al.* [SCEP09] [SEPC09] proposed an off-line approach to scheduling and synthesis of control applications where the output of the algorithm is the schedule that minimizes the cost for a given number of controllers. Simulations were used to validate the approach, and measurements in a PC running Linux was used for complexity analysis.

An important change with respect to previous works has been the use of an on-line scheduler that uses feedback to dynamically adjust the control task attributes in order to optimize the global control performance. This feedback scheduling concept was used initially in [EHÅ00] and further developed in [CEBÅ02]. Eker *et al.* [EHÅ00] considered linear quadratic (LQ) state feedback controllers and derived expressions relating the LQ-cost to the sampling interval. The optimization process was based on the *a priori* relation between cost and the sampling periods. A linear approximation was used to represent the cost-period relation. They also proposed an on-line optimization algorithm for sampling period assignment, that iteratively adjusts the sampling rates based on the CPU load. An extension to general linear dynamic controllers is found in [CEBÅ02]. In both approaches the scheduler attempts to keep the processor utilization as close as possible to a utilization set-point by manipulating the sampling periods. The proposed optimization algorithms were implemented in TrueTime. Four simulated inverted pendulum were used as the controlled plants. The same idea but targeting model predictive controllers was presented by Henriksson *et al.* [HCAÅ02], which was validated using TrueTime simulations on a quadruple-tank laboratory process.

Martí *et al.* [MLB+04] introduced feedback from the actual control performance. The scheduler obtains feedback information from the controlled process. The sampling period reassignment is based on the current instantaneous cost measurement from the controlled plant. This approach took as the underlying idea the fact that a controlled plant in a transient phase, caused by an external disturbance, may require more resources (that is shorter sampling periods), than a controlled plant in a steady state free from disturbances. The algorithm was implemented in a Linux 2.4.20 kernel-based platform (named RBED [BBLB03]) and extensive experiments were performed on simulated inverted pendulums. The same authors presented an extension [MLB+09] where the

relation between its cost function and standard quadratic cost functions was established.

Henriksson *et al.* [HC05] further formalized this approach for linear quadratic controllers by incorporating the current plant states into a finite-horizon quadratic cost function, which also took the expected future plant noise into account. The extension to the general case of linear controllers was given in [CMV+06]. Both feedback scheduling strategies were simulated with TrueTime using three different second-order plants: ball and beam, DC motor, and harmonic oscillator. A further improvement is offered by Cervin *et al.* [CVMC10] where the performance of each controller is captured in a finite horizon cost function, taking into account the sampling period, the computational delay, and the amount of noise acting on the plant. In addition, the cost function was developed for general linear dynamic controllers (and not only for state feedback controllers as in much of previous work). A proof-of-concept implementation, where the feedback scheduling approach is put to test in a real system, was presented based on the Erika real-time kernel and electronic double integrator circuits as the controlled plants.

Ben Gaid *et al.* [GCHI06], [GcH09] presented a complementary approach for the optimal integrated control and real-time scheduling of control tasks. It combined non-preemptive optimal cyclic schedules according to the $H_2$ performance criterion, with an efficient on-line scheduling heuristic in order to improve responsiveness to disturbances. The validation of these approaches was performed using simulated inverted pendulums.

### 4.2.2 Event-driven control

Initial research on event-based scheduling is found in Arzen [Årz99] where the integration of an analog event detector was proposed, in order to trigger a PID controller. Simulations were conducted in order to compare the control performance and the processor load of the event-based controller with a time-trigger controller. As a result the event-based controller achieved large reductions in the processor load with minor performance degradation. Processor load was measured as a percentage of utilization, and a simulated double-tank process was used as a plant.

In Heemels *et al.* [HGvZ+99], a control system to synchronize the position of two motors based on asynchronous measurements was presented. In this approach an external interruption is received by the controller in order to measure the error between the motors. The proposed design, for this first order system, was implemented and tested with real inductor motors, using a Texas Instrument digital signal processor (TMS-320C40).

In Zhao *et al.* [ZZ99], an event feedback strategy was suggested where the scheduling policy chooses one and only one plant among $N$ plants to be controlled at any time. Specific conditions of asymptotical and exponential stability are then given and an exponential upper bound of states norm is estimated for the event-based strategy. An algorithm based on event feedback was presented to determine the control laws of the plants in order to meet the performance specifications. Simulation study showed that the proposed strategy has a better performance with a sequential scheduling policy. The control performance was compared by analyzing the transient response.

Event-based control for a first-order stochastic system was studied in [AB02]. It was shown that an event-based controller for an integrator plant disturbed by white noise requires on average, only one fifth of the sampling rate of an ordinary, periodic controller to achieve the same output variance. Simulations were carried out to validate the approach and control performance was measured in terms of output variance.

Miskowicz [Mis06] proposed a data collection strategy where the sampling action is triggered if the signal deviates by delta, defined as a significant change of its value in relation to the most recent sample. This schema was targeted for wireless sensor networking due to the need of effective energy consumption. Analytical solutions were presented for first and second order systems. A similar approach was proposed by [SNR07], where a modified Kalman algorithm was implemented to reduce sensor data traffic with relatively small estimation performance degradation. Experiments

were conducted on sensor boards (Amtel AT90CAN128) connected through a CAN network, a third order plant was simulated in a PC with Matlab real-time workshop. The performance was measured by the number of data transmission by each sensor.

In Heemels *et al.* [HSB08] an event-driven control scheme was presented where the the control update is only triggered when the tracking or stabilization error is large. In this manner, the average processor and communication load can be reduced significantly.

An sporadic control schema consists in an aperiodic event-based control with a specified minimum inter-event time. In Johannesson *et al.* [JHC07] proposed a scheduling schema that reduces the average frequency of control events and also the variance of the system state. In this approach, it is assumed that the process state is measured continuously and that a control action can be taken at any point in time, but not more often than the minimum inter-arrival time. The performance of the system was measured by a cost function with two terms: the state cost represents the stationary process variance and the control cost represents the average number of events per time unit. Simulations on this approach were conducted over first order systems. In Henningsson *et al.* [HJC08] an sporadic controller was compared with a periodic controller in a first order lineal system. The performance was measured by the stationary state cost (state error) and by the number of control actions within an interval (control rate).

In self-triggered systems, the control task determines its next release time based on samples of the state gathered at the current release time. Velasco *et al.* [VMF03] presented a self-triggered task model that drives control task executions according to controlled system performance and available processing capacity. The model, which extended the original state space representation of a controlled plant with the control task period as an additional state variable, allowed control task to adjust their execution rate acting as a co-scheduler. Simulations were conducted using two ball and beam plants to analyze the system dynamics. Martí *et al.* [MVB09] formulated the optimal boundary and regulator design problem that minimizes the resource utilization of an event-driven controller that achieved a cost equal to the case of periodic controllers. The standard quadratic cost function was used to specify the optimization problem. Simulations were conducted using a double integrator plant model.

Tabuada [Tab07] presented a self-triggered schema where the decision to execute the control tasks was determined by a feedback mechanism based on the state of the plant. Simulation for this approach were conducted to verify the system stability, where system error and system state values were measured. A second-order plant was used during simulation. Although it was not addressed, the paper mentioned that the proposed strategy requires special purpose hardware to trigger the event. This approach was further analyzed by Anta *et al.* in [AT08a], [AT08b], [AT09], [AT10] where the self-trigger model was extended for non-linear systems.

In Lemmon *et el.* [LCH$^+$07], the self-triggered mechanism was implemented using the elastic scheduling [BLA98]. In the elastic task model or elastic scheduling the deadline misses are avoided by increasing tasks periods until some desirable utilization level is achieved. Simulations were conducting over three inverted pendulums plants, mainly to compare the proposed approach with a time-triggered approach with similar characteristics. Control performance was evaluated by analyzing their respective transient response. This work was extended by Wang *et al.* in [WL08b], [WL08a], [WL09a], [WL09b]. In [WL08b], [WL09a] where bounds were derived on a task's sampling period and deadline to quantify how robust the control system's performance is with respect to variations in these parameters. An inverted pendulum model was used to simulate the system plant, and the normalized state error was used for performance measurement.

In Mazo *et al.* [MAT09] a general procedure leading to self-triggered implementations of feedback controllers was proposed. The approach was simulated with a fourth order batch reactor model as the controlled plant, and the performance was measured by the size of the inter-execution times. In [MT09] the proposed approach was analyzed to study its robustness with respect to disturbances.

In Velasco *et el.* [VMB08] it was provided a general explicit approximated solutions to compute activation times for event-driven control jobs, and extends both the FP and EDF schedulability analysis to the control-driven tasks. For the analysis of activation patterns and the comparison of the event-based control with the time-triggered control, simulations were conducted using a ball and beam plant.

### 4.2.3 Common formulations for FBS and EDC

The considered scenario consists on $n$-control loops competing for processor time. Each control loop contains the controller characterized by an state feedback gain $L^i$ and the controlled plant, which can be modelled by the linear continuous time state-space representation

$$\begin{array}{rcl} \dot{x}^i(t) & = & A^i\,x^i(t) + B^i\,u^i(t) \\ y^i(t) & = & C^i\,x^i(t) \end{array} \tag{4.1}$$

with $x^i \in \mathbb{R}^{n\times 1}$, $A^i \in \mathbb{R}^{n\times n}$, $B^i \in \mathbb{R}^{n\times m}$, $u^i \in \mathbb{R}^{m\times 1}$, and $C^i \in \mathbb{R}^{1\times n}$. Let

$$u^i(t) = u^i_k = L^i\,x^i(a^i_k) = L^i\,x^i_k \qquad \forall t \in [a^i_k, a^i_{k+1}[ \tag{4.2}$$

be the control updates given by each linear feedback controller $L^i$ using only samples of the state at discrete instants $a^i_0, a^i_1, \ldots, a^i_k$. Between two consecutive control updates, $u^i(t)$ is held constant. In periodic sampling $a^i_{k+1} = a^i_k + h^i$ where $h^i$ is the period of the controller. The controller execution time is given by $c^i$.

In most of the FBS cases, the feedback gain $L^i$ is designed as mandated by each method in the discrete time domain considering

$$\begin{array}{rcl} x^i_{k+1} & = & \Phi^i(h^i)\,x^i_k + \Gamma^i(h^i)\,u^i_k \\ y^i_k & = & C^i\,x^i_k \end{array} \tag{4.3}$$

with $\Phi^i(t) = e^{A^i t}$ and $\Gamma^i(t) = \int_0^t e^{A^i s}\mathrm{d}sB^i$, and where $h^i$ may vary following different patterns. On the contrary, for EDC methods, the feedback gain is often designed in the continuous time domain.

Regardless of the design procedure for the feedback gain, all the controllers are characterized by the sampling interval $h^i$ that will vary according to the particular approach resource/performance-aware policy. For the feedback scheduling approaches, in general, each task is associated a cost function $J^i(h^i)$, which gives the control cost (or benefit) as a function of the sampling interval. Then, $h^i$ is the solution of solving

$$\text{minimize} \qquad \sum_{i=1}^{n} J^i(h^i) \qquad \text{w.r.t. } h^i \tag{4.4}$$

$$\text{subject to} \qquad \sum_{i=1}^{n} \frac{c^i}{h^i} \leq U_{ref} \tag{4.5}$$

where $U_{ref}$ is the desired resource utilization level for the set of control loops. The optimization problem is constrained by two key aspects: the set of optimal sampling periods must guarantee closed loop stability and task set schedulability. Stability is either guaranteed by the formulation of the optimization problem, or it is not explicitly imposed in the formulation but analyzed after solving the optimization problem. Task set schedulability is often imposed by resource utilization tests. A few methods, instead of providing optimal sampling periods, provide job sequences. That is, the outcome of the optimization problem is an optimal sequence of jobs for each control task to be executed periodically. In several approaches, the outcome of the optimization problem also

includes the gains $L^i$.

In EDC, the controller is activated upon some condition on the system status and not periodically. The condition called even-condition or execution rule, mandates to take a new control action when the system state variables (or measured signals) have deviated sufficiently from the set-point. A common formulation is as follows. For the EDC approaches, the variation on the sampling interval is given by $h^i = \Lambda^i(x_k^i, \Upsilon^i, \eta^i)$, where $\Lambda(\cdot)$ is the time spent by each closed loop trajectory from the sampled state $x_k^i = x(a_k^i)$ to reach the given boundary. Boundaries can be described by

$$f^i(e_k^i(t), x_k^i, \Upsilon^i) \leq \eta^i \tag{4.6}$$

where $\Upsilon^i$ is a set of free parameters of $f^i$, $\eta^i$ is the error tolerance, and $e_k^i(t) = x^i(t) - x_k^i$ is the error evolution between consecutive samples with $t \in [a_k^i, a_{k+1}^i[$. Therefore, the complete dynamics of the each event-driven system is given by

$$\begin{array}{rcl} a_{k+1}^i & = & a_k^i + \Lambda^i(x_k^i, \Upsilon^i, \eta^i) \\ x_{k+1}^i & = & (\Phi^i(\Lambda^i(x_k^i, \Upsilon^i, \eta^i)) + \\ & & \Gamma^i(\Lambda^i(x_k^i, \Upsilon^i, \eta^i))L^i)x_k^i. \end{array} \tag{4.7}$$

Velasco *et el.* [VML08] remarks that to find an expression for $\Lambda^i(x_k^i, \Upsilon^i, \eta^i)$ is sometimes feasible by approximating $\Phi$ and $\Gamma$ by Taylor expansion. An alternative technique for finding $\Lambda^i$ under several assumptions is given by [WL09a]. Otherwise, $\Lambda^i$ can only be computed numerically, according to the particular formulation given in each approach, or approximated. Whenever finding $\Lambda^i$ is feasible, the event-driven control scheme can be implemented as a self-triggered approach: each controller execution computes when the next execution should occur in time. Otherwise, the event-driven control scheme must be implemented using dedicated hardware for detecting the event condition.

## 4.3 Taxonomy

This section presents a taxonomy on the resource/performance-aware policies from the approaches presented in the previous section. The taxonomy reveals key characteristics and tendencies on embedded control systems which support the definition of the performance evaluation framework specifications. A preliminary taxonomy can be found in [LVM07].

### 4.3.1 Methods

Many of the novel methods go beyond than just finding the *best* values for control task periods. They provide complete real-time frameworks tailored to effective concurrent execution of control tasks. They can be characterized by *which* criterion is used to select sampling periods, thus establishing *what* real-time paradigm is demanded in the underlying executing platform, *who* should decide which task to execute, *when* the decision is taken, *where* the dynamics accounted for are located, and *how* the decision is enforced. By reviewing the surveyed resource/performance-aware policies, a taxonomy analysis is summarized in Table 4.1.

### 4.3.2 Criterion

A key aspect of these policies is the theoretical criterion used to obtain the set of sampling periods (or sequences or schedules). Two main criterion can be identified: optimization approach or bounding the inter-sampling dynamics.

In the optimization approaches sampling periods are selected to solve an optimization problem. They assume that there is a cost function parameterized in terms of control performance and sampling periods that has to be minimized or maximized depending on whether it denotes penalty or benefit. The optimization problem domain is restricted by closed loop stability and task set schedulability constraints. For example, in Seto et *al.* [SLSS96] sampling periods are statically assigned to each control loop in order to obtain the best overall performance. In Martí *et al.* [MLB+04] shorter sampling periods are dynamically assigned to the plant with the current largest error (cost function).

In the approaches based on bounding the inter-sampling dynamics, sampling periods are selected to keep each closed loop dynamics within predefined thresholds. Thresholds, which are derived from pure control theoretical approaches or hardware set-up (measurements methods), are used to bound changes in the dynamics or to ensure closed loop stability in different forms. For example, in Arzen [Årz99] an external hardware interruption triggers the task jobs, meanwhile in Lemmon *et el.* [LCH+07] the task thresholds definitions are specified in order to ensure system stability.

Usually, many of the feedback scheduling approaches use the optimization criteria while the event-driven control systems approaches use the bounding in the inter-sampling dynamics criteria.

### 4.3.3 Triggering paradigm and entity

These two categories influence whether the period selection solution requires a real-time architecture that follows a time-triggered (TT) or an event-triggered (ET) paradigm. All the solutions to the optimization approaches require a time-triggered architecture while all the solutions based on bounding closed-loop dynamics require an event-triggered architecture.

The classification considers who is in charge of selecting sampling periods (triggering entity). All solutions requiring a TT architecture are based on a global coordinator that decides the *best* periods for the set of control tasks. This can be in the form of a special purpose task acting as a feedback scheduler (e.g. [MLB+04], [HC05]) or it can be implemented directly into the real-time kernel (e.g. [EHÅ00]). On the contrary, in the solutions requiring an ET architecture, two options can be identified: control tasks are in charge of deciding their periods (self-triggered approaches e.g. [VMF03], [LCH+07]) or specific purpose hardware must be used to detect the event condition that will trigger each control task (e.g. [Årz99], [HGvZ+99]).

### 4.3.4 Solving the problem

The previous classification (TT vs. ET) relates to whether the period selection is performed off-line or on-line. In all ET approaches periods are derived on-line. However, in the TT approaches, some solutions have to be computed off-line (e.g. [SLSS96]) while others are on-line (e.g. [MLB+04], [HC05])

It is important to identify when the sampling periods are selected for two main reasons: overhead and adaptability. On-line algorithms introduce computational overhead which may be considered a disadvantage, some approaches propose the use of look-up tables to reduce the overhead, e.g. [CMV+06]. On the other hand, on-line algorithms have the ability to adapt to workload changes by either varying the available resources [EHÅ00] or varying demands from the control applications [MLB+04], this adaptability feature can be considered an advantage for on-line approaches.

Table 4.1: Taxonomy of resource management approaches.

| | Which<br>Criterion | What<br>Triggering<br>Paradigm | Who<br>Triggering<br>Entity | When<br>Solving<br>the problem | Where<br>Dynamics | How<br>Solution |
|---|---|---|---|---|---|---|
| Set96 [SLSS96] | Optimizat. | TT | Coord. | Off-line | None | Static periods |
| Set98 [SLS98] | Optimizat. | TT | Coord. | Off-line | None | Static periods |
| Arz99 [Årz99] | Bound dyn. | ET | Task | On-line | Event detector | Ext. interrupt |
| Hee99 [HGvZ+99] | Bound dyn. | ET | Task | On-line | Event detector | Ext. interrupt |
| Zha99 [ZZ99] | Bound dyn. | ET | Coord. | On-line | Plant state | Sequences |
| Eke00 [EHÅ00] | Optimizat. | TT | Coord. | On-line | Kernel | Varying periods |
| Reh00 [RS00] | Optimizat. | TT | Coord. | Off-line | None | Sequences |
| Ast02 [AB02] | Bound dyn. | ET | Task | On-line | Plant state | Time intervals |
| Vel03 [VMF03] | Bound dyn. | ET | Task | On-line | Plant meas. state | Time intervals |
| Hen02 [HCAÅ02] | Optimizat. | TT | Coord. | On-line | Kernel | Varying periods |
| Cer02 [CEBÅ02] | Optimizat. | TT | Coord. | On-line | Kernel | Varying periods |
| Pal02 [PPSV+02] | Optimizat. | TT | Coord. | Off-line | Plant stability | Static periods |
| Cha03 [CLS03] | Optimizat. | TT | Coord. | Off-line | None | Static periods |
| Mar04 [MLB+04] | Optimizat. | TT | Coord. | On-line | Plant (instant.) | Varying periods |
| Hen05 [HC05] | Optimizat. | TT | Coord. | On-line | Plant (finite hor.) | Varying periods |
| Pal05 [PPBSV05] | Optimizat. | TT | Coord. | Off-line | Plant stability | Static periods |
| Ben06 [GCHI06] | Optimizat. | TT | Coord. | On-line | Plant (finite hor.) | Sequences |
| Cas06 [CMV+06] | Optimizat. | TT | Coord. | On-line | Plant (finite hor.) | Varying periods |
| Mis06 [Mis06] | Bound dyn. | ET | Task | On-line | Event detector | Ext. interrupt |
| Tab07 [Tab07] | Bound dyn. | ET | Task | On-line | Plant state | Time intervals |
| Lem07 [LCH+07] | Bound dyn. | ET | Task | On-line | Plant meas. state | Time intervals |
| Joh07 [JHC07] | Bound dyn. | ET | Task | On-line | Plant state | Sporadic |
| Suh07 [SNR07] | Bound dyn. | ET | Task | On-line | Event detector | Ext. interrupt |
| Hen08 [HJC08] | Bound dyn. | ET | Task | On-line | Plant state | Sporadic |
| Ant08 [AT08a] | Bound dyn. | ET | Task | On-line | Plant state | Time intervals |
| Ant08a [AT08b] | Bound dyn. | ET | Task | On-line | Plant state | Time intervals |
| Bini08 [BC08] | Optimizat. | TT | Coord. | Off-line | None | Periods-delays |
| Hee08 [HSB08] | Bound dyn. | ET | Task | On-line | Plant meas. state | Time intervals |
| Wan08 [WL08b] | Bound dyn. | ET | Task | On-line | Plant meas. state | Time intervals |
| Wan08a [WL08a] | Bound dyn. | ET | Task | On-line | Plant stability | Time intervals |
| Ant09 [AT09] | Bound dyn. | ET | Task | On-line | Plant state | Time intervals |
| Ben09 [GcH09] | Optimizat. | TT | Coord. | On-line | Plant (finite hor.) | Sequences |
| Mar09 [MVB09] | Bound dyn. | ET | Task | On-line | Plant meas. state | Time intervals |
| Mar09a [MLB+09] | Optimizat. | TT | Coord. | On-line | Plant (instant.) | Varying periods |
| Maz09 [MAT09] | Bound dyn. | ET | Task | On-line | Plant state | Time intervals |
| Maz09a [MT09] | Bound dyn. | ET | Task | On-line | Plant state | Time intervals |
| Sam09 [SCEP09] | Optimizat. | TT | Coord. | Off-line | Schedule | Periods/sequences |
| Sam09a [SEPC09] | Optimizat. | TT | Coord. | Off-line | Schedule | Periods/sequences |
| Vel09 [VMB09a] | Bound dyn. | ET | Task | On-line | Plant meas. state | Time intervals |
| Wan09 [WL09a] | Bound dyn. | ET | Task | On-line | Plant meas. state | Time intervals |
| Wan09a [WL09b] | Bound dyn. | ET | Task | On-line | Plant meas. state | Time intervals |
| Ant10 [AT10] | Bound dyn. | ET | Task | On-line | Plant state | Time intervals |
| Cer10 [CVMC10] | Optimizat. | TT | Coord. | On-line | Plant (finite hor.) | Varying periods |

### 4.3.5 Dynamics

It refers to where the dynamics that are accounted for in the optimization problem are located: kernel (resource) and/or plant. Pure kernel dynamics are considered only in few approaches ( [EHÅ00], [HCAÅ02], [CEBÅ02]). In the case where the plant defines the optimization process, there are different considerations: [MLB+04] considers the plant instantaneous error, [HC05] considers the plant finite horizon dynamics, the event-based methods require an event-condition that can be defined in terms of the plant state [TW06], the measured state [LCH+07], or implemented via an external hardware interrupt [Årz99]. Other optimization approaches focuses primarily in improving the robustness of the controlled system [PPSV+02], or selecting adequate schedules that exploit the available computation resources to optimize the control performance in the running mode [SEPC09].

### 4.3.6 Solution

Once periods are selected based on a specific dynamics, they must be enforced by the underlying real-time architecture. Therefore, it is important to examine how the solutions are enforced. Although the taxonomy reviews methods for sampling period selection, some methods ( [RS00], [GCHI06], [GcH09]) do not establish sampling periods, rather they provide periodic sequences of ordered control task instances. All the others provide periods in different forms, i.e. static periods ( [SLS98]), varying periods ( [EHÅ00], [MLB+04], [HC05]), self-triggered aperiodic time intervals ( [TW06], [LCH+07]), and aperiodic time intervals caused by external interruptions ( [Årz99], [HGvZ+99]).

All solutions demanding a time-triggered architecture can enforce the derived timing constraints for control tasks using well known scheduling strategies such as earliest deadline first (EDF) and fixed priority (FP). For the solutions demanding an event-based architecture, the scheduling policy, that can enforce the presented solution, is lacking in the general case. Only the result provided in [LCH+07] integrates the presented even-triggered control with existing scheduling theory. At each job execution the deadline for the following job is predicted and the elastic scheduling is invoked to accommodate the new timing demands, considering the whole task set. However, if the elastic scheduling can not meet them, problems may occur.

### 4.3.7 Discussion

By analyzing the presented taxonomy, the following considerations need to be included in the specifications of any real-time control systems evaluation framework:

- According with the optimization criterion, control performance and resource utilization represent the two most important evaluation parameters. Hence, the performance evaluation framework must be able to measure both parameters.
- The triggering paradigm indicates that both event-based and time-trigger architectures must be supported by the evaluation framework.
- There is a clear tendency to implement on-line optimization algorithms. This tendency reflects and aims at meeting the demands of modern embedded systems that are required to work in dynamic environments, being adaptive to the available resources that can change abruptly, or to the resource demands of control applications that can be considered as varying depending on the state of the controlled plants. Therefore, the evaluation framework must provide proper services to allow the execution of on-line optimization algorithms according to either kernel and/or plant dynamics. Also workload changes must be allowed and computation overhead should be included as a measurement parameter.

| Approach | Trigger | When | Dynamics | Exec. Rule |
|---|---|---|---|---|
| Static approach [ÅW97] | TT | | | |
| Off-line FBS [SLSS96] | TT | Off | | |
| On-line FBS-Inst. [MLB+04] | TT | On | kernel/plant | |
| On-line FBS-FH [HC05] | TT | On | kernel/plant | |
| Heuristic Self-triggered [VMF03] | ET | On | kernel/plant | state/utilization |
| Self-triggered [LCH+07] | ET | On | kernel/plant | meas. state |
| Optimal self-triggered [MVB09] | ET | On | kernel/plant | meas. state |

Table 4.2: Selected methods of FBS and EDC showing key distinctive features.

- In order to cover a wide variety of approaches, static and varying periods, as well as sequences and aperiodic time intervals must be supported by the evaluation framework.
- Scheduling policies such as EDF and FP must be part of the services provided by the framework.

## 4.4 Selected methods for performance evaluation

Considering the previous taxonomy, this section presents which subset of the resource/performance-aware policies will be evaluated. The subset of selected methods represent major tendencies identified in the taxonomy. They are summarized in Table 4.2 and characterized by the following parameters:

- Triggering paradigm: time-triggered (TT) for feedback scheduling or event-triggered (ET) for event-based scheduling.
- When to solve the optimization problem: off-line or on-line.
- Which kind of dynamics are accounted for in the optimization problem: resource (kernel) and/or plant.
- Execution rule: event condition can be defined as a function of the system state or the measured state (applies only for EDC methods).

### 4.4.1 Static Approach

This is the only approach that does not belong to the class of feedback scheduling nor event based scheduling methods but it is here included for comparative purposes. It implements the traditional approach to real-time implementation of computer controlled systems. That is, each control task is assigned off-line an *arbitrary* sampling period (time-triggered) selected according to well established procedures [ÅW97], taking also into account task set utilization.

### 4.4.2 Off-line FBS (feedback scheduling)

The off-line FBS is represented by the work by [SLSS96] which can be considered one of the seminal papers on sampling period selection subject to control performance optimization for real-time

control systems. An off-line optimization is performed in order to reduce control cost, once the optimal periods are calculated, the control tasks are scheduled under EDF. Although different on their formulations of the optimization problem in terms of objective functions and restrictions, existing results such as [SLS98], [RS00], [PPSV+02], [CLS03], [PPBSV05], [BC08], [SCEP09], [SEPC09] can be included in a subset of works that share in common that they are off-line feedback scheduling methods. That is, sampling periods are derived before run-time and kept constant during execution.

### 4.4.3 On-line FBS (feedback scheduling)

The method presented by [EHÅ00], further developed in [HCAÅ02] and [CEBÅ02], is the first one that uses the term *feedback scheduler*. The key aspect presented in [EHÅ00] is to on-line adjust sampling periods considering the dynamics of the processor load. Looking at the outer loop of the resource manager is the feedback scheduler that, having available the system workload from the real-time kernel, i.e. *resource aware* (RA) and given a utilization set-point, keeps the desired utilization by modifying workload via on-line sampling period selection, while optimizing the total control performance. A step further is also to optimize control performance by on-line adjusting sampling periods according to both kernel workload and plant dynamics, idea introduced by [MLB+04]. The intuitive idea behind this kind of approaches is to provide more processing capacity to control tasks whose plants are experiencing severe transients due to e.g. perturbations or noise.

Within the existing work that consider on-line period adjustment that accounts for plant dynamics and kernel workload, two main flavors can be distinguished depending on whether decisions are taken looking an instantaneous plant states or looking at predictions of the plant dynamics using for example a finite horizon cost function. The work by [MLB+04] will be taken in the performance evaluation as example policy whose decisions rely on an instantaneous metric. This work was further refined in [MLB+09]. The work by [HC05] will be taken in the performance evaluation as example of policy whose decisions rely on a finite horizon metric. This work was further developed in [CMV+06] and [CVMC10]. This type of approach was also adopted in [GCHI06] and [GcH09].

### 4.4.4 Heuristic self-triggered

This method is based on event-driven control and it is represented by the work by [VMF03], where a self-triggered control approach is implemented to heuristically optimize control performance and resource utilization. The key idea is that at each job activation, the software tasks select themselves the next job release time according to a specific execution rule. Discrete-time controller gains are adapted to each new sampling interval. In this approach the event condition or execution rule is defined as a function of the system state and processor utilization.

### 4.4.5 Self-triggered

This method is based on event-driven control and it represented by the work by [LCH+07], where a self-triggered scheme is presented based on a robust control formulation. In this approach the event condition or execution rule is defined only as a function of the system state and the controller state is invariant, and designed in the continuous time domain. This approach was further developed in [WL08a], [WL08b], [WL09a], and [WL09b]. Similar event-driven control approaches that can be represented under this category are [TW06], [Tab07], [AT08a], [AT08b], [MAT09], [MT09], [VMB09a], [AT09] and [AT10]. Note that some of them do not present the event-driven approach formulated as a self-triggered method. But the key aspect is that they can be transformed to self-triggered, and more important, that sampling intervals are defined as a function of the system

state. However, it is also clear that each method will provide different performance numbers in case of being evaluated.

### 4.4.6 Optimal self-triggered

This method is based on event-driven control, and is an enhancement of [LCH+07] that was presented by [MVB09]. The key aspect is that the different controller settings such as next sampling interval and controller gains are a result of an optimization algorithm that is executed at each controller execution. In any case, optimal parameters are selected according to the plant dynamics and a standard quadratic cost function.

### 4.4.7 Other EDC approaches

Apart from the previous selected methods for evaluation, there are still different event-driven approaches such as [Årz99], [HGvZ+99], [ZZ99], [AB02], , [Mis06], [JHC07], [SNR07], [HSB08], and [HJC08] that could be implemented. However, many of them rely on specific hardware to detect event conditions and their transformation to a self-triggered approach should be carefully analyzed. In addition, many of them trigger the event condition based on a function of a subset of the state variables, rather than the full set, as the selected methods. Therefore, by observing this difference, it is not clear whether fair comparisons can be made.

# Chapter 5

# Performance evaluation framework

## 5.1 Introduction

Usually feedback scheduling (FBS) and event-driven control (EDC) advances are evaluated and compared with the performance achieved by the traditional static approach considering similar circumstances. Therefore the benefits of each novel approach are measured taking the static approach as a reference. However the performance evaluation does not consider any other similar resource/performance-aware policy. Hence, it becomes very difficult to analyze and evaluate the real benefits of each new approach compared to the state-of-the-art results. Moreover, rarely implementation issues are reported.

In order to analyze the implementation feasibility of these methods and evaluate state-of-the-art under fair conditions, it is required to define and implement a common framework capable to offer basic services which allow the implementation and the performance evaluation of a wide-variety of methods (FBS and EDC included). This framework must allow evaluating whether theoretical approaches can be implemented in practice, and how different strategies impact on control performance, resource utilization and computational overhead. This will provide an insight on the benefits and drawbacks of each algorithm. Apart from permitting evaluation of feedback scheduling methods and event-driven methods, it must also allow the implementation of different control task models.

The design of the performance evaluation framework is the result of the analysis performed in the taxonomy (Chapter 4). Among other specifications, the framework must support different triggering paradigms (event-driven, time-trigger), different optimization algorithms (on-line, off-line), different evaluation parameters (control performance, resource utilization), and different sampling periodicity (static periods, varying periods, aperiodic intervals, sequences).

The framework is composed by a simulation platform and by an experimental platform. Each platform has been designed considering different functional modules. To validate the correct operation of the framework, the group of feedback scheduling and event-driven control methods selected in Section 4.4 have been implemented.

## 5.2 Problem set-up

This section presents how state-of-the-art FBS and EDC approaches were evaluated in terms of control performance and/or resource efficiency. The analysis of these methods focuses also in the characteristics of the different evaluation platforms used for each case.

Table 5.1: FBS evaluation parameters and platform

| Method | Plant | Evaluation Parameters: Control Performance | Evaluation Parameters: Processor Load | Platform: Simulation | Platform: Experimental |
|---|---|---|---|---|---|
| Set96 [SLSS96] | bubble control | Quadratic | % | Yes | No |
| Set98 [SLS98] | temperature control, bubble control | Quadratic | No | No | No |
| Eke00 [EHÅ00] | inverted pendulum | Quadratic | No | Yes | No |
| Reh00 [RS00] | inverted pendulum | Quadratic | No | Yes | No |
| Hen02 [HCAÅ02] | quadruple tank process | Error | No | Yes | No |
| Cer02 [CEBÅ02] | inverted pendulum | Quadratic | No | Yes | No |
| Pal02 [PPSV+02] | scalar system | Robustness | No | Yes | No |
| Cha03 [CLS03] | temperature control, bubble control, inverted pendulum | Quadratic | No | Yes | Yes |
| Mar04 [MLB+04] | inverted pendulum | Error | % | Yes | Yes |
| Hen05 [HC05] | integrator | Quadratic | No | Yes | No |
| Pal05 [PPBSV05] | inverted pendulum | Robustness | No | No | Yes |
| Ben06 [GCHI06] (NET) | unknown | Error | No | Yes | No |
| Cas06 [CMV+06] | ball and beam, dc motor, harmonic oscillator | Quadratic | No | Yes | No |
| Bini08 [BC08] | scalar plants | Quadratic | No | Yes | No |
| Ben09 [GcH09] | inverted pendulum, dc motor | Error | Rate | Yes | Yes |
| Mar09a [MLB+09] | inverted pendulum, ball and beam | Quadratic | % | Yes | No |
| Sam09a [SEPC09] | inverted pendulum, ball and beam, dc servos, harmonic oscillator | Quadratic | Runtime | No | Yes |
| Cer10 [CVMC10] | double integrator | Quadratic | Overhead | No | Yes |

Table 5.1 shows a summary of the evaluation parameters and the platform characteristics for existing FBS methods obtained from Table 4.1. It also includes the targeted plants used during the method evaluation. For the evaluation parameters the table indicates how performance is measured and if processor load is considered. For the platform characteristics, it identifies if simulation and/or real experiments were conducted.

Similarly, Table 5.2 shows a summary of the evaluation parameters and the platform characteristics for the reviewed EDC methods obtained from Table 4.1. It also includes the targeted plants used during the method evaluation.

By analyzing both tables, it can be noticed that in most of the cases the proposed algorithms were only simulated using a computational tool, and only in few cases real experiments were developed in order to validate the proposed approach.

Another important aspect is the selection of the evaluation metrics. In practically all the FBS approaches the main parameter to be measured is control performance and in most of the cases it is measured by using a quadratic cost function. Processor load for FBS is calculated only in few cases, where it is measured in terms of percentage of use, and in general the computational overhead caused by the implementation of the algorithm is not taken into consideration. For the EDC approaches the main parameter to be measured is processor load, in most of the cases it

Table 5.2: EDC evaluation parameters and platform

| Method | Plant | Evaluation Parameters | | Platform | |
|---|---|---|---|---|---|
| | | Control Performance | Processor Load | Simulation | Experimental |
| Arz99 [Årz99] | double tank process | No | % | Yes | No |
| Hee99 [HGvZ+99] | electrical motor | Error | No | No | Yes |
| Zha99 [ZZ99] | second-order plant | Transient response | No | Yes | No |
| Ast02 [AB02] | integrator | Variance | No | Yes | No |
| Vel03 [VMF03] | ball and beam | Transient response | % | Yes | No |
| Tab07 [Tab07] | second-order plan | Error | No | Yes | No |
| Lem07 [LCH+07] | inverted pendulum | Transient response | No | Yes | No |
| Joh07 [JHC07] | first-order plant | Quadratic | Jobs | Yes | No |
| Hen08 [HJC08] | first-order plant | Quadratic | Jobs | Yes | No |
| Ant08 [AT08a] | non-linear plant | Error | Jobs | Yes | No |
| Ant08a [AT08b] | jet engine compressor | Error | Jobs | Yes | No |
| Hee08 [HSB08] | | | | | |
| Wan08 [WL08b] | inverted pendulum | Error | Jobs | Yes | No |
| Wan08a [WL08a] | inverted pendulum | Error | Jobs | Yes | No |
| Ant09 [AT09] | non-linear plant | No | Runtime | Yes | No |
| Mar09 [MVB09] | double integrator | Transient response | Jobs | Yes | No |
| Maz09 [MAT09] | batch reactor model | Stability | Jobs | Yes | No |
| Maz09a [MT09] | batch reactor model | Stability | Jobs | Yes | No |
| Vel09 [VMB09a] | double integrator | Quadratic | % | No | Yes |
| Wan09 [WL09a] | inverted pendulum | Error | Jobs | Yes | No |
| Wan09a [WL09b] | inverted pendulum | No | Jobs | Yes | No |
| Ant10 [AT10] | jet engine compressor | Error | Jobs | Yes | No |

is measured by counting the number of jobs executed during a specific period, and just in few cases the load is measured in terms of percentage of processor utilization or total runtime. The control performance for EDC approaches is just analyzed by observing the transient response and sometimes by measuring the deviation from the desired set-point, i.e. transient error.

Given the great diversity of approaches, the framework must be generic enough to accommodate the different policies, but it also must be flexible and accurate enough to facilitate the implementation of these policies while permitting to assess their exact operation in fair/comparable scenarios.

Regarding the simulation tools and experimental platform for the evaluation framework, the following requirements have been placed. The simulation platform must allow co-simulation of 1) real-time control tasks executing on top of a real-time kernel and 2) plant's dynamics. It must permit conducting extensive evaluation of different policies, considering a wide variety of scenarios. The experimental platform has as a main goal to permit proving that each method can be implemented in a real physical system.

The simulation tool chosen as a basis of the simulation part of the evaluation framework is the TrueTime toolbox [LU10] integrated with Matlan/Simulink [Mat10]. TrueTime has been shown to be a well accepted simulation tool among the real-time and control community as demonstrated by the large number of publications presenting the simulator and its modifications [1], as well as for the

[1] See, `http://www.control.lth.se/truetime/`

large number of publications where TrueTime has been the tool for validating diverse theoretical results on control and real-time systems co-design. Matlab/Simulink with the TrueTime toolbox offers a computer block that simulates a computer with a flexible real-time kernel executing user-defined threads and interrupt handlers. Threads may be periodic or aperiodic and are used to simulate controller tasks, communication tasks etc. Interrupt handlers are used to serve internal and external interrupts. The kernel maintains a number of data structures commonly found in real-time kernels, including a ready queue, a time queue, and records for threads, interrupt handlers, events, monitors etc. It interfaces with other Simulink blocks. The input signals are assumed to be discrete, except the signals connected to the A/D port which may be continuous. All output signals are discrete. The Schedule and Monitors ports provide plots of the allocation of common resources (processor) during the simulation.

For the experimental part of the evaluation framework, the Erika real-time kernel [Srl08a] running on top of a Full Flex board [Srl08b] equipped with a Microchip dsPIC33 microcontroller [Mic05] has been chosen. Although being a relatively new kernel (released in its first form in 2003), it has an active development and support, and it has been shown to be a good platform for testing state-of-the-art research and educational results on embedded control systems [MVF+10]. The Erika Enterprise kernel has been developed with the idea of providing the minimal set of primitives which can be used to implement a multitasking environment. Erika kernel is a real-time operating system for small microcontrollers based on an API similar to those proposed by the OSEK/VDX Consortium [OSE]. The Erika kernel implements scheduling algorithms such as Fixed Priority (FP) with preemption thresholds, and Earliest Deadline First (EDF) which can be used to schedule periodic tasks with real-time requirements (time-triggered schedule). In addition, it can handle the interrupts that are raised by the I/O interfaces, internal events and timers which allows linking a handler written by the user into an interrupt vector in order to schedule aperiodic tasks (event-driven scheduling). Erika kernel consist of two layers: the Kernel Layer and the Hardware Abstraction Layer (HAL). The Kernel Layer contains a set of modules that implement task management and real-time scheduling policies. The Hardware Abstraction Layer contains the hardware dependent code that handles context switches and interrupt handling. The Microchip dsPIC33 microcontroller family is supported by the Erika kernel HAL. The Microchip dsPIC33 microcontroller is a high-performance 16-bit digital signal controller (DSC) designed for embedded systems solutions. Specifically an embedded board for dsPIC33 named Full Flex has been used as the hardware experimental platform, see Figure 5.1. The Full Flex board mounts a Microchip dsPIC33 microcontroller, and exports almost all the pins of the microcontroller. The Full Flex board, integrates an extra-robust power supply circuitry, which allows the usage of a wide range of power suppliers. It accepts voltage ranges between 9-36V. The power supply signal is filtered and adapted to the internal levels.

Finally, it is important to stress that the surveyed results in Tables 5.1 and 5.2 also showed the diversity of controlled plants that have been used. For the selection of the plant, a few factors were considered. First, many standard basic and advanced controller design methods rely on the accuracy of the plant mathematical model. The more accurate the model, the more realistic the simulations, and the better the observation of the effects of the controller on the plant. Hence, the plant was selected among those for which an accurate mathematical model could easily be derived. Plants such as an inverted pendulum or a direct current motor are the *defacto* plants for benchmark problems in control engineering. However, their modelling is not trivial and the resulting model is often not accurate. Second, it was desired to have a plant that could directly be plugged into a microcontroller without using intermediate electronic components. That is, the transistor-transistor logic (TTL) level signals provided by the microcontroller should be enough to carry out the control. Note that this is not the case, for example, for many mechanical systems. Such a simplification in terms of hardware reduces the modelling effort to study the plant and no

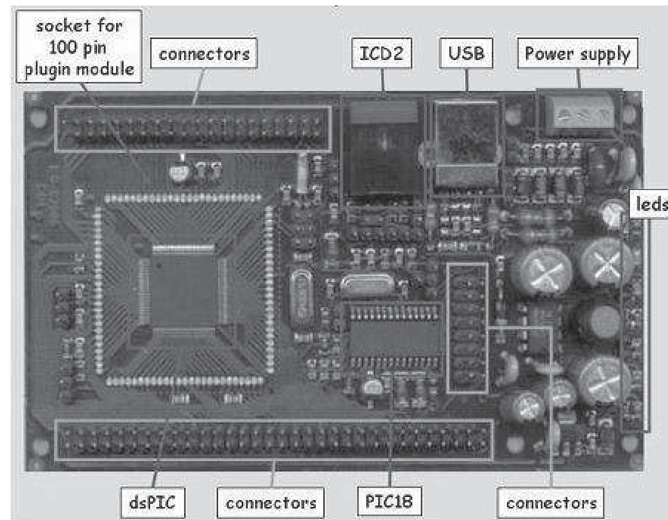



Figure 5.1: Full Flex board with a dsPIC33 microcontroller.

models for actuators or sensors are required. Third, it was also desired to have a plant that can be easily reproduced, that is, it is easy and cheap to built. Regarding all this requirements, the election was an electronic circuit in the form of a double integrator. Further details will be given in Subsection 5.3.4.

## 5.3 Evaluation framework

This section describes the characteristics of the proposed performance evaluation framework. First, the general services provided by the framework are presented and then the evaluation parameters used in the framework for performance comparison purposes are introduced. Afterward, the design and implementation of the performance evaluation framework is presented.

### 5.3.1 Framework services

This section describes the common characteristics that both, the simulation part and the experimental part have to fulfill. To this extend, several services have been identified. Framework services refer to the functional modules that constitute the performance evaluation framework. Each functional module performs specific activities that constitute each service. They have been conceptually defined for providing a flexible and scalable evaluation platform. Framework services are baseline configuration, resource manager, task controller, optimization method and performance measurement (see Figure 5.2).

#### 5.3.1.1 Baseline configuration

The baseline configuration module is responsible for providing an interface between the plants and the other functional models. The proposed framework considers the case of a single processor with multitasking capabilities controlling several continuous plants, as showed in Figure 5.3 for the case of three plants. This configuration is defined in this module. However, other cases can be supported if a new configuration is defined within this module, e.g. multiple-processors controlling one plant

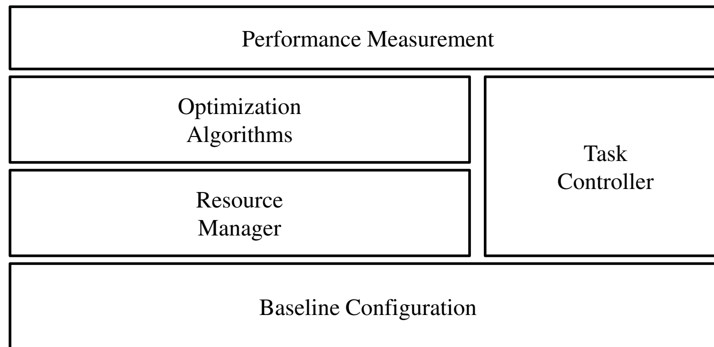



Figure 5.2: Framework functional modules.

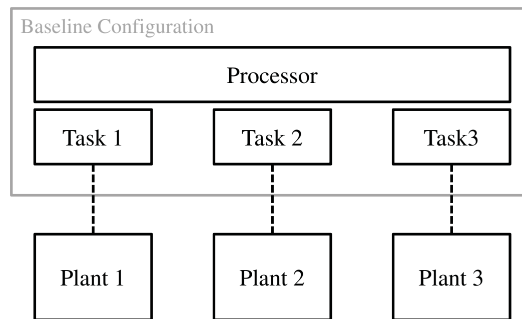



Figure 5.3: Framework multitasking single processor configuration.

each, or multiple-processors operating as network nodes. The module defines the connectivity between the processor and the plants, this includes analog lines, analog to digital converters, and digital to analog converters.

#### 5.3.1.2 Resource manager and optimization method

The resource manager module obtains the system dynamics information in the form of kernel workload, plant error (instantaneous or finite horizon), plant measurement state or any system feedback information used to re-allocate resources. Using this information, the resource manager executes the previously selected optimization algorithm (see Figure 5.4). The optimization method module is conformed by a repository of algorithm routines. In particular, the algorithms shown in Table 4.2 have been implemented. Each algorithm represent a specific FBS or EDC approach. Off-line optimization algorithms are executed just once during the system initialization process, meanwhile on-line optimization algorithms are executed periodically (mainly for FBS approaches) or aperiodically (for EDC approaches). The optimization module receives the system dynamics information from the resource manager, then the optimization procedure results are sent to the task controller to indicate new settings such as new values for task periods or new controller gains.

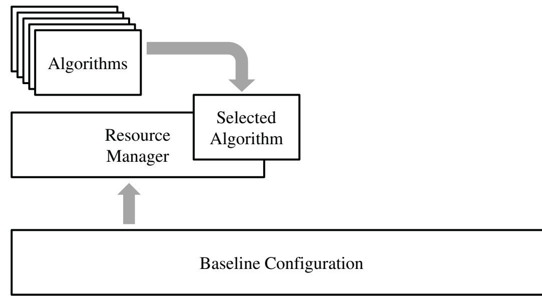



Figure 5.4: Framework resource manager module.

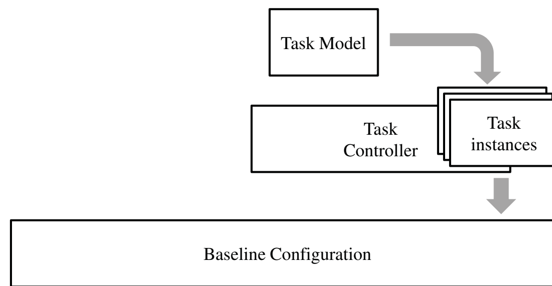



Figure 5.5: Framework task controller module.

#### 5.3.1.3 Task controller

The task controller module is the responsible to create task instances according to a specific task model (see Figure 5.5). A set of control task models definitions are available in this module, such as naif, one-sample, split, switching and one-shot task models. Task instances support both periodic and aperiodic interruptions. Timing parameters for the task instances can also be modified on-line by the optimization method module. This module indicates to the baseline configuration module when the different activities within each closed loop operation must be executed, that is, when sampling, control and actuation actions must take place.

#### 5.3.1.4 Performance measurement

The performance measurement module is able to obtain information from any of the other modules. Raw information such as algorithm execution time, plants' errors and plants' states are processed by this module in order to perform a complete assessment using proper metrics to evaluate control performance, resource utilization and computational overhead. This module allows performance evaluation of any resource/performance-aware policy under similar circumstances, providing a fair comparison among them. It can be tuned to process different system (computer and/or plant) data, and to evaluate using alternative metrics.

For simulations, control performance is measured using a continuous standard quadratic cost

function

$$J_{control} = \int_{0}^{t_{eval}} \left[ x^T(t)Qx(t) + u^T(t)Ru(t) \right] dt. \tag{5.1}$$

where the $Q$ and $R$ represents the cost weighting matrices and $t_{eval}$ the simulation period. For experiments, control performance is obtained from a discrete-time quadratic cost function

$$J^d_{control} = \sum_{k=0}^{t_{eval}} \left[ x^T(k)Q_d x(k) + 2x^T(k)N_d u(k) + u^T(k)R_d u(k) \right], \tag{5.2}$$

where the $Q_d$,$N_d$ and $R_d$ represent the discrete cost weighting matrices. Appendix A, details on how to obtain (5.2) from (5.1). Since the experimental platform uses the microcontroller to measure periodically the plant states, then a discrete cost function is required. Meanwhile the simulation platform is capable of obtaining continuous measurements directly from the plant.

Resource utilization is measured as a percentage of use of the processor during each evaluation period ($t_{eval}$). So the resource utilization is defined as

$$J_{resource} = \left( \frac{1}{t_{eval}} \sum_{i=1}^{n} E_i \right) * 100, \tag{5.3}$$

where $n$ is the number of tasks sharing the same processor, and $E$ corresponds to either the total processor time assigned to a specific control task during the simulation period or the total processor time measured for a specific control task during the experimentation period. Therefore, for each specific control task, the total processor time (assigned or measured) is

$$E = \sum_{j=1}^{m} C_j \tag{5.4}$$

where $m$ is the number of times that a specific control task is invoked during the simulation/execution period, and $C$ corresponds to the task execution time. Computational overhead is included in this parameter.

A performance index is defined in order to incorporate control performance and the resource utilization in one metric. Therefore performance index for simulation is defined by

$$PI = J_{control} * J_{resource}, \tag{5.5}$$

and performance index for experiments is defined by

$$PI^d = J^d_{control} * J_{resource}. \tag{5.6}$$

### 5.3.2 Framework design and implementation: simulation part

The simulation platform implementing the required services can be described from two main views. The structural view defines the static elements that integrate the platform, meanwhile the execution view defines the sequence of events or steps conducted during the simulation process. The software for the simulation platform can be found at `http://dcs.upc.es/`.

From a structural view the simulation platform has two levels. Matlab programs running in the Matlab is the first level. Simulink blocks and the TrueTime elements running in the Simulink environment represent the second level.

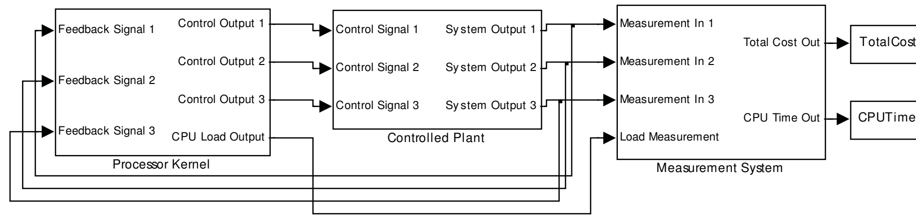



Figure 5.6: Simulink/TrueTime model

Matlab programs are physically and logically grouped in five main folders. The root folder contains the main program which represents the simulation starting point and it contains configuration files for timing and disturbance data, and plant model definition. The resource manager folder and task controller folder contains programs which are triggered by Simulink/TrueTime blocks. The optimization algorithms and the task models folders represents a repository of programs that implement different EDC and FBS strategies. Partial Matlab source code for main modules can be found at Appendix B.

The Simulink/TrueTime model is composed by three elements: processor kernel, controlled plant and measurement system (see Figure 5.6 for the model in the case of three control tasks). The processor kernel sends control signals to the plants, plants send feedback information to the processor, and the measurement system is capable of obtaining metrics from the other two main blocks.

The processor kernel simulates a multitasking processor with a real-time operating system (see Figure 5.7 for details of this block in the scenario of three control loops). Internally this block is composed by a task controller and a resource manager. The task controller is in charge of the creation of task instances based on a specific control task model. Resource manager obtains feedback information and executes a specific resource optimization algorithm and modifies tasks's controllers timing parameters if required by the algorithm. The controlled plants group contains a Simulink space-state block for each plant being under control. Each plant can be affected by disturbance data which is feed during run-time. For the current implementation, it is assumed that plant states are available and therefore there is no need for observers. However if required, observers and estimators can be incorporated in this block. The measurement system block contains two elements, one to measure control performance and the other to measure processor load. Basically each element integrates the individual measurements using the defined cost functions and stores the results in a data structure. If there is a need for additional metrics, this module can be enhanced to support them.

From an execution point of view, the simulation platform can be described with the flow diagram showed in Figure 5.8. Each element in the diagram has the following description:

- System configuration. During this step the main configuration parameters are selected, including the control task model (naif, one-shot, one-sample, etc.), the resource optimization algorithm (static, optimal, event-driven) and the plant model. This configuration section contains the definition of the system model, the configuration is modified each time a new model is simulated.

- Valid configuration. Once the parameters are selected a validation process is executed in order to verify that the selected parameters are correct, e.g. whether the selected task model supports the selected resource optimization algorithm. If the configuration is not valid the

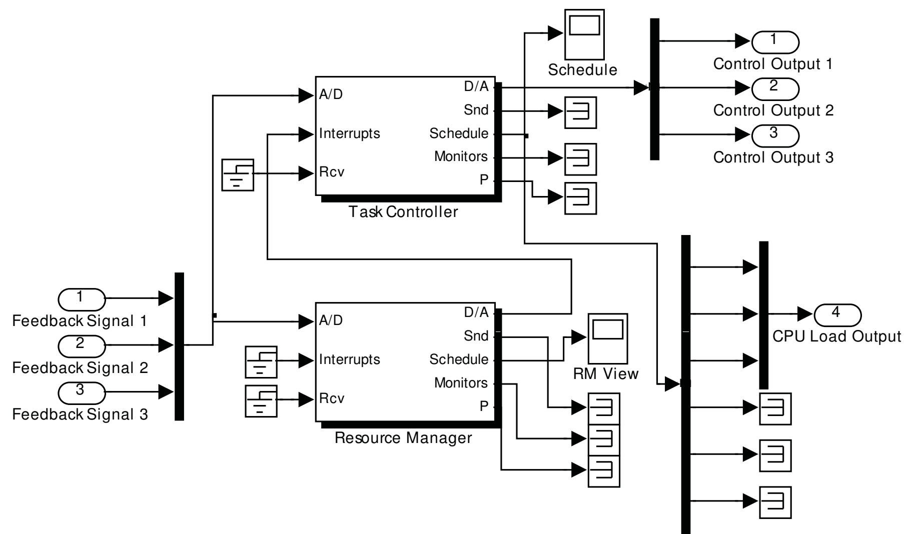



Figure 5.7: Processor kernel model

system execution ends.

- Load data. Before the simulation execution, the timing and disturbance data must be loaded into the system. This data indicates how many execution runs will be conducted, how long each simulation will last, when and how many set-point changes or disturbances will appear during the execution time, or what the magnitude of each disturbance or new set point is. All this information should be previously generated, usually in a random procedure, and stored in a specific file.

- Execute simulation. Once configuration and data is ready, the simulation is executed. The simulation consists in the upload and execution of the Simulink/TrueTime model. This model is generic for all simulations. What it makes the difference is the parameters received from the system configuration and load data blocks.

- Generate partial results. When a simulation run is completed partial results are stored in a data structure. This partial results include information such as tasks execution time and control performance.

- More data. The system verifies if there is more data available for a next simulation. If all data loaded has been already used, then the system proceeds to summarize the results (using the plot block). If there is data available, another simulation run is executed. In this way, different simulation scenarios can be executed for the same system model, and also the same scenarios can be used later on for other models.

- Summarize results and plot. Finally, the accumulated data is processed and summarized in order to provide results and plots which represents the performance of the specific system model.

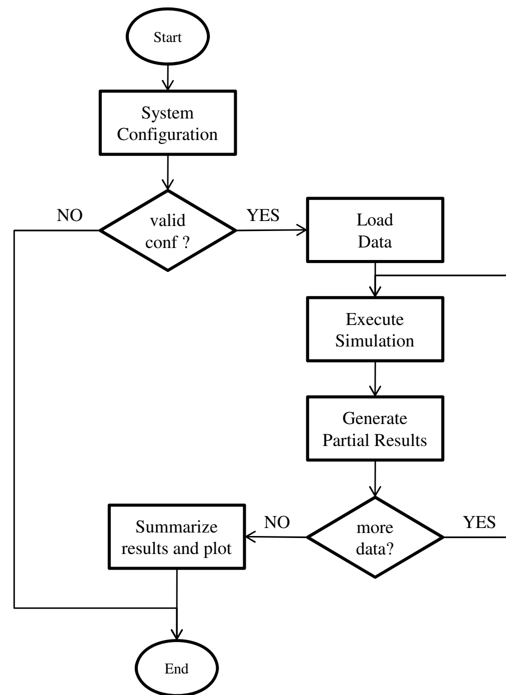



Figure 5.8: Simulation framework flow diagram

### 5.3.3 Framework design and implementation: experimental part

The experimental platform environment can be divided into two main configurations: run-time and programming, as it allows easy and scalable implementation of different resource aware policies. The software for the experimental platform can be found in `http://dcs.upc.es`. The run-time configuration defines the elements used during the microcontroller execution time. This configuration provides an adequate connectivity among the hardware elements in order to execute the control tasks and extract the evaluation data from the plants through the microcontroller. The programming configuration refers to the microcontroller code generation process, it provides an adequate environment to program, compile and download the software that will run in the dsPIC33 microcontroller for the different policies.

#### 5.3.3.1 Run-time configuration

In the run-time configuration the Matlab IDE (Integrated Development Environment) is used as an interactive environment that enables the user to perform computationally intensive tasks. Matlab provides key parameters and receives on-line information from the microcontroller. A Matlab program (m-file) is executed in order to trigger the execution of the microcontroller program. During the microcontroller execution, Matlab is receiving raw data about control performance and processor load. This data is then processed in order to generate plots and obtain summarized information regarding the behavior of the embedded control system which is conformed by the microcontroller and the controlled plants. In the dsPIC33 microcontroller several task instances area created, each one controlling one plant. The case of one task controlling an electronic double integrator systems

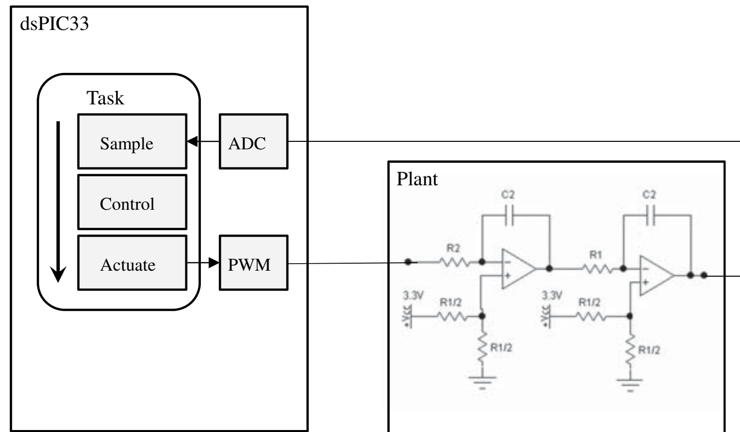



Figure 5.9: A microcontroller task controlling one double integrator plant.

is illustrated in Figure 5.9. Each task is configured in order to be scheduled by the kernel. When a task job is executed, three activities are performed. First, the controlled plant is sampled by reading a value from the microcontroller ADC (Analog-to-Digital Converter) which is connected to the plant output, then it calculates the control signal value according to a specific algorithm, and finally it sends the control action through the microcontroller PWM (Pulse-Width-Modulator) that is connected to the plant input. The hardware for the controlled plants is implemented in the daughter board. According with the performance evaluation framework design, the performance measurement module is constituted by the Matlab IDE, the m-file, and the plots and results. Meanwhile the disPIC33 microcontroller constitutes the implementation of the baseline configuration, resource manager, task controller and optimization algorithms modules.

#### 5.3.3.2 Programming configuration

The programming configuration is composed by the following elements: Erika kernel, source code, Eclipse/RT Druid, MPLAB IDE (Integrated Development Environment) and ICD2 (In-Circuit Debugger), and the Full Flex board. Erika and the Full Flex board equipped with the dsPIC33 Microcontroller were described in Section 5.2. The source code is the main part of the programming configuration and it is composed by three files embedded in a project: 1)`conf.oil` that contains the system configuration and the tasks, counters and alarm definition using OIL (OSEK Implementation Language); 2)`code.c` that represents the source code main program, contains the controller specific activities using C language; 3)`setup.c` contains a group of common-used functions for the Full Flex board. Main source code functions and definitions can be found at Appendix C. Additional sources files may be added to enhance system functionalities. Besides, MPLAB IDE [Mic10], that is an integrated tool-set for the development of embedded applications employing Microchip's microcontrollers, is used to import the object file produced by the compilation of the source code in the Eclipse/RT-Druid [Fou10] environment, and to download the program to the dsPIC33 microcontroller trough the MPLAB In-Circuit Debbuger (ICD2) interface [Mic09].

**Algorithm 5**: `int main(void)`

```
begin
    Clock_setup()
    Timer1_program()
    Full_Flex_setup()
    SetRelAlarm(AlarmReferenceChange, 1000, 1000)
    SetRelAlarm(AlarmPeriodicController, 1000, 50)
    SetRelAlarm(AlarmOnlineOptimization, 1000, 10)
    SetRelAlarm(AlarmEventController, 1000, 0)
    SetRelAlarm(AlarmSend, 1000, 5)
    while(1)
    begin
        background activities (if any) should go here
    end
end
```

Figure 5.10: Main program pseudo-code

#### 5.3.3.3 Code details

The main services implemented in the experimental platform inside of the `code.c` source file are described next.

The generation of time in Erika is based on the *Timer1* register, which is programmed to rise an interruption every *tick* (1ms). The interrupt handler is the *CounterTick* function that shoots the diverse programmed alarms. Each alarm has an associated task, which is then activated for execution. The task can be programmed to be executed just once (aperiodic task) or repeatedly (periodic task). The overall code has been divided into 5 tasks:

**TaskReferenceChange:** periodic task that generates the reference signal by modifying the plant set-point.

**TaskPeriodicController:** periodic task that implements a time-triggered controller. This task is used when implementing any FBS approach.

**TaskOnlineOptimization:** periodic task that executes on-line optimization activities used for some FBS approaches.

**TaskEventController:** aperiodic task that implements a event-driven controller. This task is used when implementing any EDC approach.

**TaskSend:** periodic task that sends plant state information from the dsPIC33 to the PC using RS232 serial communication.

The main code initializes software and hardware components (such as clock, timer, ADC), configures alarms and activate tasks, as shown in the pseudo-code of Figure 5.10. The *SetRelAlarm* primitive is used to fire an alarm which activates a task with an specific offset and periodicity value, if periodicity is zero the task is executed just once. For example the alarm *AlarmEventController* activates the task *TaskEventController* after an offset of 1000ms and it is executed once, meanwhile the alarm *AlarmPeriodicController* activates the task *TaskPeriodicController* that is periodically executed every 50ms after an offset of of 1000ms.

The *TaskPeriodicController* implements the controller code for a periodic task (Figure 5.11). First, it reads the reference signal value (set-point), then it reads the plant state variables through

**Algorithm 6**: TASK(TaskPeriodicController)

```
begin
    Read_reference()
    Read_state()
    Obtain_tracking_error()
    u=Calculate_control_signal()
    Apply_control_signal(u)
end
```

Figure 5.11: Periodic controller pseudo-code

**Algorithm 7**: TASK(TaskOnlineOptimzation)

```
begin
    status=Obtain_plant_or_processor_status()
    new_period=Optimzation_process(status)
    if ( new_period <> current_period ) begin
        Update_controller_gain()
        CancelAlarm(AlarmPeriodicController)
        SetRelAlarm(AlarmPeriodicController, new_period, new_period);
        ActivateTask(TaskPeriodicController);
    end
    old_period=current_period
end
```

Figure 5.12: On-line optimization pseudo-code

the dsPIC33 ADC. The tracking error is calculated using the state variables and the set-point. Later, it calculates the control signal $u$ using the controller gain $L$, and finally it sets the PWM duty cycle according with the control signal.

The *TaskOnlineOptimization* executes on-line optimization activities according to the specifications given by a particular FBS approach. If required, this task is capable to modify the task periodicity by cancelling the current alarm and releasing the alarm with a new task activation period. The pseudo-code in Figure 5.12, shows a generic optimization tasks which first obtains the plant state or the processor status in order to conduct the optimization process; then if a new task period is obtained, the controller gain $L$ is updated and a new task period is set.

The *TaskEventController* implements the controller code for an aperiodic task (Figure 5.13). Notice that according to the code shown in Figure 5.10, the alarm that activates the task has been configured to be executed once. The code is similar to the *TaskPeriodicController*, however, at the end, it computes its next activation time via *Calculate_next_activation_time* function, and it uses this value to set the associated alarm. This code correspond to a self-triggered approach, but it can also support event-triggered approaches by attaching this task to an external interrupt handler.

### 5.3.4 Plant details

For both simulation and experimental parts, an electronic circuit in the form of a double integrator has been used as a controlled plant. Figure 5.14 shows the physical implementation of the double integrator. 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 dsPIC33, and thus no negative voltages are available, the 0V voltage ($V_{ss}$) in the

**Algorithm 8**: `TASK(TaskEventController)`

```
begin
    Read_reference()
    Read_state()
    Obtain_tracking_error()
    u=Calculate_control_signal()
    Apply_control_signal(u)
    Event_time = Calculate_next_activation_time()
    SetRelAlarm(AlarmEventController, Event_time, 0)
end
```

Figure 5.13: Event controller pseudo-code

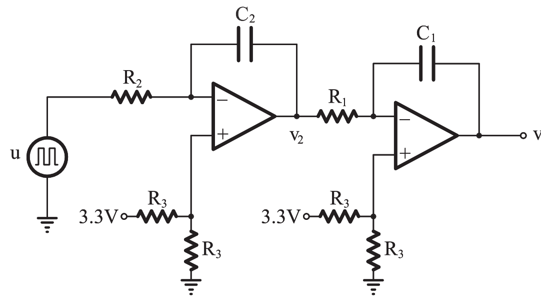



Figure 5.14: Electronic double integrator circuit

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 values are shown in Table 5.3.

The operational amplifier in integration configuration can be modelled by

$$V_{\text{out}} = \int_0^t -\frac{V_{\text{in}}}{RC}\,dt + V_{\text{initial}} \tag{5.7}$$

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

Taking into account (5.7), and the scheme shown in Figure 5.14, the double integrator plant

| Component | Nominal value |
|---|---|
| $R_{1/2}$ | $1k\Omega$ |
| $R_1$ | $100k\Omega$ |
| $R_2$ | $100k\Omega$ |
| $C_1$ | $470nF$ |
| $C_2$ | $470nF$ |

Table 5.3: Electronic components nominal values

dynamics can be modelled by

$$\begin{aligned}\frac{dv_2}{dt} &= \frac{-1}{R_2C_2}u\\ \frac{dv_1}{dt} &= \frac{-1}{R_1C_1}v_2\end{aligned}$$

In state space form, the model is

$$\begin{aligned}\begin{bmatrix}\dot{v}_1\\ \dot{v}_2\end{bmatrix} &= \begin{bmatrix}0 & \frac{-1}{R_1C_1}\\ 0 & 0\end{bmatrix}\begin{bmatrix}v_1\\ v_2\end{bmatrix} + \begin{bmatrix}0\\ \frac{-1}{R_2C_2}\end{bmatrix}u\\ y &= \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}v_1\\ v_2\end{bmatrix}\end{aligned}$$

The model validation has been performed by applying a standard control algorithm with a sampling period of $h = 50$ms, with reference changes, and comparing the theoretical results obtained from a Simulink model with those obtained from the plant. With the validated values for the components, the model used for controller design is given by

$$\begin{aligned}\dot{x} &= \begin{bmatrix}0 & -21.2766\\ 0 & 0\end{bmatrix}x + \begin{bmatrix}0\\ -21.2766\end{bmatrix}u\\ y &= \begin{bmatrix}1 & 0\end{bmatrix}x\end{aligned} \tag{5.8}$$

where the state vector is $x = [v_1 \quad v_2]^T$.

Figure 5.15 shows the results of this validation. In particular, the controller gain $L$ is obtained using linear quadratic (LQ) optimal design which minimizes a discrete cost function equivalent to the continuous cost function

$$J = \int_0^\infty \left(x^T(t)Qx(t) + u^T(t)Ru(t)\right)dt \tag{5.9}$$

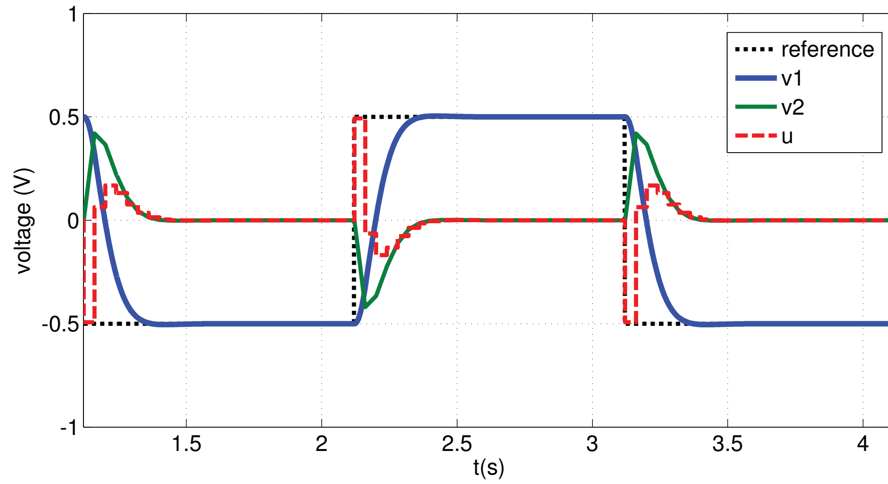
with $Q$ being the identity and $R = 1$, for the different sampling period choices. Since the voltage input of the operational amplifier is 1.6V (which is half $V_{cc}$: the measured $V_{cc}$ is 3.2V although it is powered by 3.3V), the tracked reference signal has been established to be from 1.1V to 2.1V (±0.5V around 1.6V). For the tracking, the feed-forward matrix $N_u$ is zero and $N_x = \begin{bmatrix}1 & 0\end{bmatrix}$ [ÅW97].

The goal of the controller is to make the circuit output voltage ($v_1$ in Figure 5.14) to track a reference signal by giving the appropriate voltage levels (control signals) $u$. Both states $v_1$ and $v_2$ can be read via the ADC port of the microcontroller and $u$ is applied to the plant through the PWM output.

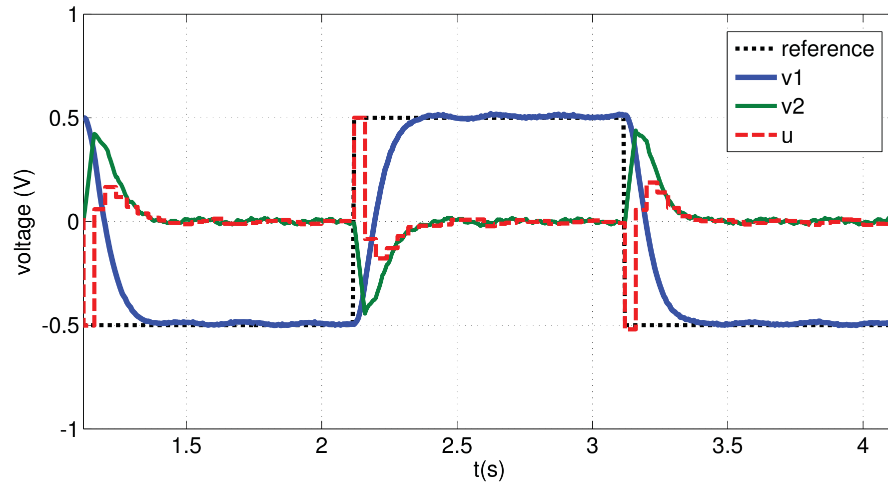
As a prototype and performance demonstrator, three plants in the form of double integrator electronic circuits that are controlled by three control tasks concurrently executing in Erika and scheduled under the EDF scheduling algorithm have been implemented in the daughter board, as shown in Figure 5.16. The three plants can be easily interfaced to the Full Flex base board.

## 5.4 Implementation and evaluation of selected methods

This section describes the implementation and evaluation of selected FBS and EDC methods using the performance evaluation framework. The methods have been selected due their representative characteristics within these two tendencies, as discussed in Chapter 4. The objective is to provide valuable elements of analysis, through the results obtained from the evaluation, in order to discuss




(a) Theoretical simulated plant response




(b) Experimental plant response

Figure 5.15: Model validation.

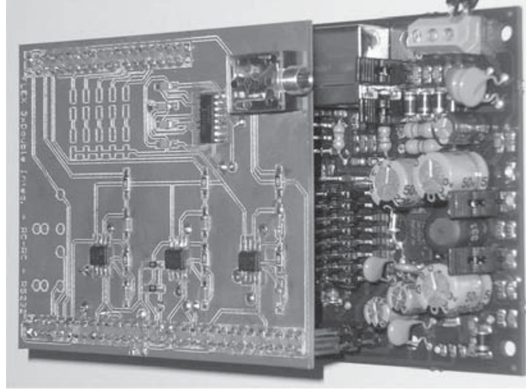


Figure 5.16: Experimental setup

| Approach | Type | Task 1 | Task 2 | Task 3 |
|---|---|---|---|---|
| Static [ÅW97] | | 0.0300 | 0.0900 | 0.0900 |
| Off-line FBS [SLSS96] | FBS | 0.0500 | 0.0500 | 0.0500 |
| On-line FBS-Inst. [MLB+04] | FBS | 0.0300 | 0.0900 | 0.0900 |
| On-line FBS-FH [HC05] | FBS | 0.0300 | 0.0900 | 0.0900 |
| Heuristic self-triggered [VMF03] | EDC | 0.0812 | 0.0812 | 0.0812 |
| Self-triggered [LCH+07] | EDC | 0.0555 | 0.0555 | 0.0555 |
| Optimal Self-triggered [MVB09] | EDC | 0.0576 | 0.0576 | 0.0576 |

Table 5.4: Simulation tasks sampling periods (seconds)

the key elements of each method. Both control experiments and simulations are reported.

### 5.4.1 Simulation

The simulation environment correspond to the one specified in the performance evaluation framework, considering that the model is integrated by three control loops: three plants are being controlled by three control tasks, which are executed on the real-time kernel processor. Control tasks can be executed either on a time-triggered basis or on an event-triggered basis, depending on the resource/performance-aware policy under evaluation. The total simulation time is 25s and each control loop has a set-point change every 3s.

#### 5.4.1.1 Tasks settings for each method

In order to have a fair control performance evaluation, the task periods for FBS methods and the average task periods for the EDC methods, were selected in order to provide similar CPU load (5% approximately), as shown in Table 5.4. For simulation purposes, the execution time of any task is always 10ms, regardless the computation complexity in the task, this consideration was taken for the sake of simplification, however in the control experiments real execution times are measured.

A detailed description of the settings applied to each method, including algorithm details or gain details, is presented next:

- Static approach: periods for each task are heuristically selected and the corresponding discrete-time controllers designed before run-time using LQ optimal controller design using the cost

| Approach | Control Performance $J_{control}$ | Resource Utilization $J_{resource}$ | Performance Index $PI$ |
|---|---|---|---|
| Static [ÅW97] | 1.9905 | 4.3720 | 8.7026 |
| Off-line FBS [SLSS96] | 1.4644 | 4.6720 | 6.8415 |
| On-line FBS-Inst. [MLB+04] | 1.3286 | 4.5320 | 6.0211 |
| On-line FBS-FH [HC05] | 1.3288 | 4.5680 | 6.0700 |
| Heuristic self-triggered [VMF03] | 1.3636 | 2.9560 | 4.0309 |
| Self-triggered [LCH+07] | 1.3469 | 4.3240 | 5.8238 |
| Optimal Self-triggered [MVB09] | 1.3498 | 4.0840 | 5.5127 |

Table 5.5: Control performance and resource utilization simulation results

function specified in the framework.

- Off-line FBS [SLSS96]: since the three plants are equal, the off-line optimization procedure mandates to execute each task with the same period. The *a priori* relation between a control performance index expressed in terms of cost and a range of sampling frequencies is defined. This relation is approximated by a decreasing exponential function. After guaranteeing a maximum feasible period to each control task, an off-line optimization procedure re-scales periods until the task set is feasible under EDF while minimizing the cost, considering a desired resource utilization level $U_{ref} = 0.05$. Once periods are set, control tasks are scheduled under EDF.

- On-line FBS - Inst. [MLB+04]: the final outcome of the method mandates to consider at run-time only two periods. Tasks (and controller' gains) switch between these two periods whenever the plant with highest error changes. Table 5.4 shows the sampling period values considering that task 1 has the largest instantaneous error. Each task can apply two discrete-time controller gains designed using optimal control considering the cost function specified in the framework for the two possible periods.

- On-line FBS - FH [HC05]: this method mandates to switch periods at run-time continuously within the specified range according to the optimization procedure. Switches of tasks periods (and controllers' gains) occur at a given periodicity, called the period of the feedback scheduler $T_{fbs} = 500$ms. Table 5.4 show the sampling period values considering that task 1 has the largest finite-horizon error. Controller gains are an output of the optimization procedure, which give LQ optimal controllers considering the cost function specified in the framework.

- Heuristic self-triggered [VMF03]: in this event-driven method the desired sampling period $h_{k+1}$ for the next task instance execution is heuristically specified as

$$h_{k+1} \quad = \quad (h_{max} - h_{min})\, e^{-K|x_l|} + h_{min} \tag{5.10}$$

where $|x_k|$ is the norm of the state variables, $K$ determines how abrupt are the changes in the sampling period (for this implementation $K = 4$), and $h_{max}$ and $h_{min}$ defines the sampling period range. With this specification small errors (low $|x_k|$ values) produce large sampling periods and viceversa. The sampling period range is $h_{min} = 0.030$s and $h_{max} = 0.090$s, Table 5.4 shows the expected average sampling period. The controller gain $L$ is calculated on-line (each sampling period) using LQ optimal controller design using the cost function specified in the framework.

| Approach | Type | Task 1 | Task 2 | Task 3 | $\eta$ |
|---|---|---|---|---|---|
| Static [ÅW97] | | 0.0300 | 0.0900 | 0.0900 | |
| Off-line FBS [SLSS96] | FBS | 0.0500 | 0.0500 | 0.0500 | |
| On-line FBS-Inst. [MLB+04] | FBS | 0.0300 | 0.0900 | 0.0900 | |
| On-line FBS-FH [HC05] | FBS | 0.0300 | 0.0900 | 0.0900 | |
| Heuristic self-triggered [VMF03] | EDC | 0.0840 | 0.0840 | 0.0840 | |
| Self-triggered [LCH+07] | EDC | 0.0589 | 0.0589 | 0.0589 | 0.45 |
| Optimal Self-triggered [MVB09] | EDC | 0.0591 | 0.0591 | 0.0591 | 0.50 |

Table 5.6: Experimental tasks sampling periods (seconds)

- Self-triggered [LCH+07]: in this event-driven method, the execution rule, that triggers a control task, has been defined as a function of the measured state,

$$e_k(t)^T M e_k(t) = \eta x_k^T M x_k, \tag{5.11}$$

  where $0 < \eta \leq 1$ specify the relative size of the boundaries, and $M$ defines the shape of the boundaries. Robust control techniques are used to defined the boundary thresholds to ensure stability. Table 5.4 shows the expected average sampling period. In an event-based system $\eta$ can be used to adjust the processor load. In this case $\eta = 0.6$, and it was selected to provide a similar processor load as the FBS approaches. Event-driven control methods cannot use the same LQ optimal techniques because no periodic sampling occurs. Hence, in order to provide a fair performance evaluation, $L$ was designed using an iterative optimization algorithm. Given a specific $\eta$, and according to the plant dynamics, the cost function and the boundary shape $M$, the implemented algorithm searches for an optimal $L$ value that provides the minimum control cost.

- Optimal self-triggered [MVB09]: in this event-based method, the execution rule is also defined as a function of the measured state as specified by (5.11). However, in this method the boundary $M$ is considered as an optimization variable, therefore the optimal control gain $L$ and the optimal boundary $M$ are obtained from an iterative algorithm given a specific $\eta$, and according to the plant dynamics and the cost function. Table 5.4 shows the expected average sampling period. For this approach it was set $\eta = 0.65$ in order to provide a similar processor load than the case of the FBS approaches.

#### 5.4.1.2 Simulation Results

This section summarized the results obtained from the simulation of the different methods. Table 5.5 present the control performance and the resource utilization results for each FBS and EDC method.

During the simulation it is assumed that there are not timing variations caused by the task scheduling, i.e. jitters. As demonstrated by [LMVF08], the jitters degrade control performance and may hide the true performance that can be achieved by the different FBS and EDC methods. Therefore for simulation purposes, the degrading effects caused by the jitters were removed completely.

Analyzing the control performance method by method, it is observed that the static method provides the worst performance, as expected. Then considering only the FBS methods, the on-line algorithms provide better control performance. The three EDC methods have a similar control performance, however the heuristic approach has the best resource utilization efficiency. Comparing

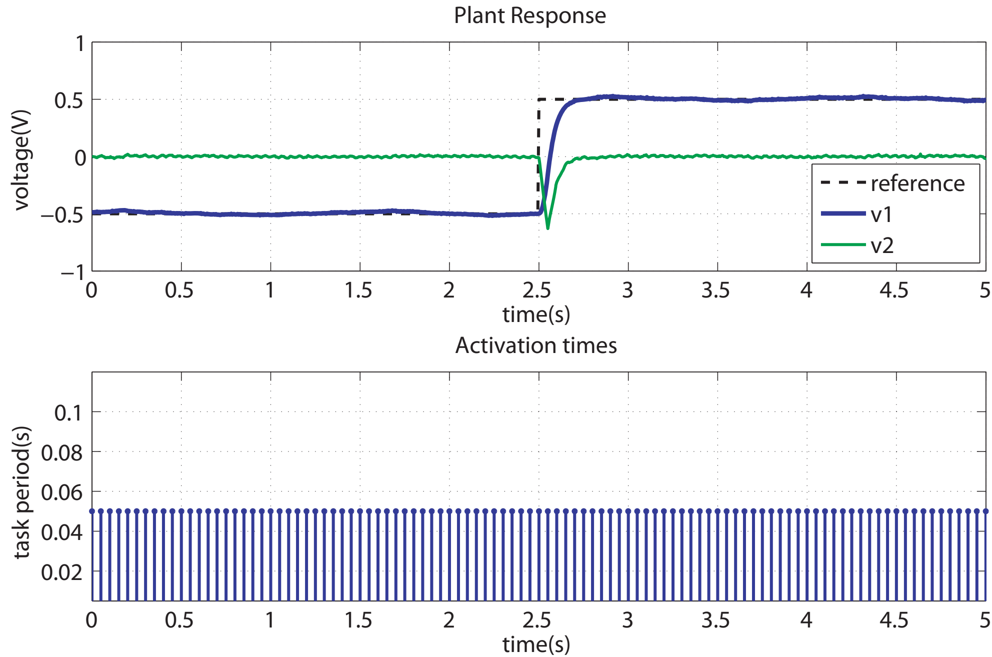



Figure 5.17: Off-line FBS plant response and activation times.

on-line FBS versus EDC methods, it can be noticed that FBS approaches have better control performance, but the EDC methods have better results in terms of resource utilization. Specifically the heuristic self-triggered approach has the best overall control performance.

Notice that computation overhead cannot be derived from the resource utilization performance since it was assumed that the execution time for every task is always 10ms, so the calculation complexity is not taken into account. Computation overhead will be analyzed later when the experimental results are presented.

It is important to highlight that these results are not intended to be definitive in the sense that always one method will perform better than other, even though similar results were obtained in the experimental platform. But instead, these results can be taken as an indicator of the potential of some approaches and to provide valid information to discuss the benefits and drawbacks of each tendency under specific circumstances.

### 5.4.2 Experiments

The experimental platform corresponds to the one specified in the performance evaluation framework. Three double-integrator circuits are used as the plants that are being controlled by the three control tasks, running in the dsPIC33 microcontroller on the Erika real-time kernel. The control tasks can be executed either on a time-triggered basis or on a event-triggered basis, depending on the resource/performance-aware policy under evaluation. The total simulation time is 25s and each control loop has a set-point change every 3s.

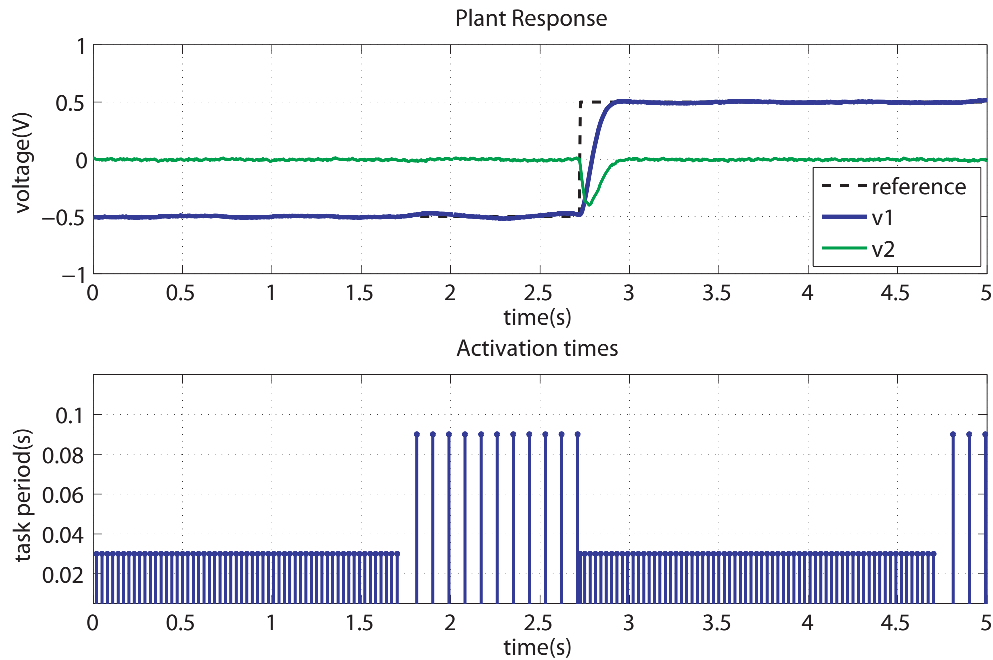



Figure 5.18: On-line FBS-Inst. plant response and activation times.

#### 5.4.2.1 Tasks settings for each method

The task periods for FBS methods are the same as the one used during simulations. However the $\eta$ values for the EDC methods are different compared with the simulation values. The $\eta$ values selected in the experiments were chosen in order to obtain similar control performance compared with the FBS on-line methods. The EDC off-line optimization process was conducted over the real plant dynamics. This is summarized in Table 5.6.

A detailed description of how the different approaches were implemented in the experimental platform is presented next:

- Static approach: tasks' periods are heuristically selected. No optimization process, either on-line or off-line, is executed. The real execution time of each control job is 0.214ms.

- Off-line FBS [SLSS96]: tasks' periods with their corresponding controller gain $L$ are obtained from an off-line optimization process. These values remain constant during the complete experiment. The time spent by the microcontroller for each control job is also 0.214ms. Figure 5.17 shows the plant transient response and activation times for one control loop. Notice that task's activation periods are always fixed.

- On-line FBS - Inst. [MLB+04]: two available task periods ($h_{max}$,$h_{min}$) are defined with their corresponding controller gain ($L_{max}$,$L_{min}$), then $h_{min}$ is assigned to the task with the plant highest error, and $h_{max}$ to the other to tasks. An on-line optimization task is executed every 500ms to obtain the current error from the three plants and reassign task periods and controller gains if necessary. The execution time for each control job is 0.214ms and for each optimization job is 0.039ms. Figure 5.18 shows the plant transient response and activation

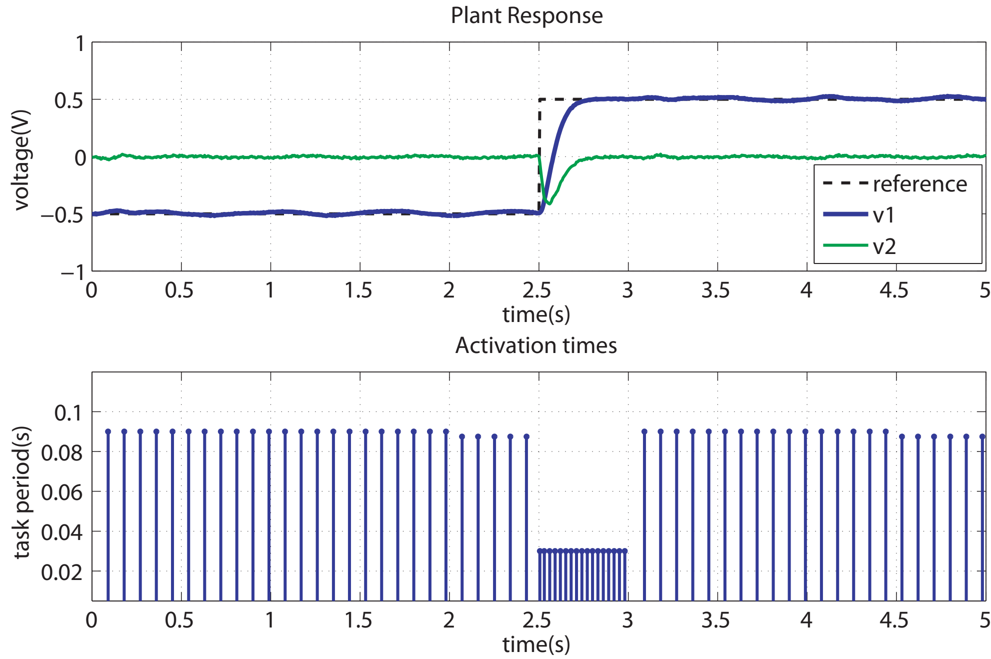



Figure 5.19: On-line FBS-FH plant response and activation times.

times for one control loop. Notice that task periods are smaller when a set-point changes occur, since the plant presents the highest error.

- On-line FBS - FH [HC05]: in this method the on-line optimization task is also executed periodically every 500ms. This task obtains the finite-horizon error from the three plants and then selects appropriate periods for each task according to the optimization procedure. The execution time for each control job is 0.214ms, meanwhile the time spent by the microcontroller in each optimization job is 1.459ms. This time can be reduced to 0.097ms when look-up tables are used to simplify the calculation complexity in the optimization task. This requires storing in a table task periods and controller gain values. As illustrated in Figure 5.19, task periods are smaller when there is a change in the set-point, then the periods change to higher values once the plant reaches the set-point.

- Heuristic self-triggered [VMF03]: the three control task are configured as aperiodic. Each task defines its own next activation time by using the heuristic equation (5.10). Once the $h_{n+1}$ is defined, its corresponding controller gain $L$ is selected. The execution time of each control job is 0.256ms which is a few milliseconds higher than in the FBS methods. Figure 5.20 shows the plant transient response and activation times for one control loop. Since $h_{n+1}$ depends on the norm of the state variables, for large errors small task periods are obtained. A value of 4 is assigned to $K$ in order to have a fast response when set-point changes occur. The sampling period range is given by $h_{min} = 0.030$s and $h_{max} = 0.090$s.

- Self-triggered [LCH+07]: an iterative optimization algorithm is executed off-line in order to obtain an optimal boundary $M$ and an optimal controller gain $L$. In this approach $L$ remains

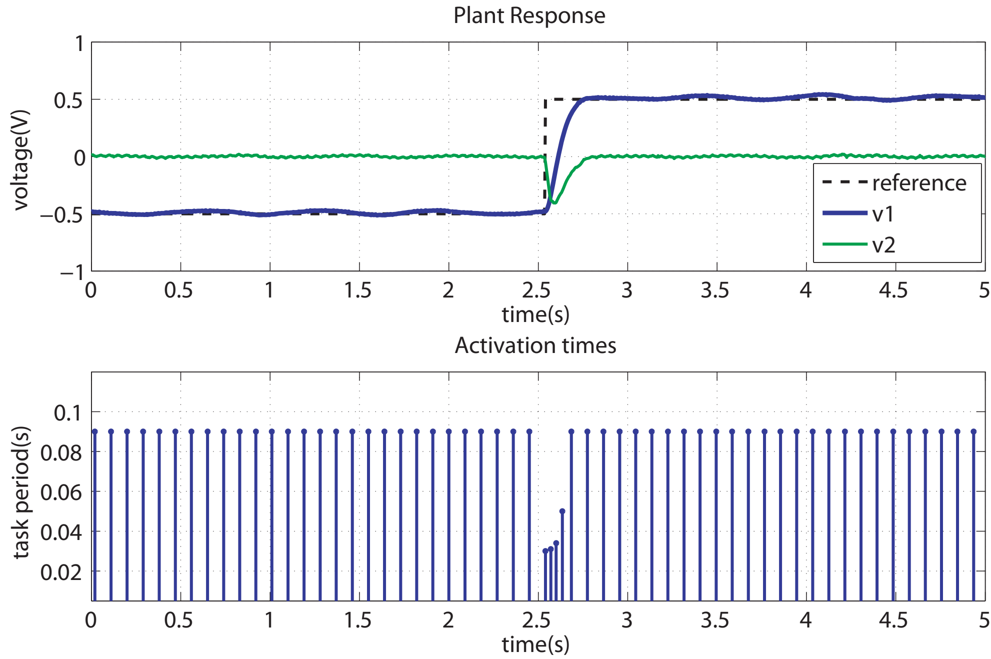



Figure 5.20: Heuristic self-triggered plant response and activation times.

fixed during the complete experiment and $\eta$ is selected to obtain a similar control performance as the FBS on-line methods. Each one of the three non-periodic tasks calculates its next activation time from the measured state using the event condition defined by equation (5.11). The next activation time routine can be implemented by using pre-computed values of the explicit solution given by [VMB08], or by obtaining the smallest positive root from this second order approximation equation provided by [VMF$^+$10],

$$t = \frac{\sqrt{-4[A_{cl}x_k]^T M[A_{cl}x_k](-\eta)x_k^T M x_k}}{2[A_{cl}x_k]^T M[A_{cl}x_k]} \tag{5.12}$$

where $A_{cl} = (A - BL)$. The pre-computed values solution spends less processor time (0.225ms) in comparison with the second order approximation solution (0.419ms). However the pre-computed solution is less flexible since it only supports fixed magnitude set-point changes. As illustrated in Figure 5.21, the activation times pattern of this self-triggered approach has an oscillating behavior, as already indicated by [VML08]

- Optimal self-triggered [MVB09]: an iterative optimization algorithm is executed off-line to obtain an optimal $L$ regardless the boundary shape $M$. Also here $L$ remains fixed during the complete experiment and $\eta$ is selected to obtain a similar control performance than the on-line FBS methods. As in the Self-triggered approach, the next activation time routine can be implemented by using pre-computed values or by solving equation (5.12). The first one has an execution time of 0.225ms for each job while the second spends 0.419ms. Figure 5.22 shows an activation times pattern that starts with small periods when a set-point change

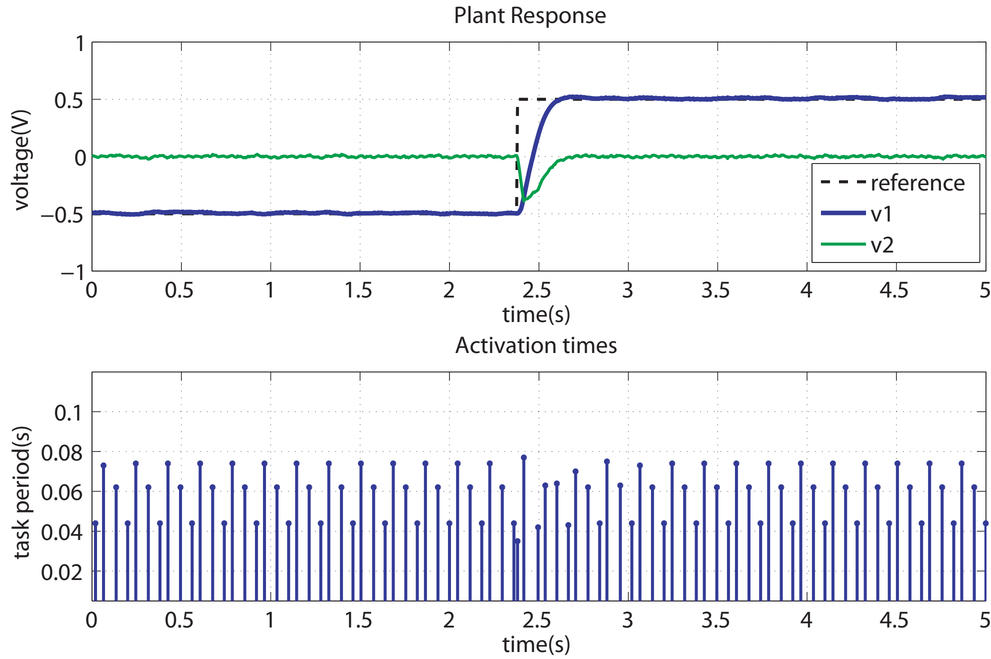



Figure 5.21: Self-triggered plant response and activation times.

occurs. Then one large period is introduced, and later on the periods are lineally increasing. This pattern differs from the Self-triggered approach since the boundary shapes are different.

#### 5.4.2.2 Experimental Results

This section summarizes the experimental results obtained from the evaluation of the different methods. Table 5.7 presents the control performance and the resource utilization results for each FBS and EDC method.

By comparing simulation results (Table 5.5) with the experiment results (Table 5.7), it can be notice that control performance results are similar. However resource utilization for simulation is higher compared with the experiment results. This is because in simulation a high execution time value was assumed (10ms) meanwhile for experiments the actual execution time spent by the microcontroller is measured. Therefore, resource utilization results for the experiments reflects the real load in the microcontroller.

Analyzing the control performance method by method, it is observed that the static method again provides the worst control performance. The on-line FBS methods provides the best control performance, while the EDC methods lie in the middle between the on-line FBS and the off-line FBS performances. The best overall control performance results is obtained when On-line FBS-Inst. approach is executed.

Now lets analyze the resource utilization results. By considering the Static approach as a reference (1.1888), it can be noticed that the On-line FBS-Inst. has a similar resource utilization (1.1951), meaning that the execution of the on-line optimization algorithm is efficient in terms of processor time, however in the On-line FBS-FH the on-line algorithm increases the utilization

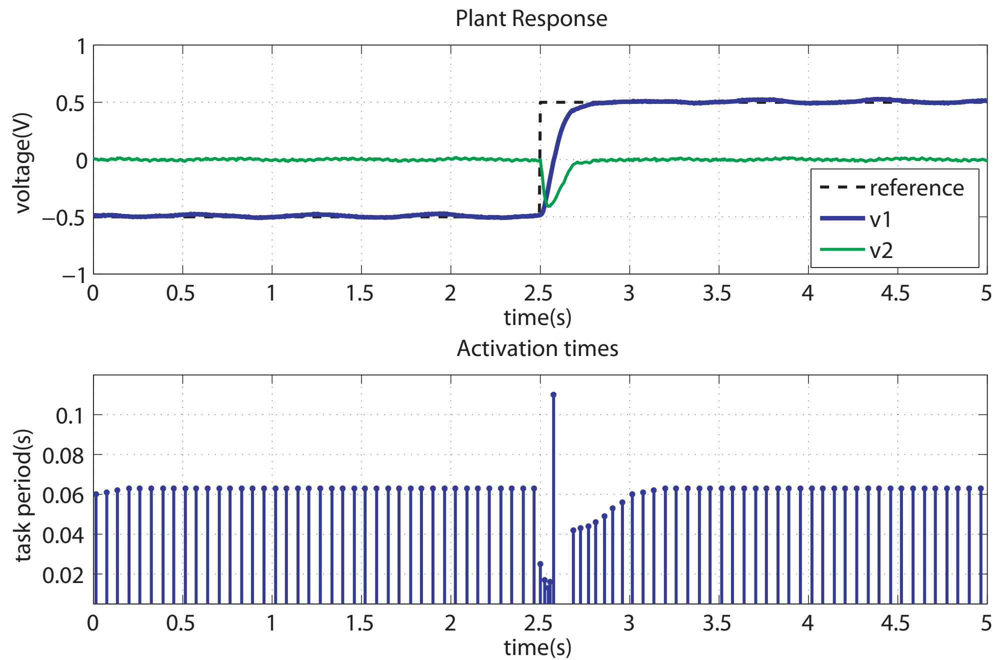



Figure 5.22: Optimal self-triggered plant response and activation times.

considerably due the complexity of the algorithm (1.6254). To solve this problem a look-up table can be used to reduce the processor load (1.2168). The Heuristic self-triggered approach has the best overall resource utilization (0.9153) since less task jobs are triggered compared with the FBS approaches, and because the computation of the next activation time is relatively simple. The Self-triggered and the Optimal Self-triggered have higher processor load if the next activation time value is computed on-line (2.1858, 2.1772 respectively). Therefore pre-computed values have to be used in order to improve the processor load (1.1464, 1.1419 respectively) obtaining better results compared to the Static approach.

### 5.4.3 Discussion

This section has presented simulation and experimental results of selected methods for embedded control systems, by evaluating the control performance and the resource utilization. The evaluation of the different methods, reveals the following key aspects.

- FBS and EDC have the ability to improve control performance with respect to the standard approach for real-time implementation of control loops.

- On-line FBS methods outperform the off-line FBS one if the task in charge of solving the optimization procedure (the feedback scheduler task) has a lower processor demand.

- Among the FBS methods, the On-line FBS-Inst. approach has the best results considering both control performance and resource utilization.

| Approach | Calculation | Control Performance $J^d_{control}$ | Resource Utilization $J_{resource}$ | Performance Index $PI^d$ |
|---|---|---|---|---|
| Static [ÅW97] | | 2.3090 | 1.1888 | 2.7449 |
| Off-line FBS [SLSS96] | | 1.8331 | 1.2840 | 2.3537 |
| On-line FBS-Inst. [MLB+04] | On-line | 1.4699 | 1.1951 | 1.7567 |
| On-line FBS-FH [HC05] | On-line | 1.4919 | 1.6254 | 2.4249 |
| On-line FBS-FH [HC05] | Look-up table | 1.4919 | 1.2168 | 1.8153 |
| Heuristic self-triggered [VMF03] | On-line | 1.5931 | 0.9153 | 1.4583 |
| Self-triggered [LCH+07] | On-line | 1.6271 | 2.1858 | 3.5563 |
| Self-triggered [LCH+07] | Pre-computed | 1.6271 | 1.1464 | 1.8652 |
| Optimal Self-triggered [MVB09] | On-line | 1.5558 | 2.1772 | 3.3872 |
| Optimal Self-triggered [MVB09] | Pre-computed | 1.5558 | 1.1419 | 1.7765 |

Table 5.7: Control performance and resource utilization experimental results

- In general, on-line FBS methods are capable of providing better control performance compared with the EDC approaches.
- In general, EDC methods have a lower processor load compared to FBS approaches if look-up table strategies are adopted. In other words, the number of executed control job in EDC approaches is lower than in the FBS cases.
- A drawback for EDC in self-triggered form is that the computation of the next activation time at each job execution can be too expensive in terms of resource utilization. Hence simple triggering conditions should be adopted.
- Among the EDC methods, the Heuristic self-triggered approach has the best results considering both control performance and resource utilization.
- EDC methods are a promising approach for networked control systems where the resource limitation is the communication bandwidth. In the networked scenario, bandwidth consumption only maps to the number of executed jobs and it is not affected by the computation of the next activation time (which is the critical point in processor-based control systems).

Also it is important to stress that this evaluation has proven that the theoretical approaches can be implemented in practice. In particular on-line FBS methods and self-triggered EDC methods are capable to provide similar performance. In addition EDC methods can provide a more efficient resource utilization, while the FBS methods provide better control performance. However a deeper performance evaluation and analysis is required in order to understand the benefits of FBS and EDC methods. Specifically the two methods with the best performance/resource utilization (On-line FBS-Inst. and Heuristic self-triggered) have been selected for a detailed evaluation that is presented in the next chapter.

# Chapter 6

# Performance evaluation: a detailed experience

## 6.1 Introduction

This chapter presents an experimental evaluation of two representative resource/performance-aware policies for multitasking real-time control systems. The first one, presented by Martí *et al.* [MLB+04], belongs to the class of FBS policies, where a resource manager is responsible for modifying each control task progress. The second policy, presented by Velasco *et al.* [VMF03], belongs to the class EDC policies, where each control task decides its progress.

In the previous chapter, these two approaches were assessed under the evaluation framework, together with other policies. In this chapter, the evaluation of these two approaches considers a richer set of scenarios, which permits to elaborate on more implementation details, as well as on providing new conclusions that did not arise in the previous analysis.

## 6.2 Problem set-up

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 section discusses the theoretical aspects of two different resource/performance-aware policies for multitasking real-time control systems. Then, the demands that each policy poses to the computing platform are analyzed

### 6.2.1 Theoretical aspects

Consider the embedded control 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]

$$\begin{aligned} \dot{x}^i(t) &= A^i x^i(t) + B^i u^i(t) \\ y^i(t) &= C^i x^i(t), \end{aligned} \tag{6.1}$$

[1] Henceforth, within this chapter, the superscript $i$ will specify a control loop, identifying either the controller or the controlled plant.

where the state variables $x^i(t) = [x^i_1(t), \dots, x^i_n(t)]$ denote the system state at time $t$. For each plant, the norm of its state variables is defined as the plant error

$$e^i(t) = |x^i(t)|. \tag{6.2}$$

In terms of timing constraints, each control task is characterized by its period $h^i$ (corresponding to 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

$$U = \sum_{i=1}^{n} \frac{c^i}{h^i}, \tag{6.3}$$

which is the fraction of processor time spent in the execution of the task set [But05]. 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

$$r^i = \frac{c^i}{h^i}, \tag{6.4}$$

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 resource/performance-aware 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 FBS (also named "*coordinated*") approach and the EDC (also named "*self-triggered*") approach is to decide for each control task how the rate should be increased (i.e., how the period should be shortened),

$$r^i = r^i_{min} + \Delta r^i \tag{6.5}$$

such that the overall control performance is improved subject to

$$\sum_{i=1}^{n} \Delta r^i \leq U_s. \tag{6.6}$$

Constraint (6.6) specifies that the additional resources $\Delta r^i$ 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 (6.2), the higher the rate $r^i$ (or the shorter the

[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.

sampling period $h^i$) to be allocated.

- For the set of control tasks, a given constraint on the utilization factor must be kept (6.6).

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.

#### 6.2.1.1 FBS approach

In the FBS approach [MLB+04], each control task $\tau^i$ 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. [CEBÅ02]. As explained in [MLB+04] 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 [MVB09] 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 [CMV+06] 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 [MLB+04] is a good choice.

For a given set of $n$ control tasks, $\tau^1, \ldots, \tau^n$, the FBS approach was formulated as a constrained optimization problem

$$\begin{aligned} \text{maximize} \quad & \textstyle\sum_{i=1}^{n} w^i e^i p^i(r^i) && (6.7) \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{n} \Delta r^i \leq U_s && (6.8) \\ & \Delta r^i \geq 0 \end{aligned}$$

where the solution are the rate increments $\Delta r^i$, $i = 1, \ldots, n$, and weights $w^i$ in (6.7) can be used to permit appropriate comparisons between control loops.

The solution to (6.7)-(6.8) 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 FBS approach tackles the slack redistribution problem considering that a coordinator, i.e. resource manager or feedback scheduler, 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 (6.1) using classic design procedures. In particular, let

$$x_{k+1} = \Phi(h)x_k + \Gamma(h)u_k \tag{6.9}$$

be the discretization of the system equation in(6.1) [ÅW97]. The input is given by

$$u_k = -L(h)x_k \tag{6.10}$$

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 [MLB+04] and its extension [MVB09], and references therein for further details on the optimization, controller design and stability analysis.

#### 6.2.1.2 EDC 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 [VMF03]. Although the analysis in this thesis 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.

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 [VMF03]. 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 (6.9) by imposing the desired slack management dynamics

$$\begin{bmatrix} x_{k+1} \\ h_{k+1} \end{bmatrix} = \begin{bmatrix} \Phi(h) \cdot x_k \\ \Upsilon(x_k, U_s) \end{bmatrix} + \begin{bmatrix} \Gamma(h) \\ 0 \end{bmatrix} u_k,$$

in terms of a new state variable: the desired sampling period $h_{k+1}$ 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 $h_{k+1}$ in [VMF03] were heuristically specified as

$$h_{k+1} \quad = \quad (h_{max} - h_{min})\, e^{-K|x_k|} + h_{min} \tag{6.11}$$

in such a way that it mathematically behaves as required by the problem specifications. If there is no error ($|x_k| = 0$), then $h_{k+1} = h_{max}$. 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 $x_k$, (6.11) 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

$$h_{min} = \frac{c}{U_s + r} \tag{6.12}$$

where $r$ is the current task rate as defined in (6.4). 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 (6.11) is nonlinear. Note that this type of non-linear models is common for self-triggered control approaches [AT08a], [WL09a]. 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 (6.10), equal to the case of the coordinated approach (the same type of control

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

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.

## 6.3 A FBS and EDC implementation

The implementation of the FBS and EDC approaches has to take into account that the main variables that have to be considered for solving the slack redistribution problem, specified either in (6.7)-(6.8) or in (6.11) respectively, 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. Hence, both approaches to resource/performance-aware 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.

### 6.3.1 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, $x^i$. With both information, the new rate of progress, in the form of the next sampling period $h^i$, is calculated using (6.11). 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 [MLB$^+$04], 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 $x^i$ 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 $\Delta r^i$, which corresponds to the solution (6.7)-(6.8). Having all $\Delta r^i$, 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. [CEBÅ02], [HC05] or [CMV$^+$06], 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 $x^i$ 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.

### 6.3.2 Code implementation details

The performance evaluation framework, described in Subsection 5.3.3, was used to conduct the experiments. The framework was configured with three double integrator circuits controlled by three control tasks executing on top of the Erika kernel using the Full Flex board hardware with a dsPIC33 microcontroller. For the implementation of both policies, the exchange of information

**Algorithm 9**: `void Periodic_controller_task(void)`

```
begin
    x_1^i=read_input()
    x̂^i=observer(x_1^i,t_a^i)
    u^i=calculate_output(x̂^i,h^i)
end
```

Figure 6.1: Pseudo-code for a standard periodic control task

between kernel and control tasks is achieved by means of a system call and by accessing shared memory.

#### 6.3.2.1 Standard controller

The task model for implementing control tasks is the one-shot task model (see Chapter 2). As a reference Figure 6.1 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 $x_1^i$, observes the state (which is the implementation of the one shot task model) $\hat{x}^i$ at the actuation instant $t_a^i$, 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 $t_a^i$ relative to the job release time. In particular, in the implementation, $t_a^i = h^i$, 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.

#### 6.3.2.2 FBS implementation

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

Figure 6.2 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 6.1) 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 resetting 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 [LB05] 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., $|x_k| = 0$, 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 $|x_k| < 0.05V$, then the plant is considered to be in the equilibrium.

#### 6.3.2.3 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 6.3 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

**Algorithm 10**: `void Coordinated_controller_task(void)`

```
begin
    h^{i,next}=read_shared_memory()
    x_1^i=read_input()
    x̂^i=observer(x_1^i,t_a^i)
    u^i=calculate_output(x̂^i,h^{i,next})
    write_shared_memory(x̂^i)
end
```

**Algorithm 11**: `void Feedback_scheduler_task(void)`

```
begin
    U_s=read_shared_memory()
    for (each_control_task)
    begin
        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})
    end
end
```

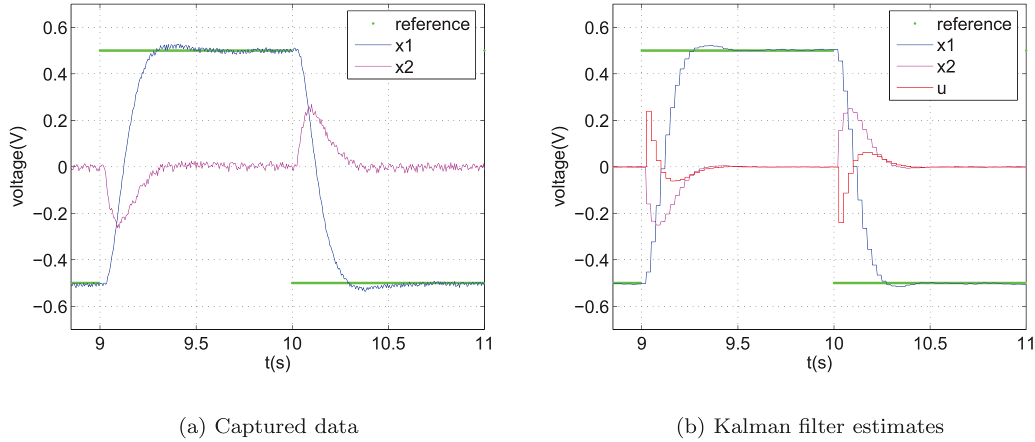
Figure 6.2: Pseudo-code for the two tasks coordinated approach

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 $|x_k| \geq 0.05V$. 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.

**Algorithm 12**: `void Selftriggered_controller_task(void)`

```
begin
    U_s=read_shared_memory()
    x_1^i=read_input()
    x̂^i=observer(x_1^i,t_a^i)
    h^{i,next}=Υ(x̂^i,U_s)
    u^i=calculate_output(x̂^i,h^{i,next})
    set_rel_alarm(h^{i,next})
end
```

Figure 6.3: Pseudo-code for the self-triggered approach




(a) Captured data

(b) Kalman filter estimates

Figure 6.4: Experimental data for observer design

### 6.3.3 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 (5.8) with the validated components' values, which minimizes a discrete cost function equivalent to the continuous cost function

$$J = \int_0^\infty \left(x^T(t)Qx(t) + u^T(t)Ru(t)\right) dt \tag{6.13}$$

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 [ÅW97].

For the observer design, a Kalman filter was designed taking into account the following noise covariances $Q_k = E(w \cdot w^T) = 1 \cdot 10^{-10}BB^T$ and $R_k = E(v \cdot v^T) = 5 \cdot 10^{-5}$, extracted from the electronic circuit, where $w$ and $v$ are the plant noise and the measurement noise, respectively.

In particular, for a sampling period $h = 25$ms, Figure 6.4 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 subfigure shows the plant response by plotting the reference signal and the two state variables. The bottom subfigure 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.

### 6.3.4 Workload generation

In the following experiments, the three control tasks controlling each double integrator circuit implemented the same optimal controller described in Section 6.3.3 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

Table 6.1: Worst case execution times

| | Operations | Static | Coord. | Self |
|---|---|---|---|---|
| 1 Controller | read_input | 28$\mu$s | 28$\mu$s | 28$\mu$s |
| | observer | 116$\mu$s | 116$\mu$s | 116$\mu$s |
| | calculate_output | 48$\mu$s | 48$\mu$s | 48$\mu$s |
| | compute $e^i$ | | | 3$\mu$s |
| | compute $\Upsilon()$ | | | 61$\mu$s |
| | set_rel_alarm | | | 5$\mu$s |
| | other | 5$\mu$s | 5$\mu$s | 5$\mu$s |
| Total | | 197$\mu$s | 197$\mu$s | 266$\mu$s |
| Feedback scheduler | compute max $e^i$ | | 14$\mu$s | |
| | 3 set_ref_alarm | | 15$\mu$s | |
| Total | | | 29$\mu$s | |

Table 6.2: Sampling periods in the experiments

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

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 Subsection 6.3.2.1) and is used to examine the overall performance benefit of adaptive slack redistribution allocation.

#### 6.3.4.1 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.

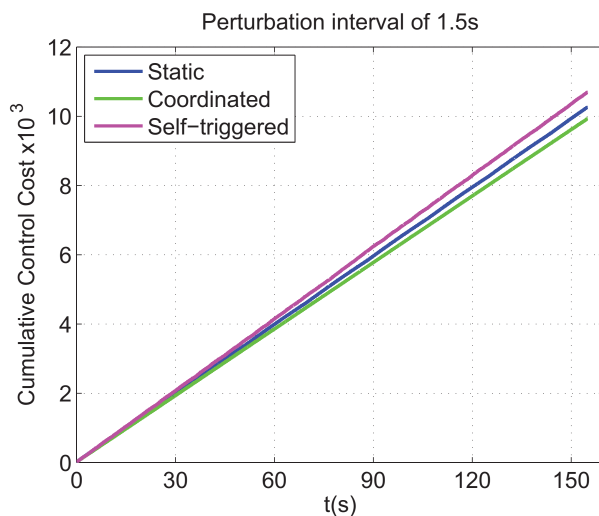



Figure 6.5: Cumulative cost for the three policies

#### 6.3.4.2 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 6.1 shows the main time consuming operations for each method, which have been measured using an oscilloscope plugged to the board.

For each control task of the coordinated policy, the shortest and longest sampling periods are $h_{min}^{i,coord} = 25$ms and $h_{max}^{i,coord} = 50$ms 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 $h^{fs} = 25$ms. Therefore, using the numbers of Table 6.1, 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

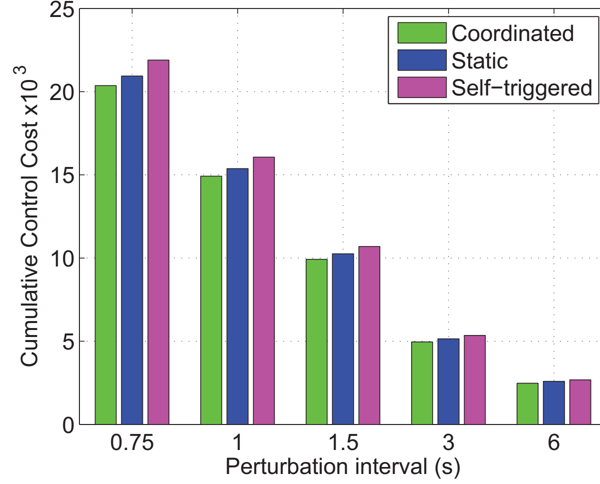



Figure 6.6: Cumulative control cost histogram for the three policies and for all the perturbation intervals

usage than the coordinated policy, that is

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

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

As a summary, Table 6.2 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.

### 6.3.5 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

$$J_{control}^{d} = \sum_{n=0}^{t_{eval}} \left[ x^T(n) Q_d x(n) + 2x^T(n) N_d u(n) + u^T(n) R_d u(n) \right], \tag{6.14}$$

where the $Q_d$,$N_d$ and $R_d$ represent the discrete cost weighting matrices (Appendix A describes how to obtain these matrices from $Q$ and $R$ of equation (6.13)). Each $n$-state and $n-$control signal is extracted from the board every 5ms. The $n$-subscript rather than the $k$-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.

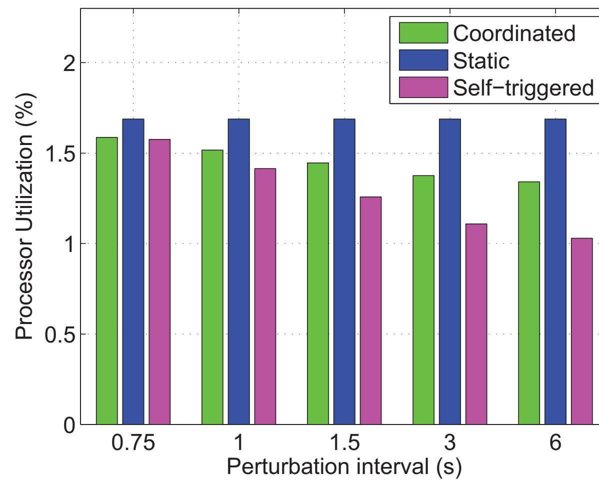



Figure 6.7: Processor usage histogram for the three policies and all the perturbation intervals

## 6.4 Results

Results of this evaluation are presented as follows. Subsection 6.4.1 compares the coordinated and self-triggered approaches versus the static approach in terms of control performance, and Subsection 6.4.2 focuses on processor utilization. Subsection 6.4.3 summarizes the experimental results and Subsection 6.4.4 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 Subsection 6.4.5.

### 6.4.1 Control performance analysis

Figure 6.5 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 6.6 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 6.6 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 Subsection 6.3.4, 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.

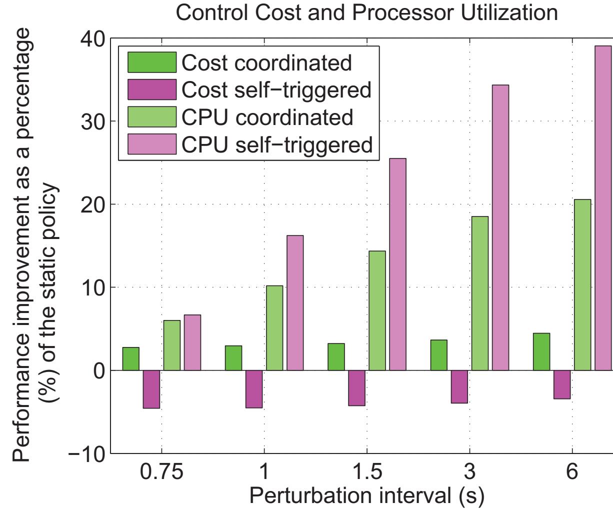



Figure 6.8: Control performance and processor usage improvement as a percentage (%) of the static policy

### 6.4.2 Resource utilization analysis

Figure 6.7 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.

### 6.4.3 Summary of the experimental results

Figure 6.8 gives a complementary view of the experimental results. It shows the previous control performance and processor usage analysis for the coordinated 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

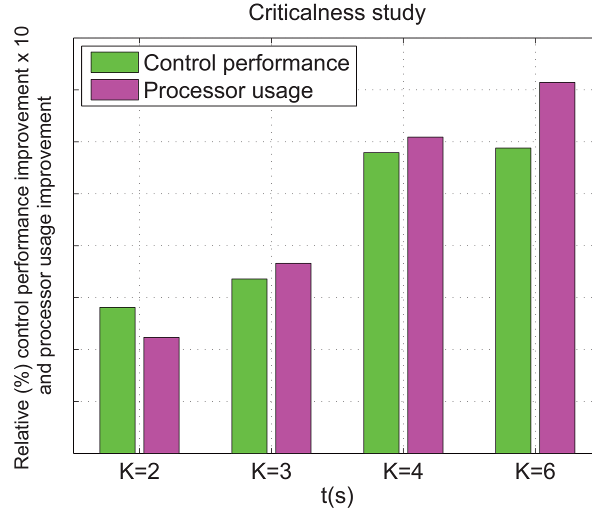



Figure 6.9: Criticalness ($K$) study: control performance improvement and processor time savings relative to the case $K = 1$

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.

### 6.4.4 Overhead analysis

The processor utilization analysis presented in Subsection 6.4.2 implicitly incorporate the overhead analysis introduced by each slack redistribution policy. That is, in Figure 6.7, 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 6.2 and the time measures shown in Table 6.1 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 6.1, in the worst case scenario, during 50ms, the coordinated policy uses 58$\mu$s for the slack redistribution (two executions of the feedback scheduling task) while the self-triggered uses 276$\mu$s (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 $h^{i,coord}_{max}$, 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

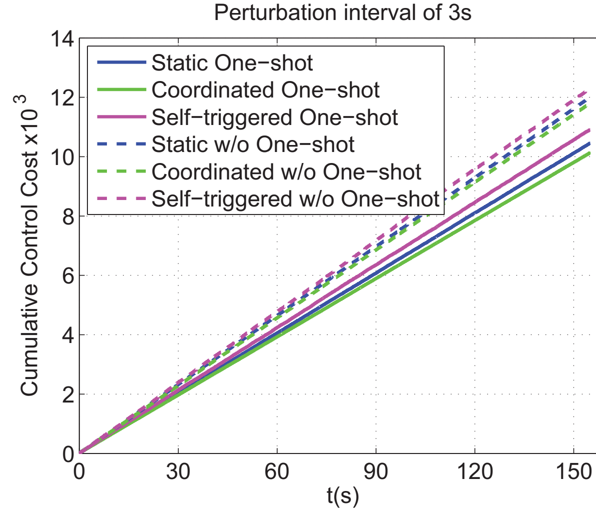



Figure 6.10: Jitter evaluation

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 (6.11)), 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.

### 6.4.5 Key points

In all the previous results, the criticalness parameter in the self-triggered policy was $K = 4$. Figure 6.9 shows for different values of $K$ the relative control 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

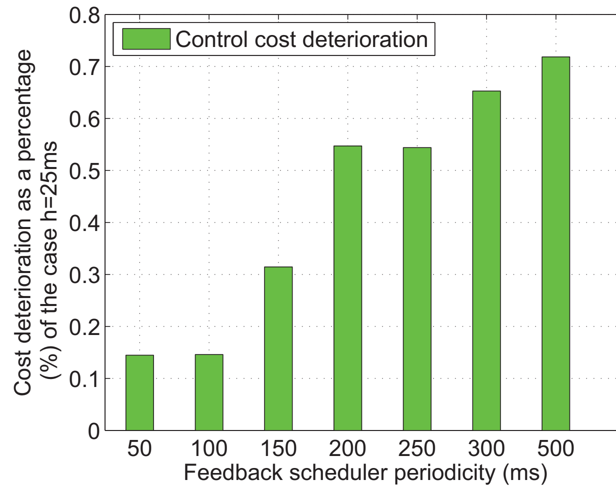



Figure 6.11: Deterioration of the cumulative control cost as a function of the period of the feedback scheduler task.

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 6.10 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.

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 respect to control performance is shown in Figure 6.11, 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 [HC05].

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. Figure 6.12 shows in the top subfigure the experimental numbers of control performance measured using (6.14) 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 subfigure. 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 subfigure 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.

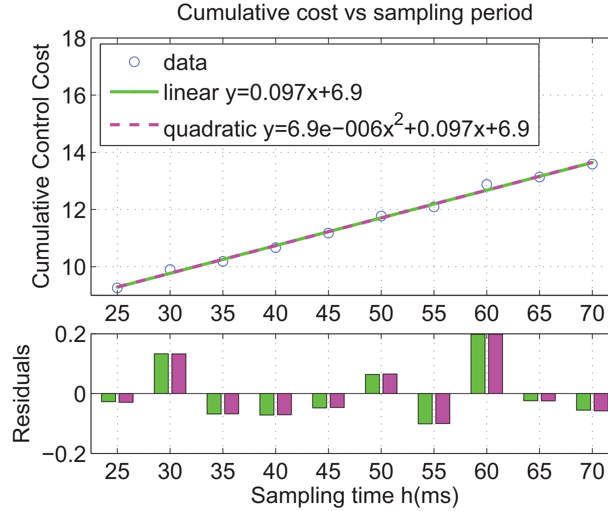



Figure 6.12: Statistical analysis of the linear performance benefit.

### 6.4.6 Discussion

The lesson learned from the experimental evaluation of the coordinated and self-triggered approach to slack redistribution in real-time embedded 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.

# Chapter 7

# Conclusions

This thesis provides two major contributions in the area of real-time embedded control systems: 1) presents the analysis, design and implementation of a novel control task model, named one-shot, capable of accommodating diverse timing requirements while improving control performance; 2) introduces the development of a performance evaluation framework for resource-constrained real-time embedded control systems that allows the evaluation of different control and resource management strategies. The conclusions are divided into two main categories.

## 7.1 One-shot task model

Sampling and latency jitters represent a well-known problem for the real-time computing of control systems. Jitters degrade control performance and although different solutions have been proposed from the control perspective and from the real-time perspective, few solutions have considered both aspects.

A novel task model for real-time control systems has been presented that combines both aspects. This task model is synchronized at the actuation instants rather than at the sampling instants. This has been shown to provide interesting properties. From the scheduling point of view, the new task model can be seamlessly integrated into existing scheduling theory and practice, while minimizing the hardware interrupts required by previous solutions, which in turn improves task set schedulability. From a control perspective, the one-shot task model absorbs jitters because it allows irregular sampling, and improves reactiveness in front of perturbations, which even permits to achieve better performance than the case where controllers would be executed in isolated processors. When compared with other task models, the one-shot obtains the best overall control performance. In general, simulations and control experiments have reported results which corroborate the promised benefits and show the implementation feasibility of the one-shot task model.

The application of the one-shot task model has been extended to the case of noisy measurement. In this situation two problems can deteriorate the control performance: jitters and noise. The proposed solution is to embed the Kalman filter into the one-shot task model. Simulation and experimental results over a control loop with a noisy plant and scheduling jitters have demonstrated that their integration preserves their own benefits: noise removal and jitters' effects elimination.

Since the Kalman filter was adapted to the case of irregular sampling, two different Kalman implementation approaches were presented. The first approach considers the complete sampling period to implement the Kalman algorithm, and the second approach implement the Kalman correction just from sampling to actuation instants. Similar control performance results were obtained from both approaches, however the first one demands slightly more resources than the second one.

## 7.2 Performance evaluation framework

Feedback scheduling (FBS) and event-driven control (EDC) represent the main tendencies on resource/performance-aware policies for embedded control systems. Although most of the existing research has focused on proposing new methods to improve control performance or resource utilization, only few works have considered the practical implementation aspects.

A performance evaluation framework has been implemented to allow the deployment and evaluation of different resource/performance-aware policies under similar circumstances. The framework design is the result of the taxonomical analysis of different resource management and control optimization strategies. Among other specifications, the framework supports different triggering paradigms (event-based, time-trigger), different optimization algorithms (on-line, off-line), different evaluation parameters (control performance, resource utilization), and different sampling periodicity (static periods, varying periods, aperiodic intervals, sequences).

The performance evaluation framework is composed by a simulation platform and by an experimental platform. Each platform has been designed considering five functional modules (baseline configuration, resource manager, optimization algorithms, task controller and performance measurement). To validate the correct operation of the framework, a group of four representative feedback scheduling methods and three event-driven control methods were evaluated using the framework. The results indicate that on-line FBS and EDC methods have the ability to improve control performance with respect to the standard approach. The best control performance is achieved by the on-line FBS methods while EDC approaches are more efficient in terms of resource utilization.

Based on these initial results, two representative resource/performance-aware approaches (the FBS with the best performance and the EDC with the best resource utilization) were selected for a detailed experimental evaluation using the performance evaluation framework.

The selected approaches represent two alternative policies for multitasking real-time control systems: "*coordinated*" (FBS) vs. "*self-triggered*" (EDC). In the coordinated policy a resource manager is responsible for modifying each control task progress. On the contrary, in the self-triggered policy, each control task decides its progress. In terms of performance, the coordinated approach provided higher benefits, as expected. And in terms of resource utilization, both policies showed 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.

## 7.3 Future work and open problems

Future works will focus in the following aspects:

- Extend the use of the one-shot task model in the context of networked control systems. Specifically the one-shot integrated with the Kalman filter can be an interesting solution for control systems with network delays.
- Enhance the performance evaluation framework services to support network communication. Then evaluate and compare the performance of different implementation approaches for networked control systems.
- Event-driven control systems present interesting research challenges mainly due the lack of a mature system theory and the promise to optimize resource consumption. Therefore topics such as event-driven scheduling policies, optimal event-boundaries formulations and computational load regulation constitute significant open problems in the field of resource-constrained real-time embedded control systems.

# References

[10096] POSIX.13 (1998). IEEE Std. 1003.13-1998. Information technology -standardized application environment profile- posix realtime application support (aep), 1996.

[AB02] K. J. Astrom and B. M. Bernhardsson. Comparison of riemann and lebesgue sampling for first order stochastic systems. *Proceedings of the 41st IEEE Conference on Decision and Control*, Dec. 2002.

[ABRW94] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. STRESS: A simulator for hard real-time systems. *SoftwarePractice and Experience*, 24:543–564, 1994.

[ÅBW05] Karl-Erik Årzén, Anders Blomdell, and Björn Wittenmark. Laboratories and real-time computing. *IEEE Control Systems Magazine*, 25(1):30–34, February 2005.

[AC99] P. Albertos and A. Crespo. Real-time control of non-uniformly sampled systems. *Control Engineering Practice*, 7(4):445–458, 1999.

[ÅC05] Karl-Erik Årzén and Anton Cervin. Control and embedded computing: Survey of research directions. In *Proc. 16th IFAC World Congress*, Prague, Czech Republic, July 2005.

[ÅCES00] K. E. Årzen, A. Cervin, J. Eker, and L. Sha. An introduction to control and scheduling co-design. *39th IEEE Conference on Decision and Control*, 5:4865–4870, 2000.

[ACR+00] P. Albertos, A. Crespo, I. Ripoll, M. Vallés, and P. Balbastre. RT control scheduling to reduce control performance degradation. *39th IEEE Conference on Decision and Control*, 2000.

[ARGH01] M. Aldea-Rivas and M. González-Harbour. Marte OS: an Ada kernel for real-time embedded applications. *Proceesings of the International Conference on Reliable Software Technology, Ada-Europe-2001*, May 2001.

[ART10] ARTEMIS. Advanced Research and Technology for EMbedded Intelligence and Systems. `http://www.artemis.eu`, 2006-2010.

[Årz99] K. E. Årzén. A simple event-based PID controller. In *14th World Congress of IFAC*, Beijin, China, Jan. 1999.

[Årz05] Karl-Erik Årzén. Timing analysis and simulation tools for real-time control. In Paul Pettersson and Wang Yi, editors, *Formal Modeling and Analysis of Timed Systems*, volume 3829 of *LNCS*. Springer, September 2005. Extended astract in the Proceedings of FORMATS 2005, Uppsala. Invited Talk.

[AS90] P. Albertos and J. Salt. Digital regulators redesign with irregular sampling. *IFAC World Congress*, 1990.

[AT08a] A. Anta and P. Tabuada. Self-triggered stabilization of homogeneous control systems. In *American Control Conference*, 2008.

[AT08b] A. Anta and P. Tabuada. Space-time scaling laws for self-triggered control. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 4420 –4425, Dec. 2008.

[AT09] A. Anta and P. Tabuada. Isochronous manifolds in self-triggered control. In *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, 2009.

[AT10] A Anta and P Tabuada. To sample or not to sample: Self-triggered control for nonlinear systems. *Automatic Control, IEEE Transactions on*, 2010.

[ÅW97] K. J. Åström and B. Wittenmark. *Computer-Controlled Systems: Theory and Design.* Prentice Hall, New Jersey, USA, third edition, 1997.

[BBGL99] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. *6th International Conference on Real Time Computing Systems and Applications*, pages 62–69, Nov. 1999.

[BBLB03] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. *24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

[BC08] Enrico Bini and Anton Cervin. Delay-aware period assignment in control systems. In *Proc. 29th IEEE Real-Time Systems Symposium*, Barcelona, Spain, December 2008.

[BLA98] G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. *IEEE Real-Time Systems Symposium*, 1998.

[Boj05] Edward Boje. Approximate models for continuous-time linear systems with sampling jitter. *Automatica*, 41(12):2091 – 2098, 2005.

[BRVC04] P. Balbastre, I. Ripoll, G. Vidal, and A. Crespo. A task model to reduce control delays. *Journal of Real-Time Systems*, 27(3), 2004.

[But97] G. Buttazzo. Hard real time computing systems. predictable scheduling algorithms and applications. *Springer*, 1997.

[But05] G. Buttazzo. *Hard Real-Time Coimputing Systems: Predictable Scheduling Algorithms and Applications.* Springer, second edition, 2005.

[But06] G. Buttazzo. Research trends in real-time computing for embedded systems. *ACM SIGBED Review*, 3(3):1–10, July 2006.

[CA06] A. Cervin and P. Alriksson. Optimal on-line scheduling of multiple control tasks: A case study. *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, 2006.

[CE03] A. Cervin and J. Eker. The control server: a computational model for real-time control tasks. *15th Euromicro Conference on Real Time Systems*, page 113, 2003.

[CEBÅ02] A. Cervin, J. Eker, B. Bernhardsson, and K. E. Årzen. Feedback-feedforward scheduling of control tasks. *Real Time Systems*, 23(1-2):25–53, Nov. 2002.

[Cer01] A. Cervin. Improved scheduling of control tasks. *11th Euromicro Conference on Real Time Systems*, pages 4–10, June 2001.

[CHL+03] Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and Karl-Erik Årzén. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.

[CLS03] R. Chandra, X. Liu, and L. Sha. On the scheduling of flexible and reliable real-time control systems. *Real Time Systems*, 24(2):153–169, March 2003.

[CMC+04] M. Cirinei, A. Mancina, D. Cantini, P. Gai, and L. Palopoli1. An educational open source real-time kernel for small embedded control systems. *Computer and Information Sciences - ISCIS 2004*, pages 866–875, 2004.

[CMV+06] R. Castané, P. Mart; M. Velasco, A. Cervin, and D. Henriksson. Resource management for control tasks based on the transient dynamics of closed-loop systems. *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, 2006.

[CVMC10] Anton Cervin, Manel Velasco, Pau Martí, and Antonio Camacho. Optimal on-line sampling period assignment: Theory and experiments. *IEEE Transactions on Control Systems Technology*, Accepted for publication, June 2010.

[EHÅ00] J. Eker, P. Hagander, and K. E. Årzen. A feedback scheduler for real-time controller tasks. *Control Engineering Practice*, 2000.

[Ell59] P. Ellis. Extension of phase plane analysis to quantized systems. *Automatic Control, IRE Transactions on*, 4(2):43–54, 1959.

[Fou10] The Eclipse Foundation. Eclipse foundation homepage. `http://www.eclipse.org`, 2010.

[FSR04] E. Fridman, A. Seuret, and J.-P. Richard. Robust sampled-data stabilization of linear systems: an input delay approach. *Automatica*, 40(8):1441–1446, 2004.

[GAGB01] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[GcH09] M.E.-M. Ben Gaid, A.S. Çela, and Y. Hamam. Optimal real-time scheduling of control tasks with state feedback resource allocation. *IEEE Transactions on Control Systems Technology*, 17(2):309 – 326, mar 2009.

[GCHI06] M. M. Ben Gaid, A. Cela, Y. Hamam, and C. Ionete. Optimal scheduling of control tasks with state feedback resource allocation. *American Control Conference*, 2006.

[HC05] D. Henriksson and A. Cervin. 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*, pages 4469–4474, Dec. 2005.

[HCÅ02] D. Henriksson, A. Cervin, and K.-E. Årzén. TrueTime: Simulation of control loops under shared computer resources. *15th IFAC World Congress*, 2002.

[HCAÅ02] D. Henriksson, A. Cervin, J. Akesson, and K. E. Årzen. Feedback scheduling of model predictive controllers. *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Application Symposium*, 2002.

[HGvZ+99] W. P. M. H. Heemels, R. J. A. Gorter, A. van Zijl, P. P. J. van den Bosch, S. Weiland, W. H. A. Hendrix, and M. R. Vonder. Asynchronous measurement and control: a case study on motor synchronization. *Control Engineering Practice*, 7(12):1467 – 1482, 1999.

[HHK01] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *First International Workshop on Embedded Software*, pages 116–184, 2001.

[HJC08] Toivo Henningsson, Erik Johannesson, and Anton Cervin. Sporadic event-based control of first-order linear stochastic systems. *Automatica*, 44(11):2890–2895, November 2008.

[HLL+03] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, Department of Electrical Engineering and Computer Science, University of California Berkeley, USA, 2003.

[HSB08] W. P. M. H. Heemels, J. H. Sandee, and P. Bosch. Analysis of event-driven controllers for linear systems. *International Journal of Control*, 81(4), 2008.

[ISIEIS96] 1996 International Standard ISO/IEC 9945-1: 1996 (E) IEEE Std 1003.1. Portable operaring system interface (posix) standard, 1996.

[JHC07] E. Johannesson, T. Henningsson, and A. Cervin. Sporadic control of first-order linear stochastic systems. *Hybrid Systems: Computation and Control*, 2007.

[JS93] K. Jeffay and D.L. Stone. Accounting for interrupt handling costs in dynamic priority tasks systems. *Proc. 14th IEEE Real-Time System Symposium*, 1993.

[Kal60] R.E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME - Journal of Basic Engineering*, 82:35–45, 1960.

[LB05] C. Lin and S.A. Brandt. Improving soft real-time performance through better slack reclaiming. In *26th IEEE Real-Time Systems Symposium*, December 2005.

[LCH+07] M. Lemmon, T. Chantem, X. Hu, , and M. Zyskowski. On self-triggered full information H-infinity controllers. *Hybrid Systems: Computation and Control*, April 2007.

[Lin02] B. Lincoln. Jitter compensation in digital systems. *American Control Conference*, 2002.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LM07] A. Lem and R. McCann. Event-based measurement updating Kalman filter in network control systems. *IEEE Region 5 Technical Conference*, April 2007.

[LMVF08] C. Lozoya, P. Martí, M. Velasco, and J.M. Fuertes. Control performance evaluation of selected methods of feedback scheduling of real-time control tasks. In *17th World Congress of IFAC*, Seoul, Korea, July 2008.

[LSX08] W. Lia, S.L. Shaha, and D. Xiao. Kalman filters in non-uniformly sampled multirate systems: For fdi and beyond. *Automatica*, 44:199–208, January 2008.

[LU10] Department of Automatic Control Lund University. Truetime: Simulation of networked and embedded control systems. http://www.control.lth.se/truetime/, 2010.

[LVM07] C. Lozoya, M. Velasco, and P. Martí. A 10-year taxonomy on prior work on sampling period selection for resource-constrained real-time control systems. In *Work in Progress 19th Euromicro Conference on Real-Time Systems (ECRTS 07)*, Pisa, Italy, July 2007.

[LVM08] C. Lozoya, M. Velasco, and P. Martí. The one-shot task model for robust real-time embedded control systems. *IEEE Transactions on Industrial Informatics*, 4(3), August 2008.

[Mar02] P. Martí. *Analysis and Design of Real-Time Control Systems with Varying Control Timing Constraints*. PhD thesis, Technical University of Catalonia, Pau Gargallo 5, 08028 Barcelona, Spain, June 2002.

[MAT09] Manuel Mazo, Adolfo Anta, and Paulo Tabuada. On self-triggered control for linear systems: Guarantees and complexity. In *10th European Control Conference*, 2009.

[Mat10] MathWorks. Matlab and Simulink for technical computing. http://www.mathworks.com, 2010.

[May79] P.S. Maybeck. *Stochastic models, estimation, and control*. Academic Press, 1979.

[MFRF01] P. Martí, G. Fohler, K. Ramamritham, and J. M. Fuertes. Jitter compensation for real-time control systems. *22nd IEEE Real-Time Systems Symposium*, pages 39–48, Dec. 2001.

[MFVF01] P. Martí, J. M. Fuertes, R. Villà, and G. Fohler. On real-time control tasks schedulability. In *European Control Conference*, pages 2227–2232, Porto, Portugal, Sep. 2001.

[Mic05] Microchip. dsPIC30F/33F Programmer's Reference Manual. http://www.microchip.com, 2005.

[Mic09] Microchip. MPLAB ICD2. http://www.microchip.com, 2009.

[Mic10] Microchip. MPLAB IDE. http://www.microchip.com, 2010.

[Mir07] L. Mirkin. Some remarks on the use of time-varying delay to model sample-and-hold circuits. *IEEE Trans. Automat. Control*, 52(6):1109–1112, 2007.

[Mis06] Marek Miskowicz. Send-on-delta concept: An event-based data reporting strategy. *Sensors*, 6(1):49–63, 2006.

[MLB+04] P. Martí, C. Lin, S. Brandt, M. Velasco, and J.M. Fuertes. Optimal state feedback based resource allocation for resource-constrained control tasks. *25th IEEE Real-Time Systems Symposium*, pages 161–172, 2004.

[MLB+09] Pau Martí, Caixue Lin, Scott A. Brandt, Manel Velasco, and Josep M.Fuertes. Draco: Efficient resource management for resource-constrained control tasks. *IEEE Transactions on Computers*, 58(1):90–105, January 2009.

[MLV+08] R. Marau, P. Leite, M. Velasco, P. Marti, L. Almeida, P. Pedreiras, and J.M. Fuertes. Performing flexible control on low-cost microcontrollers using a minimal real-time kernel. *Industrial Informatics, IEEE Transactions on*, 4(2):125 –133, may 2008.

[MT09] M. Mazo and P. Tabuada. Input-to-state stability of self-triggered control systems. In *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, 2009.

[MVB09] P. Martí, M. Velasco, , and E. Bini. The optimal boundary and regulator design problem for event-driven controllers. *12th International Conference on Hybrid Systems: Computation and Control*, April 2009.

[MVF+07] P. Martí, M. Velasco, J. M. Fuertes, R. Villà, J. Yépez, and C. Lozoya. The one-shot task model for implementing real-time control tasks. *II Congreso Español de Informática (CEDI2007)*, Sep. 2007.

[MVF+10] P Marti, M Velasco, J Fuertes, A Camcho, and G Buttazzo. Design of an embedded control systems laboratory experiment. *Industrial Electronics, IEEE Transactions on*, 2010.

[OSE] OSEK. Osek/vdx: Open systems and the corresponding interfaces for automotive electronics. `http://www.osek-vdx.org/mirror/os21.pdf`.

[OY98] S.H. Oh and S.M. Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. *5th International Conference Real-Time Computing Systems and Applications*, 1998.

[PLLA02] L. Palopoli, G. Lipari, G. Lamastra, and L. Abeni. An objectoriented tool for simulating distributed realtime control systems. *Software-Practice and Experience*, pages 907–932, 2002.

[PMRC07] S. Peiro, M. Masmano, I. Ripoll, and A. Crespo. PaRTiKle OS, a replacement for the core of RTLinux-GPL. *9th Real Time Linux Workshop*, 2007.

[PPBSV05] L. Palopoli, C. Pinello, A. Bicchi, and A. Sangiovanni-Vincentelli. Maximizing the stability radius of a set of systems under real-time scheduling constraints. *Automatic Control, IEEE Transactions on*, 50(11):1790–1795, Nov. 2005.

[PPSV+02] L. Palopoli, C. Pinello, A. L. Sangiovanni-Vincentelli, L. Elghaoui, and A. Bicchi. Synthesis of robust control systems under resource constraints. *Hybrid Systems: Computation and Control*, pages 337–350, 2002.

[PSCC08] R. Piza, J. Salt, A. Cuenca, and V. Casanova. Kalman filtering applied to profibus-dp systems. multirate control systems with delayed signals. *34th Annual Conference of IEEE Industrial Electronics Society*, Nov. 2008.

[REKT04] O. Redell, J. El-Khoury, and M. Torngren. The AIDA toolset for design and implementation analysis of distributed realtime control systems. *Journal of Microprocessors and Microsystems*, pages 163–182, 2004.

[RS00] H. Rehbinder and M. Sanfridson. Integration of off-line scheduling and optimal control. *12th Euromicro Conference on Real-Time Systems*, page 137, 2000.

[SÅ03] Ricardo Sanz and Karl-Erik Årzén. Trends in software and control. *IEEE Control Systems Magazine*, 23(3):12–15, June 2003.

[SAÅ+04] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real-time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2–3):101–155, November 2004.

[SB09] J. Skaf and S. Boyd. Analysis and synthesis of state-feedback controllers with timing jitter. *Automatic Control, IEEE Transactions on*, 54(3):652 –657, march 2009.

[SCEP09] Soheil Samii, Anton Cervin, Petru Eles, and Zebo Peng. Integrated scheduling and synthesis of control applications on distributed embedded systems. In *Proc. Design, Automation & Test in Europe (DATE'09)*, April 2009.

[SEPC09] Soheil Samii, Petru Eles, Zebo Peng, and Anton Cervin. Quality-driven synthesis of embedded multi-mode control systems. In *Proc. 46th Design Automation Conference (DAC)*, San Francisco, CA, July 2009.

[SKSC98] D. Seto, B. Krogh, L. Sha, and A. Chutinan. Dynamic control systems upgrade using Simplex architecture. *IEEE Control*, August 1998.

[SKSH06] P. Sucha, M. Kutil, M. Sojka, and Z. Hanzalek. TORSCHE scheduling toolbox for Matlab. *IEEE International Symposium on Computer-Aided Control Systems Design*, 2006.

[SL96] M.F. Storch and J.W.S. Liu. A simulation framework for complex realtime systems. *Proceedings of the 2nd IEEE RealTime Technology and Applications Symposium*, pages 160–169, 1996.

[SLS98] D. Seto, J. P. Lehoczky, and L. Sha. Task period selection and schedulability in real-time systems. *IEEE Real-Time Systems Symposium*, pages 188–198, Dec. 1998.

[SLSS96] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On task schedulability in real-time control systems. *IEEE Real-Time Systems Symposium*, 1996.

[SNR07] Y.S. Suh, V.H. Nguyena, , and Y. S. Roa. Modified Kalman filter for networked monitoring systems employing a send-on-delta. *Automatica*, 43:332–338, Feb. 2007.

[Srl08a] Evidence Srl. ERIKA Enterprise basic manual. `http://www.evidence.eu.com`, 2008.

[Srl08b] Evidence Srl. FLEX Modular solution for embedded applications. `http://www.evidence.eu.com`, 2008.

[Sta96] J. A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Comput. Surv.*, 28(4), dec 1996.

[Tab07] P. Tabuada. Event-triggered real-time scheduling of stabilizing control tasks. *Automatic Control, IEEE Transactions on*, 52(9):1680 –1685, sept. 2007.

[THÅ+06] Martin Törngren, Dan Henriksson, Karl-Erik Årzén, Anton Cervin, and Zdenek Hanzalek. Tools supporting the co-design of control systems and their real-time implementation; current status and future directions. In *2006 IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, Germany, October 2006.

[TW06] P. Tabuada and X. Wang. Preliminary results on state-triggered scheduling of stabilizing control tasks. *45th IEEE Conference on Decision and Control*, Dec. 2006.

[Vel06] M. Velasco. *Sistemas de Control con Recursos Restringidos*. PhD thesis, Technical University of Catalonia, Pau Gargallo 5, 08028 Barcelona, Spain, July 2006.

[VMB08] M. Velasco, P. Martí, and E. Bini. Control driven tasks: modeling and analysis. *29th IEEE Real-Time Systems Symposium (RTSS08)*, Dec. 2008.

[VMB09a] M. Velasco, P. Martí, and E. Bini. On lyapunov sampling for event-driven controllers. In *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, Dec. 2009.

[VMB09b] Manel Velasco, Pau Martí, and Enrico Bini. Equilibrium sampling interval sequences for event-driven controllers. In *In European Control Conference 2009*, Budapest, Hungary, 2009.

[VMF03] M. Velasco, P. Martí, and J.M. Fuertes. The self triggered task model for real-time control systems. *Work-in-progress session of the 24th IEEE Real-Time Systems Symposium*, Dec. 2003.

[VMF+10] M. Velasco, P. Martí, J. M. Fuertes, C. Lozoya, and S. Brandt. Experimental evaluation of slack management in real-time control systems: Coordinated vs. self-triggered approach. *Journal of Systems Architecture*, 56(1), January 2010.

[VML08] M. Velasco, P. Martí, and C. Lozoya. On the timing of discrete events in event-driven control systems. *11th International Conference on Hybrid Systems: Computation and Control (HSCC08)*, April 2008.

[WL08a] Xiaofeng Wang and Michael Lemmon. Event design in event-triggered feedback control systems. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, Dec. 2008.

[WL08b] Xiaofeng Wang and Michael Lemmon. State based self-triggered feedback control systems with l2 stability. In *17th IFAC World Congress*, jul. 2008.

[WL09a] X. Wang and M. Lemmon. Self-triggered feedback control systems with finite-gain l2 stability. *IEEE Transactions on Automatic Control*, 2009.

[WL09b] Xiaofeng Wang and Michael D. Lemmon. Self-triggered feedback systems with state-independent disturbances. In *ACC'09: Proceedings of the 2009 conference on American Control Conference*, pages 3842–3847, 2009.

[WNT95] B. Wittenmark, J. Nilsson, and M. Torngren. Timing problems in real-time control systems. *Proceedings of American Control Conference*, 1995.

[ZZ99] Q. C. Zhao and D. Z. Zheng. Stable and real-time scheduling of a class of hybrid dynamic systems. *Discrete Event Dynamic Systems*, 1999.

# Appendix A

# Continuous and discrete cost function

This appendix describes how to obtain a discrete-time cost function equivalent to the continuous standard quadratic cost function defined by

$$J_{control} = \int_0^{t_{eval}} \left[ x^T(t)Qx(t) + u^T(t)Ru(t) \right] dt, \tag{A.1}$$

now lets identify the continuous cost weighting matrices $Q$ and $R$ as $Q_c$ and $R_c$.

Consider that the plant to be controlled is described by the continuous-time model

$$\frac{dx(t)}{dt} = Ax(t) + Bu(t), \tag{A.2}$$

where $A$ and $B$ are the system and input matrices. If this model is sampled considering sampling periods of lengths $h$ then equation A.2 can be written as

$$x(t) = \Phi(t, kh)x(kh) + \Gamma(t, kh)u(kh), \tag{A.3}$$

where $\Phi$ and $\Gamma$ are obtained

$$\Phi = e^{Ah}, \quad \Gamma = \int_0^h e^{As} B ds, \tag{A.4}$$

The continuous cost function (A.1) is transformed into a discrete-time by integrating over intervals of lengths $h$,

$$J(k) = \int_{kh}^{kh+h} \left[ x^T(t)Q_c x(t) + u^T(t)R_c u(t) \right] dt. \tag{A.5}$$

Using (A.3) into (A.5) and the fact that $u(t)$ is constant over the sampling period gives

$$J(k) = x^T(kh)Q_d x(kh) + 2x^T(kh)N_d u(kh) + u^T(kh)R_d u(kh) \tag{A.6}$$

where

$$Q_d = \int_{kh}^{kh+h} \left[ \Phi^T(s, kh)Q_c \Phi(s, kh) \right] ds, \tag{A.7}$$

$$N_d = \int_{kh}^{kh+h} \left[\Phi^T(s,kh)Q_c\Gamma(s,kh)\right] ds, \tag{A.8}$$

$$R_d = \int_{kh}^{kh+h} \left[\Gamma^T(s,kh)Q_c\Gamma(s,kh) + R_c\right] ds. \tag{A.9}$$

Minimizing the continuous cost function (A.1) when $u(t)$ is constant over the sampling period is thus the same as minimizing the discrete-cost function,

$$J^d_{control} = \sum_{k=0}^{t_{eval}} \left[x^T(k)Q_d x(k) + 2x^T(k)N_d u(k) + u^T(k)R_d u(k)\right], \tag{A.10}$$

to facilitate the writing, it is assumed in the previous equation that the sampling period is used as time unit, that is, $h = 1$.

# Appendix B

# Framework simulation source code

This appendix presents partial source code for the main files that conforms the simulation part of the performance evaluation framework.

## B.1 Main program

### B.1.1 Function: main.m

```
Function:     main
Description:  Evaluate different FBS and DCS methods
```

```
%--------------------------------------------------------
% Validate models
%--------------------------------------------------------
validate_models;

%--------------------------------------------------------
% Load distrubance data
%--------------------------------------------------------
load_disturbance_data; amplitude_values = load(amplitude_file);
delay_values = load(delay_file);

%--------------------------------------------------------
% Create plant and
% execute off-line optimization if applies
%--------------------------------------------------------
switch plant_used
    case{'rc_circuit'}
        plant_RC;
    case{'ball_beam'}
        plant_BB;
    case{'double_integrator'}
        plant_DI;
    otherwise
        disp('Select a valid process plant !!!!');
        pause;
end

%--------------------------------------------------------
% Display initial values
%--------------------------------------------------------
if debug_mode == 1
    disp(sprintf('TASK MODEL SELECTED: %s',task_model));
    disp(sprintf('SCHEDULING APPROACH: %s',fbs_approach));
    disp(sprintf('INITIAL PERIODS: [%8.6f %8.6f %8.6f]',h(1),h(2),h(3)));
    pause;
end
```

```
for i_exec=1:1:length(amplitude_values)

        %------------------------------------------------------------------
        % Select disturbance amplitud and delay
        %------------------------------------------------------------------
        disturbance_amplitude=amplitude_values(i_exec,:);
        disturbance_delay=delay_values(i_exec,:);
        cpu_entries=0;
        switch plant_used
            case{'rc_circuit'}
                disturbance_track=0;
            case{'ball_beam'}
                disturbance_track=1;
            case{'double_integrtaor'}
                disturbance_track=1;
            otherwise
                disp('Select a valid process plant !!!!');
                pause;
        end
        if disturbance_type == 1
            task_offset = disturbance_delay;
        end

        %------------------------------------------------------------------
        % Execute Simulation
        %------------------------------------------------------------------
        sim('fbs_system',[init_time end_time]);
        sim_results=[sim_results, TotalCost(length(TotalCost))];
        cpu_results=[cpu_results, CPUTime(length(CPUTime))];
        cpu_total_entries=[cpu_total_entries, cpu_entries];
        if debug_mode == 1
            disp(sprintf('Cost Function'));
            disp(sprintf('%8.6f',mean(sim_results)));
            disp(sprintf('CPU Load'));
            disp(sprintf('%8.6f',mean(cpu_results)));
            pause;
        end
end

%------------------------------------------------------------------
% Display final results
%------------------------------------------------------------------
disp(sprintf('-------------------------------'));
disp(sprintf('Cost Function')); disp(sprintf('Mean Value'));
disp(sprintf('%8.6f',mean(sim_results)));

disp(sprintf('CPU Load')); disp(sprintf('Mean Value'));
disp(sprintf('%6.2f',mean(cpu_results)));
```

## B.2 Initialization modules

### B.2.1 Function: task_controller_init.m

```
Function:    task_controller_init
Description: Initializes the Simulink task controller module
```

```
function task_controller_init

% Initialize TrueTime kernel
ttInitKernel(6, 3, tc_sched_dfn);

% Create Tasks
switch task_model
    case {'fbs'}
        ttCreatePeriodicTask('task1',0.0,h(1),1,'fbs_controller_code',data1);
        ttCreatePeriodicTask('task2',0.0,h(2),1,'fbs_controller_code',data2);
        ttCreatePeriodicTask('task3',0.0,h(3),1,'fbs_controller_code',data3);
```

```
    case {'event'}
        ttCreatePeriodicTask('task1',0.0,h(1),1,'event_controller_code',data1);
        ttCreatePeriodicTask('task2',0.0,h(2),1,'event_controller_code',data2);
        ttCreatePeriodicTask('task3',0.0,h(3),1,'event_controller_code',data3);
    otherwise
        disp('Select a valid task model !!!!');
        pause;
end
```

### B.2.2 Function: resource_manager_init.m

```
Function:    task_controller_init
Description: Initializes the Simulink resource management module
```

```
function resource_manager_init

% Create Tasks
switch optimization_approach
    case {'static','seto_edf'}
        ttCreatePeriodicTask('task0_int',0.0,null_Tfbs,1,'period_allocate',data0);
    case {'marti_optimal'}
        ttCreatePeriodicTask('task0_int',0.0,marti_Tfbs,1,'period_allocate',data0);
    case {'henrikson_finite'}
        ttCreatePeriodicTask('task0_int',0.0,henr_Tfbs,1,'period_allocate',data0);
    case {'edc_velasco','edc_lemmon','edc_marti'}
        ttCreatePeriodicTask('task0_int',0.0, null_Tfbs,1,'period_allocate',data0);
    otherwise
        disp('Select a valid scheduling approach !!!!');
        pause;
end
```

## B.3 Controllers and optimization algorithms

### B.3.1 Function: fbs_controller_code.m

```
Function:    fbs_controller_code
Description: Control task for FBS approaches
```

```
function [exectime, data] = naif_controller_code(seg, data)

switch seg,
    case 1,
        y1 = ttAnalogIn(data.y1Chan); % Read process output
        y2 = ttAnalogIn(data.y2Chan); % Read process output
        data.u=-Kd(data.task_number,:)*[y1;y2;data.y3];
        data.y3=data.u;
        exectime = data.exec_time;
    case 2,
        ttAnalogOut(data.uChan, data.u); % Output control signal
        ttSetPeriod(h(data.task_number),data.task_name);
        cpu_entries=cpu_entries+1;
        exectime = -1;
end
```

### B.3.2 File: event_controller_code.m

```
Function:    event_controller_code
Description: Control task for EDC approaches
```

```
function [exectime, data] = event_controller_code(seg, data)

switch seg,
    case 1,
        y1 = ttAnalogIn(data.y1Chan); % Read process output
        y2 = ttAnalogIn(data.y2Chan); % Read process output
        data.u=-Kd(data.task_number,:)*[y1;y2];
        h(data.task_number)=next_event(-Kd(data.task_number,:),event_parameters,[y1;
            y2]);
        exectime = data.exec_time;
    case 2,
        ttAnalogOut(data.uChan, data.u); % Output control signal
        ttSetPeriod(h(data.task_number),data.task_name);
        cpu_entries=cpu_entries+1;
        exectime = -1;
end
```

### B.3.3 Function: `period_allocate.m`

```
Function:     period_allocate
Description: Executed to call on-line optimization routines
```

```
function [exectime, data] = period_allocate(seg, data)

switch seg,
 case 1,
    switch fbs_approach
        case {'static','seto_edf'}
            data=null_allocate(data);
        case {'edc_velasco','edc_lemmon','edc_marti'}
            data=event_allocate(data);
        case {'marti_optimal'}
            data=optimal_allocate(data);
        case {'henrikson_finite'}
            data=finite_horizon_allocate(data);
        otherwise
            disp('Select a valid scheduling approach !!!!');
            pause;
    end
    exectime = data.exec_time;

case 2,
    exectime = -1;
end
```

# Appendix C

# Framework experiment source code

This appendix presents partial source code for the main files that conforms the experimental part of the performance evaluation framework. The code include functions provided by the Erika kernel.

## C.1 File: setup.c

### C.1.1 Function: EE_Flex_setup()

```
Function:     EE_Flex_setup()
Description:  Configures system clock and initialize devices
```

```
void EE_Flex_setup(void) {
    // Configure the PWM 1
    PWM_init();

    // Configure the orange led of the FLEX FULL and custom leds
    Led_init();

    // Configure digital pins to be used with the oscilloscope
    Digital_output_init();

    // Initialize the UART Port 1 to communicate with the PC via RS232
    UART1_DMA_init();

    // Initialize the Analog to Digital Converter 1
    ADC1_init();
}
```

### C.1.2 Function: PWM_init()

```
Function:     PWM_config()
Description:  Configures PWM actuator
```

```
void PWM_init(void) {
    OVDCON = 0x0000;       // PWM outputs disabled
    PTCONbits.PTEN = 1;    //PTEN: PWM Time Base Timer Enable bit
                           //1 = PWM time base is on
                           //0 = PWM time base is off
    PTCONbits.PTMOD=2;     //PTMOD<1:0>: PWM Time Base Mode Select bits
```

```
                        //11 =PWM time base operates in a Continuous Up/Down
                        //Count mode with interrupts for double PWM updates
                        //10 =PWM time base operates in a Continuous Up/Down
                        //01 =PWM time base operates in Single Pulse mode
                        //00 =PWM time base operates in a Free-Running mode
    PTPER = 0x3FFF;     //PTPER<14:0>: PWM Time Base Period Value bits
                        //Select PWM period: Setting PDCx=0 means 0% Duty cycle
                        //                           PDCx=PTPER means 50%
                        //                           PDCx=2*PTPER means 100%
    PWMCON1bits.PMOD1=1;//PWM I/O Pair Mode bits
                        //1 = PWM I/O pin pair is in the Independent PWM Output mode
                        //0 = PWM I/O pin pair is in the Complementary Output mode
    PWMCON1bits.PMOD2=1;
    PWMCON1bits.PMOD3=1;
    PWMCON1bits.PMOD4=1;
    PWMCON1bits.PEN4H=1;//PWMxH I/O Enable bits
                        //1 = PWMxH pin is enabled for PWM output
                        //0 = PWMxH pin disabled, I/O pin becomes general purpose I/O
    PWMCON1bits.PEN3H=1;
    PWMCON1bits.PEN2H=1;
    PWMCON1bits.PEN1H=1;
    PWMCON1bits.PEN4L=1;//PWMxL I/O Enable bits
                        //1 = PWMxL pin is enabled for PWM output
                        //0 = PWMxL pin disabled, I/O pin becomes general purpose I/O
    PWMCON1bits.PEN3L=1;
    PWMCON1bits.PEN2L=1;
    PWMCON1bits.PEN1L=1;
    PWMCON2bits.IUE = 0;//Immediate Update Enable bit
                        //1 = Updates to the active PDC registers are immediate
                        //0 = Updates to the active PDC registers are synchronized
                        //to the PWM time base
    PWMCON2bits.UDIS= 0;//PWM Update Disable bit
                        //1 = Updates from Duty Cycle and Period Buffer
                        // registers are disabled
                        //0 = Updates from Duty Cycle and Period Buffer
                        // registers are enabled
    OVDCON = 0xff00;    //OVERRIDE CONTROL REGISTER
                        //bit 15-8 POVDxH<4:1>:POVDxL<4:1>: PWM Output
                        //Override bits
                        //1 = Output on PWMx I/O pin is controlled by the
                        //PWM generator
                        //0 = Output on PWMx I/O pin is controlled by the
                        //value in the corresponding POUTxH:POUTxL bit
                        //bit 7-0 POUTxH<4:1>:POUTxL<4:1>: PWM Manual Output bits
                        //1 = PWMx I/O pin is driven active when the
                        //corresponding POVDxH:POVDxL bit is cleared
                        //0 = PWMx I/O pin is driven inactive when the
                        //corresponding POVDxH:POVDxL bit is cleared*/
    PDC1 = 0x0000;      //Initial duty cycle PWM1
    PDC2 = 0x0000;      //Initial duty cycle PWM2
    PDC3 = 0x0000;      //Initial duty cycle PWM3
    PDC4 = 0x0000;      //Initial duty cycle PWM4
    PTCONbits.PTEN = 1; // Enable PWM.
}
```

### C.1.3 Function: `Led_init()`

```
Function:     Led_init()
Description:  Configures FLEX FULL orange led (Jumper 4 must be closed)
```

```
void Led_init(void) {
    LATBbits.LATB14 = 0;    //set orange LED (LEDSYS/RB14) drive state low
    TRISBbits.TRISB14 = 0;  //set LED pin (LEDSYS/RB14) as output

    LATDbits.LATD8=0;       //set pin(IC1/RD8)->(CON5/Pin7) drive state low
    TRISDbits.TRISD8=0;     //set pin(IC1/RD8)->(CON5/Pin7)as output

    LATDbits.LATD9=0;       //set pin(IC2/RD9)->(CON5/Pin10) drive state low
    TRISDbits.TRISD9=0;     //set pin(IC2/RD9)->(CON5/Pin10)as output
```

```
        LATDbits.LATD10=0;          //set pin(IC3/RD10)->(CON5/Pin9) drive state low
        TRISDbits.TRISD10=0;        //set pin(IC3/RD10)->(CON5/Pin9)as output

        LATDbits.LATD11=0;          //set pin(IC4/RD11)->(CON5/Pin12) drive state low
        TRISDbits.TRISD11=0;        //set pin(IC4/RD11)->(CON5/Pin12)as output

        LATDbits.LATD12=0;          //set pin(IC5/RD12)->(CON5/Pin15) drive state low
        TRISDbits.TRISD12=0;        //set pin(IC5/RD12)->(CON5/Pin15)as output

        LATDbits.LATD13=0;          //set pin(IC6/CN19/RD13)->(CON5/Pin18) drive state low
        TRISDbits.TRISD13=0;        //set pin(IC6/CN19/RD13)->(CON5/Pin18)as output
    }
```

### C.1.4 Function: Digital_output_init()

```
    Function:      Digital_output_init()
    Description: Configures pin (AN10/RB10)-->(CON6/Pin28) from the
           FLEX FULL to get execution times with oscilloscope
```

```
    void Digital_output_init(void) {
        // set pin (AN10/RB10)-->(CON6/Pin28) drive state low
        LATBbits.LATB10 = 0;
        // set pin (AN10/RB10)-->(CON6/Pin28) as output
        TRISBbits.TRISB10 = 0;

    }
```

### C.1.5 Function: UART1_DMA_init()

```
    Function:      UART1_DMA_init()
    Description: Initialize the UART Port 1 to communicate with the
       PC via RS232
```

```
    void UART1_DMA_init() {
        cfgDma4UartTx();// This routine Configures DMAchannel 4 for transmission.
        cfgDma5UartRx();// This routine Configures DMAchannel 5 for reception.

        U1MODEbits.STSEL = 0;// 1-stop bit
        U1MODEbits.PDSEL = 0;// No Parity, 8-data bits
        U1MODEbits.ABAUD = 0;// Autobaud Disabled
        U1MODEbits.BRGH=1;// 1 = BRG generates 4 clocks per bit period (4x baud clock,
                          // High-Speed mode)
                          // 0 = BRG generates 16 clocks per bit period (16x baud clock,
                          // Standard mode)
        U1BRG = BRGVAL;   // See #ifdef BITRATE1 above for details
        // Configure UART for DMA transfers
        U1STAbits.UTXISEL0 = 1;// UTXISEL<1:0>: Transmission Interrupt Mode Selection
            bits
                               // 11 =Reserved; do not use
                               // 10 =Interrupt when a character is transferred to the
                               // Transmit Shift Register, and as a result, the transmit
                               // buffer becomes empty
                               // 01 =Interrupt when the last character is shifted out
                               // of the Transmit Shift Register; all transmit
                                   operations
                               // are completed
                               // 00 =Interrupt when a character is transferred to the
                               // Transmit Shift Register (this implies there is at least
                               // one character open in the transmit buffer)*/
        U1STAbits.URXISEL  = 1;// 11 =Interrupt is set on UxRSR transfer making the
                               // receive buffer full (i.e., has 4 data characters)
                               // 10 =Interrupt is set on UxRSR transfer making the
                               // receive buffer 3/4 full (i.e., has 3 data characters)
                               // 0x =Interrupt is set when any character is received
                               // and transferred from the UxRSR to the receive buffer.
```

```
                              // Receive buffer has one or more characters.*/
    // Enable UART Rx and Tx
    U1MODEbits.UARTEN   = 1;// Enable UART
    U1STAbits.UTXEN     = 1;// Enable UART Tx
    IEC4bits.U1EIE = 0; // UART1 Error Interrupt Enable bit
                        // 1 = Interrupt request has occurred
                        // 0 = Interrupt request has not occurred*/
}
```

### C.1.6 Function: `ADC1_init()`

```
Function:    ADC1_init()
Description: Configures ADC1
```

```
void ADC1_init(void) {

    AD1CON1bits.ADON  = 0;// ADC Operating Mode bit. Turn off the A/D converter
    AD1PCFGL = 0xFFFF;      //ADC1 Port Configuration Register Low
    AD1PCFGH = 0xFFFF;      //ADC1 Port Configuration Register High
    AD1PCFGLbits.PCFG11=0; //Plant B (Double integrator B), x1
    AD1PCFGLbits.PCFG12=0; //Plant B (Double integrator B), x2
    AD1PCFGLbits.PCFG13=0; //Plant A (Double integrator A), x1
    AD1PCFGLbits.PCFG15=0; //Plant A (Double integrator A), x2
    AD1PCFGHbits.PCFG17=0; //Plant D (RC-RC D), x2
    AD1PCFGHbits.PCFG18=0; //Plant D (RC-RC D), x1
    AD1PCFGHbits.PCFG20=0; //Plant C (Double integrator C), x1
    AD1PCFGHbits.PCFG21=0; //Plant C (Double integrator C), x2

    AD1CON2bits.VCFG = 0;   //Converter Voltage Reference Configuration bits
                            // (ADRef+=AVdd, ADRef-=AVss)
    AD1CON3bits.ADCS = 63;  // ADC Conversion Clock Select bits
                            //(Tad = Tcy*(ADCS+1) = (1/40000000)*64 = 1.6us)
                            //Tcy=Instruction Cycle Time=40MIPS */
    AD1CON2bits.CHPS = 0;    // Selects Channels Utilized bits, When AD12B = 1,
                             // CHPS<1:0> is: U-0, Unimplemented, Read as 0
    AD1CON1bits.SSRC = 7;//Sample Clock Source Select bits:
                        // 111 = Internal counter ends sampling and starts
                        // conversion (auto-convert)
                        // 110 = Reserved
                        // 101 = Reserved
                        // 100 = Reserved
                        // 011 = MPWM interval ends sampling and starts
                        // conversion
                        // 010 = GP timer (Timer3 for ADC1, Timer5 for ADC2)
                        // compare ends sampling and starts conversion
                        // 001 = Active transition on INTx pin ends sampling
                        // and starts conversion
                        // 000 = Clearing sample bit ends sampling and starts
                        // conversion
    AD1CON3bits.SAMC = 31;   // Auto Sample Time bits. (31*Tad = 49.6us)
    AD1CON1bits.FORM = 0;    // Data Output Format bits. Integer
                            // For 12-bit operation:
                            //    11 = Signed fractional
                            //    (DOUT = sddd dddd dddd 0000, where s = .NOT.d<11>)
                            //    10 = Fractional
                            //    (DOUT = dddd dddd dddd 0000)
                            //    01 = Signed Integer
                            //    (DOUT = ssss sddd dddd dddd, where s = .NOT.d<11>)
                            //    00 = Integer
                            //    (DOUT = 0000 dddd dddd dddd)
    AD1CON1bits.AD12B = 1;   // Operation Mode bit:
                             //   0 = 10 bit
                             //   1 = 12 bit
    AD1CON1bits.ASAM  = 0;   // ADC Sample Auto-Start bit:
                             //   1 = Sampling begins immediately after last
                             //   conversion. SAMP bit is auto-set.
                             //   0 = Sampling begins when SAMP bit is set
    AD1CHS0bits.CH0NA = 0;   // MUXA -Ve input selection (Vref-) for CH0.
    AD1CON1bits.ADON  = 1;   // ADC Operating Mode bit. Turn on A/D converter
}
```

# C.2 File: config.oil

## C.2.1 Function: CPU Configuration

```
Configuration: CPU specification
```

```
CPU mySystem {
    OS myOs {
        EE_OPT = "DEBUG";
        CPU_DATA = PIC30 {
            APP_SRC = "code.c";
            MULTI_STACK = FALSE;
            ICD2 = TRUE;
        };
        MCU_DATA = PIC30 {
            MODEL = PIC33FJ256MC710;
        };
        BOARD_DATA = EE_FLEX {
            USELEDS = TRUE;
        };
        KERNEL_TYPE = EDF { NESTED_IRQ = TRUE; TICK_TIME = "25ns";};
    };
```

## C.2.2 Function: Task definitions

```
Configuration: Task definitions
```

```
   TASK TaskReferenceChangeA {
        REL_DEADLINE = "0.005s";
        PRIORITY = 4;
        STACK = SHARED;
        SCHEDULE = FULL;
    };
    TASK TaskPeriodicControllerA {
        REL_DEADLINE = "0.05s";
        PRIORITY = 2;
        STACK = SHARED;
        SCHEDULE = FULL;
    };
    TASK TaskEventControllerA {
        REL_DEADLINE = "0.05s";
        PRIORITY = 2;
        STACK = SHARED;
        SCHEDULE = FULL;
    };
    TASK TaskOnlineOptimization {
        REL_DEADLINE = "0.1s";
        PRIORITY = 6;
        STACK = SHARED;
        SCHEDULE = FULL;
    };
    TASK TaskSend {
        REL_DEADLINE = "0.1s";
        PRIORITY = 1;
        STACK = SHARED;
        SCHEDULE = FULL;
    };
```

## C.2.3 Function: Alarm definitions

```
Configuration: Alarm definitions
```

```
ALARM AlarmReferenceChangeA {
    COUNTER = "myCounter";
    ACTION = ACTIVATETASK { TASK = "TaskReferenceChangeA"; };
};
ALARM AlarmPeriodicControllerA {
    COUNTER = "myCounter";
    ACTION = ACTIVATETASK { TASK = "TaskPeriodicControllerA"; };
};
ALARM AlarmEventControllerA {
    COUNTER = "myCounter";
    ACTION = ACTIVATETASK { TASK = "TaskEventControllerA"; };
};
ALARM AlarmSend {
    COUNTER = "myCounter";
    ACTION = ACTIVATETASK { TASK = "TaskSend"; };
};
ALARM AlarmOnlineOptimization {
    COUNTER = "myCounter";
    ACTION = ACTIVATETASK { TASK = "TaskOnlineOptimization"; };
};
```

## C.3 File: `code.c`

### C.3.1 Function: `main()`

```
Function:    main()
Description: main function, only to initialize software and hardware,
       fire alarms, and implement background activities
```

```
int main(void) {
    // Clock setup for 40MIPS
    CLKDIVbits.DOZEN   = 0;
    CLKDIVbits.PLLPRE  = 0;
    CLKDIVbits.PLLPOST = 0;
    PLLFBDbits.PLLDIV  = 78;

    T1_program();    // Program Timer 1 to raise interrupts
    EE_time_init(); //EDF time init
    EE_Flex_setup();//Initialize clock and devices

    // Program cyclic alarms
    SetRelAlarm(AlarmReferenceChangeA, 1500, 3000);//Reference change Task
    SetRelAlarm(AlarmReferenceChangeB, 2000, 3000);
    SetRelAlarm(AlarmReferenceChangeC, 2500, 3000);

    if( approach_type==EDC ){
        SetRelAlarm(AlarmEventControllerA, 1000, 0);//EDC Controller Tasks
        SetRelAlarm(AlarmEventControllerB, 1000, 0);
        SetRelAlarm(AlarmEventControllerC, 1000, 0);
    }
    else {
        SetRelAlarm(AlarmPeriodicControllerA, 1000, hA);//FBS Controller Task
        SetRelAlarm(AlarmPeriodicControllerB, 1000, hB);
        SetRelAlarm(AlarmPeriodicControllerC, 1000, hC);
        SetRelAlarm(AlarmOnlineOptimization, 1250, 500);//FBS optimization every 500
            ms
    }
    SetRelAlarm(AlarmSend, 1000, 5);    //Data is sent to the PC every 5ms

    // Forever loop: background activities (if any) should go here
    for (;;);

    return 0;
}
```

### C.3.2 Function: Read_StateA()

```
Function:     Read_StateA()
Description: Read Plant A integrators output voltages
```

```
void Read_StateA(void) {
    AD1CHS0 = 13;
    AD1CON1bits.SAMP = 1;   // Start conversion
    while(!IFS0bits.AD1IF);// Wait till the EOC
    IFS0bits.AD1IF = 0;     // reset ADC interrupt flag
    xA[0]=(ADC1BUF0/4096.0)*v_max-(v_max/2);//scale to relative voltage

    AD1CHS0 = 15;
    AD1CON1bits.SAMP = 1;
    while(!IFS0bits.AD1IF);
    IFS0bits.AD1IF = 0;
    xA[1]=(ADC1BUF0/4096.0)*v_max-(v_max/2);
}
```

### C.3.3 Function: TASK(TaskReferenceChangeA)

```
Function:    TASK(TaskReferenceChangeA)
Description: Changes Plant A reference value
```

```
TASK(TaskReferenceChangeA) {
    if (referenceA == -0.5)
    {
        referenceA=0.5;
        LATDbits.LATD8 = 1;
    }else{
        referenceA=-0.5;
        LATDbits.LATD8 = 0;
    }
    index_event_timeA=0;
}
```

### C.3.4 Function: TASK(TaskEventControllerA)

```
Function:    TASK(TaskEventControllerA)
Description: Event controller code for plant A
```

```
TASK(TaskControllerA) {

    //To avoid two different alarms at the same time
    CancelAlarm(AlarmEventControllerA);

    //Read plant current state
    rA=referenceA;
    Read_StateA();
    xA_hat[0]=xA[0]-rA*Nx_DI[0];
    xA_hat[1]=xA[1]-rA*Nx_DI[1];
    uA_ss=rA*Nu_DI;

    //Calculate control signal
    uA=-K1*xA_hat[0]-K2*xA_hat[1]+uA_ss;
    if (uA>v_max/2) uA=v_max/2;
    if (uA<-v_max/2) uA=-v_max/2;

    //PDC1 is the register witch sets the PWM duty cycle for the 1st DI
    PDC1=(uA/v_max)*32768+16384;

    //Estimate next event time
    event_timeA=Calculate_Next_Activation_Time('A');
```

```
    //Set next event time to fire alarm,
    SetRelAlarm(AlarmEventControllerA, event_timeA, 0);
}
```

### C.3.5 Function: TASK(TaskPeriodicControllerA)

```
Function:    TASK(TaskPeriodicControllerA)
Description: Periodic controller code for plant A
```

```
TASK(TaskPeriodicControllerA) {

    //Read plant current state
    rA=referenceA;
    Read_StateA();
    xA_hat[0]=xA[0]-rA*Nx_DI[0];
    xA_hat[1]=xA[1]-rA*Nx_DI[1];
    uA_ss=rA*Nu_DI;

    //Calculate control signal
    uA=-kA[0]*xA_hat[0]-kA[1]*xA_hat[1]+uA_ss;
    if (uA>v_max/2) uA=v_max/2;
    if (uA<-v_max/2) uA=-v_max/2;

    //PDC1 is the register witch sets the PWM duty cycle for the 1st DI
    PDC1=(uA/v_max)*32768+16384;
}
```

### C.3.6 Function: TASK(TaskOnlineOptimization)

```
Function:    TASK(TaskOnlineOptimization)
Description: On-line optimization function for resource allocation
```

```
TASK(TaskOnlineOptimization) {

    //Read plants current state
    Read_StateA();
    Read_StateB();
    Read_StateC();

    //Execute optimization
    new_allocation_required=Optimization_process();
    if ( new_allocation_required ){
        CancelControllerAlarms();
        SetControllerAlarms();
        ActivateControllerAlarms();
    }
}
```

### C.3.7 Function: TASK(TaskSend)

```
Function:    TASK(TaskSend)
Description: Send data using the UART port 1 via RS232 to the PC
```

```
TASK(TaskSend) {

    //Read plants current state
    Read_StateA();
    Read_StateB();
    Read_StateC();
```

```
        //Send data
        Send_data_to_PC();
        DMA4CONbits.CHEN  = 1;        // Re-enable DMA4 Channel
        DMA4REQbits.FORCE = 1;        // Manual mode: Kick-start the first transfer
        }
}
```