Integrated Control and Real-Time Scheduling

Integrated Control and Real-Time Scheduling

Anton Cervin

Department of Automatic Control
Lund Institute of Technology
Lund, April 2003

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 LUND
Sweden

ISSN 0280–5316
ISRN LUTFD2/TFRT--1065--SE

© 2003 by Anton Cervin. All rights reserved.
Printed in Sweden by Bloms i Lund Tryckeri AB.
Lund 2003

Abstract

The topic of the thesis is codesign of flexible real-time control systems. Integrating control theory and real-time scheduling theory, it is possible to achieve higher resource utilization and better control performance. The integration requires new tools for analysis, design, and implementation.

The problem of scheduling the individual parts of a control algorithm is studied. It is shown how subtask scheduling can reduce the input-output latency in a set of control tasks. Deadline assignment under different scheduling policies is considered.

A feedback scheduling architecture for control tasks is introduced. The scheduler uses feedback from execution-time measurements and feedforward from workload changes to adjust the sampling periods of a set of control tasks so that the combined performance of the controllers is optimized.

The Control Server, a novel computational model for real-time control tasks, is presented. The model combines time-triggered I/O with dynamic, reservation-based task scheduling. The model provides short input-output latencies and minimal jitter for the controllers. It also allows control tasks to be treated as scalable real-time components with predictable performance.

Two MATLAB-based toolboxes for analysis and simulation of real-time control systems have been developed. The Jitterbug toolbox evaluates a quadratic cost function for a linear control system with timing variations. The tool makes it possible to investigate the impact of delay, jitter, lost samples, etc., on control performance. The TrueTime toolbox facilitates detailed cosimulation of distributed real-time control systems. The scheduling and execution of control tasks is simulated in parallel with the network communication and the continuous process dynamics.

Acknowledgments

First, I would like to thank my supervisor Karl-Erik Årzén. He, together with Klas Nilsson and Ola Dahl, wrote the original proposal for the research project “Integrated Control and Scheduling”. Never short on good ideas, Karl-Erik has been an excellent advisor since the day I started my graduate studies. He has also been a constant supplier of good music over the years.

This thesis would not have turned out half as good without the help from several of my fellow PhD students and colleagues. Johan Eker is the co-author of a staggering 50% of my publications. Together, we have worked on the simulator, feedback scheduling, and, most recently, the Control Server. Dan Henriksson has been the main implementer of the new version of the simulator, called TrueTime. Bo Lincoln has implemented the Jitterbug analysis toolbox. Thank you all for the great work!

It has been wonderful to work at the Department of Automatic Control in Lund, where the people are always friendly and helpful. The professors, the secretaries, and the technical staff keep the department running very smoothly. I would especially like to thank the founder of the department, Karl Johan Åström, who lured me into the field of automatic control and encouraged me to become a PhD student. Also, I would like to thank my co-supervisors Bo Bernhardsson and Per Hagander.

During my studies, I have had the opportunity to visit colleagues abroad. I would like to thank Professor Lui Sha at the Department of Computer Science, University of Illinois at Urbana-Champaign for much inspiration and visits on two separate occasions. I would also like to thank Professor Edward Lee at the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley for a research visit in 2001.

This research project has been a collaboration between the Department of Automatic Control and the Department of Computer Science at Lund Institute of Technology. It has been a pleasure to work together with Klas

Nilsson, Patrik Persson, and Sven Gestegård Robertz.

The work in this thesis has been supported by ARTES, a real-time research network in Sweden funded by SSF. I have enjoyed visits to several summer schools, graduate student conferences, and other activities organized by ARTES over the past five years.

ARTES has provided funding for several conference trips and research visits. A travel grant from the Royal Physiographic Society in Lund is also gratefully acknowledged.

Finally, I would like thank my family, friends, fellow lindy hoppers, poker buddies, etc., for making my life so enjoyable! A special thank you goes out to Matilda for her love and inspiration.

Anton

Contents

1. Introduction 11
- 1.1 Motivation 11
- 1.2 The Codesign Problem 12
- 1.3 Goals and Contributions 14
- 1.4 An Introductory Example 17
- 1.5 Outline and Publications 29

2. Background 33
- 2.1 Introduction 33
- 2.2 Real-Time Scheduling Theory 35
- 2.3 Control Loop Timing 41
- 2.4 Control and Scheduling Codesign 50

3. Subtask Scheduling 52
- 3.1 Introduction 52
- 3.2 Task Models 55
- 3.3 Deadline Assignment 56
- 3.4 Latency Analysis 59
- 3.5 Implementation 61
- 3.6 An Example 62
- 3.7 Conclusion 66

4. Feedback Scheduling 67
- 4.1 Introduction 67
- 4.2 A Feedback Scheduling Architecture 70
- 4.3 Case Study 1: Hybrid Controllers 72
- 4.4 Optimal Resource Distribution 81
- 4.5 Case Study 2: Linear Controllers 85
- 4.6 EDF as a Feedback Scheduling Mechanism? 94
- 4.7 Conclusion 95

5. The Control Server . . . 96
5.1 Introduction . . . 96
5.2 The Model . . . 98
5.3 Control and Scheduling Codesign . . . 101
5.4 CS Tasks as Real-Time Components . . . 104
5.5 Feedback Scheduling . . . 107
5.6 Implementation . . . 109
5.7 Control Experiments . . . 112
5.8 Conclusion . . . 114

6. Analysis Using Jitterbug . . . 116
6.1 Introduction . . . 116
6.2 System Description . . . 117
6.3 Internal Workings . . . 123
6.4 Examples . . . 126
6.5 Conclusion . . . 138

7. Simulation Using TrueTime . . . 139
7.1 Introduction . . . 139
7.2 Overview . . . 141
7.3 The Kernel Block . . . 142
7.4 The Network Block . . . 149
7.5 Examples . . . 153
7.6 Conclusion . . . 161

8. Conclusion . . . 164
8.1 Summary . . . 164
8.2 Suggestions for Future Work . . . 166

Appendix . . . 168
Cost Calculation in Example 3.1 . . . 168
Proof of Theorem 4.2 . . . 170

References . . . 172

1

Introduction

1.1 Motivation

Real-time control plays an important part in modern technology. For example, a CD or DVD player could never operate without its feedback control system. Engine management systems in modern cars rely heavily on real-time computations and feedback control to improve performance, reduce fuel consumption, and minimize the amount of pollutant emissions. As the capacity of microcontrollers is increasing and the cost is decreasing, more and more functionality is realized in software. In an embedded control system, a control task is typically executing in parallel with several other tasks, including other control tasks. This puts focus on *scheduling*, i.e., the choice of which task to execute at a given time. Since the beginning of the 1970s, the academic interest in real-time scheduling has been very large. Very little of this work has, however, focused on control tasks. On the other hand, digital control theory, with its origin in the 1950s, does not address the problem of shared and limited resources in the computing system. Instead, it is commonly assumed that the controller executes as a simple loop in a dedicated computer.

Real-time scheduling is sometimes dismissed as a non-problem. With ever more powerful computers, it can be argued that most timing problems can be solved by upgrading the CPU to a later and faster model. While this might be true in some cases, developers of embedded systems will testify that they are always struggling to add yet another function to an already heavily loaded processor. To keep production costs down, manufacturers of consumer products tend, of course, to use the most inexpensive hardware possible. Only in extreme applications, such as nuclear power plants, can the cost of the computing hardware be neglected in the overall development costs.

This work aims at achieving the best possible control performance from limited computing resources. To accomplish this goal, integration of the control design and the real-time scheduling design is necessary. Today, the design of a real-time control system is typically a two-step procedure: control design followed by real-time design. Moreover, the steps are often carried out in relative isolation and by engineers with different backgrounds. The control engineer designs and evaluates the control algorithms assuming a very simple model of the computing platform. The computer scientist schedules the controllers together with other tasks and makes design trade-offs without really knowing the controller timing requirements. The isolated development introduces conservatism and leads to non-optimal solutions.

There is a strong trend towards flexibility in real-time control. In the past, developers have relied on static analysis and design, knowing that the controllers would execute on deterministic hardware and in a predictable environment. Today, both hardware and operating systems tend to be commercial-off-the-shelf (COTS) products, sometimes poorly specified, and typically optimized for high average-case performance rather than predictable worst-case performance. Controllers are more frequently being treated as software components, expected to work in different configurations and being subject to on-line upgrades. Furthermore, modern control systems are often distributed systems, where sensors, controllers, and actuators are located in different nodes in a network. A distributed system is more flexible in nature but also more nondeterministic. Communication protocols may introduce timing variations that influence both the control performance and the task scheduling in the various computer nodes.

1.2 The Codesign Problem

Successful development of a real-time control system requires *codesign* of the computer system and the control system. The computing platform must be dimensioned such that all functionality can be accommodated, and the controllers must be designed taking the hardware limitations into account. The computer system has many important aspects (numerics, memory, I/O, network, power consumption, etc.). This thesis focuses on the scheduling of control tasks in the CPU.

The *control and scheduling codesign problem* can be informally stated as follows:

> Given a set of processes to be controlled and a computer with limited computational resources, design a set of controllers and

schedule them as real-time tasks such that the overall control performance is optimized.

An alternative view of the same problem is to say that we should design and schedule a set of controllers such that the least expensive implementation platform can be used while still meeting the performance specifications.

The nature and the degree of difficulty of the codesign problem for a given system depend on a number of factors:

- *The real-time operating system.* What scheduling algorithms are supported? How is I/O handled? Can the real-time kernel measure task execution times and detect execution overruns and missed deadlines?
- *The scheduling algorithm.* Is is time-driven or event-driven, priority-driven or deadline-driven? What analytical results regarding schedulability and response times are available? How are task overruns handled? What scheduling parameters can be changed on-line?
- *The controller synthesis method.* What design criteria are used? Are the controllers designed in the continuous-time domain and then discretized or is direct discrete design used? Are the controllers designed to be robust against timing variations? Should they actively compensate for timing variations?
- *The execution-time characteristics of the control algorithms.* Do the algorithms have predictable worst-case execution times? Are there large variations in execution time from sample to sample? Do the controllers switch between different internal modes with different execution-time profiles?
- *Off-line or on-line optimization.* What information is available for the off-line design and how accurate is it? What can be measured on-line? Should the system be able to handle the arrival of new tasks? Should the system be re-optimized when the workload changes? Should there be feedback from the control performance to the scheduling algorithm?

The problems studied in this thesis will assume a real-time operating system that supports dynamic scheduling algorithms (fixed-priority or earliest-deadline-first scheduling). We will mainly consider scheduling of linear controllers whose performance is evaluated by quadratic cost functions. When on-line optimization is introduced, it will be assumed that the execution times of the tasks can be measured, and that the scheduling parameters can be changed on-line.

1.3 Goals and Contributions

Control theory and real-time systems theory have evolved as separate fields during the last couple of decades. There is clearly a lack of common understanding between the two research communities and no well-defined design interface. In the control community, the real-time system is viewed as a platform on which the controller can be trivially implemented. In the real-time systems community, a controller is viewed as a piece of code characterized by three fixed parameters: a period, a computation time, and a deadline. This thesis aims at bridging the gap between the two research areas. For this purpose, new tools and techniques for analysis, design, and implementation are needed. The goals and contributions of the thesis are outlined below.

New Analysis Tools

There is a need for better understanding of what happens when a controller is implemented and scheduled as a real-time task. For this purpose, two MATLAB-based analysis tools have been developed: TrueTime[1] and Jitterbug[2]. The tools can be used at early design stages to determine how sensitive controllers are to scheduling-induced delays and jitter. They can also be used at the implementation stage for trade-off analysis between the tasks.

TrueTime, which is based on MATLAB/Simulink, is used for detailed co-simulation of the computer system and the control system. In the tool, time is added as a new dimension to the control algorithms. The controllers are executed in a real-time operating system, which is simulated in parallel with the continuous-time plant dynamics. An arbitrary scheduling algorithm can be used, and the code can be simulated on a time-scale of choice. The tool is very general and can be used to investigate, for instance, the influence of various scheduling algorithms on the control performance. Although it falls outside the main scope of this thesis, TrueTime can also be used to simulate distributed control systems and evaluate different communication protocols from a control perspective.

Simulation is a very useful tool for control systems design but there are limitations. The user may lack exact knowledge of task execution in the target system. Also, very long simulation times may be needed to draw conclusions about performance and stability. As an alternative, Jitterbug can be used for analysis of simple models of real-time control systems. In the tool, the timing variations introduced by the real-time operating system are modeled by statistical delay distributions. Given a

[1]Available at http://www.control.lth.se/˜dan/truetime/
[2]Available at http://www.control.lth.se/˜ lincoln/jitterbug/

linear control system and a timing model, the tool analytically computes a quadratic performance index, or *cost function*. By evaluating the cost function for a wide range of timing parameters, the designer can investigate how sensitive the control loop is to timing variations. Jitterbug also supports frequency-domain analysis in the form of spectral density calculations.

More Detailed Scheduling Analysis

For a given system, the analysis tools above may indicate that controller timing is a critical issue. The implementation may introduce latency and jitter that deteriorate the performance and force the developer to use a more expensive CPU. It can then be valuable to perform a more detailed scheduling analysis in order to reduce the latencies. The remaining delay can be compensated for using control theory.

One way to reduce the input-output latency in a controller is to split the control algorithm into two parts: Calculate Output and Update State. The Calculate Output part should only contain the operations necessary to produce a control signal, while the rest of the operations should be postponed to the Update State part. In the thesis, it is shown how the two parts can be scheduled as subtasks in order to reduce the input-output latency for a set of controllers. The control performance improvements come at the expense of a slightly more complex implementation.

Introduction of Feedback in the Computing System

Real-time systems are typically designed using static (off-line) analysis techniques, even if dynamic scheduling algorithms are used in the target system. To guarantee that all deadlines are met, the analysis is based on worst-case assumptions about task arrival rates and execution times. In systems where the workload changes over time, this approach may lead to low average resource utilization. By the introduction of feedback in the computing system, a less pessimistic analysis can be carried out on-line, allowing the computing resources to be better exploited. An illustration of a general feedback scheduling structure is given in Figure 1.1.

In the thesis, we study controllers whose computational demands vary over time. A traditional, static worst-case analysis would in these cases lead to low average-case CPU utilization and possibly poor control performance. A feedback mechanism is proposed that rescales the sampling periods of the controllers based on execution-time measurements. A feed-forward path is also introduced, such that the control tasks can inform the scheduler that they are about to consume more resources. The controllers may adjust their parameters according to the current timing conditions. The proposed mechanism could be implemented in a standard operating system, provided that it supports execution-time measurements.

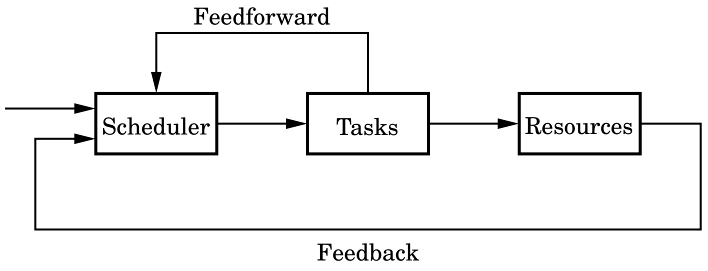


Figure 1.1 A general feedback scheduling structure. The resources are distributed among the tasks based on feedback from the actual resource use. The tasks can be use feedforward to notify the scheduler about changes in their resource demands.

A Novel Computational Model

The scheduling techniques outlined above are mainly based on the fixed-priority scheduling algorithm, which is the standard scheduling mechanism in commercial real-time operating systems (RTOS). This algorithm has several drawbacks, however, from both control and scheduling points of view. First, it introduces very irregular and hard-to-analyze delay patterns in the control loops. Second, the processor cannot be fully utilized, and the utilization bound depends on the task parameters in a non-trivial way. All of this makes for an extremely complicated control and scheduling codesign problem. In practice, the true performance of the controller cannot be known until it is running in the target system.

To remedy this problem, we propose a novel computational model for real-time control tasks, called the Control Server. The model assumes a real-time operating system that supports the earliest-deadline-first (EDF) scheduling algorithm, an optimal algorithm, which has yet to gain widespread use in commercial real-time operating systems. Interesting properties of the model include small jitter, short input-output latencies, and isolation between unrelated tasks. A control task has a single adjustable parameter—the CPU utilization factor—that uniquely determines both the control performance and the schedulability of the task. The utilization factor serves as a simple interface between the control design and the real-time design.

Controllers executing under the Control Server model can be viewed as *scalable real-time components* with well-defined control and scheduling properties. The model can ideally be combined with a feedback scheduling mechanism that optimizes the overall control performance as the system workload changes. The simple interface between the control and the scheduling design makes for a reasonably simple codesign problem to be solved on-line.

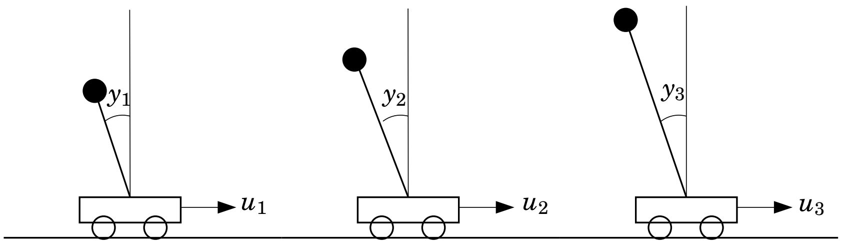


Figure 1.2 Three inverted pendulums should be stabilized using one computer.

1.4 An Introductory Example

As an introductory example, and to illustrate some of the contributions of the thesis, we will study the toy control problem depicted in Figure 1.2. Three inverted pendulums of different lengths should be controlled by a computer with (very) limited computational resources. A linear digital controller is designed for each pendulum (the details are given in Section 3.5). The pendulum lengths motivate different sampling intervals for the different controllers: h_1, h_2, h_3 = 20, 29, 35 ms. An ideal simulation of the control system where the execution of the control algorithm is disregarded is shown in Figure 1.3. It is seen that all pendulums are quickly stabilized.

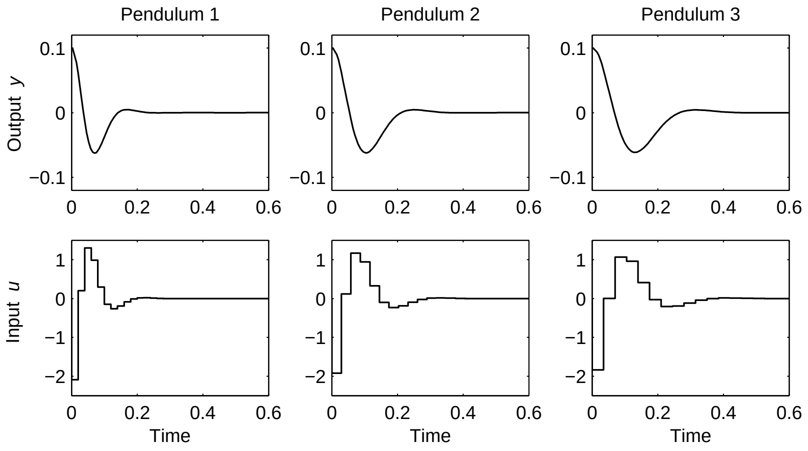


Figure 1.3 Ideal simulation of the inverted pendulum system.

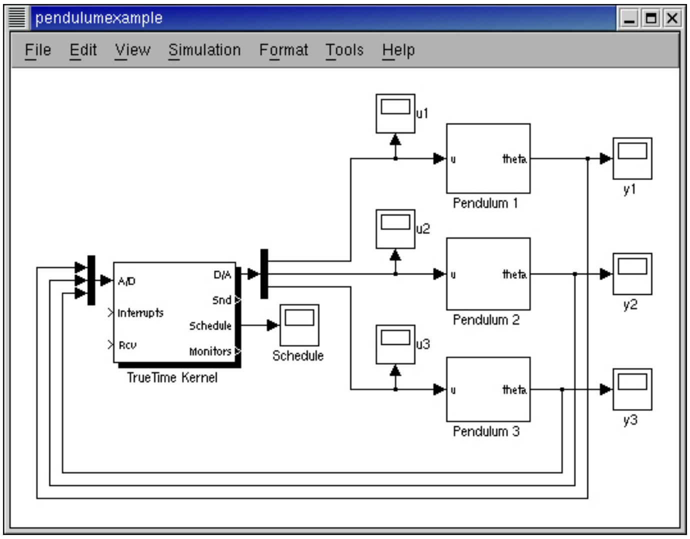


Figure 1.4 TrueTime simulation model of the inverted pendulum system. The TrueTime Kernel block simulates a real-time operating system that executes user-defined tasks.

Simulation Using TrueTime

The previous simulation of the inverted pendulum system did not capture the true performance of the controllers. First, the execution time of the control algorithm was disregarded. Second, the fact that the three controllers are executing as real-time tasks in the same CPU was ignored. To capture the true, timely behavior of the controllers executing in a real-time operating system, the TrueTime simulator can be used. A TrueTime model of the inverted pendulum system is shown in Figure 1.4. The TrueTime Kernel block simulates a full real-time operating system that executes user-defined tasks. The block has connections for analog inputs and outputs, external interrupts, network communication, etc. There is also a Schedule output that displays a trace of the executing tasks.

In the simulator, the execution time of the control algorithm is specified to $C = 7$ ms. It is assumed that rate-monotonic scheduling is used, i.e., the task with the highest rate (shortest period) is assigned the highest priority. The utilization of the task set is

$$U = \sum \frac{C}{h_i} = 0.79,$$

so the CPU is not overloaded. A simple schedulability test (see Section 2.2) shows that all deadlines will be met (assuming a relative deadline equal

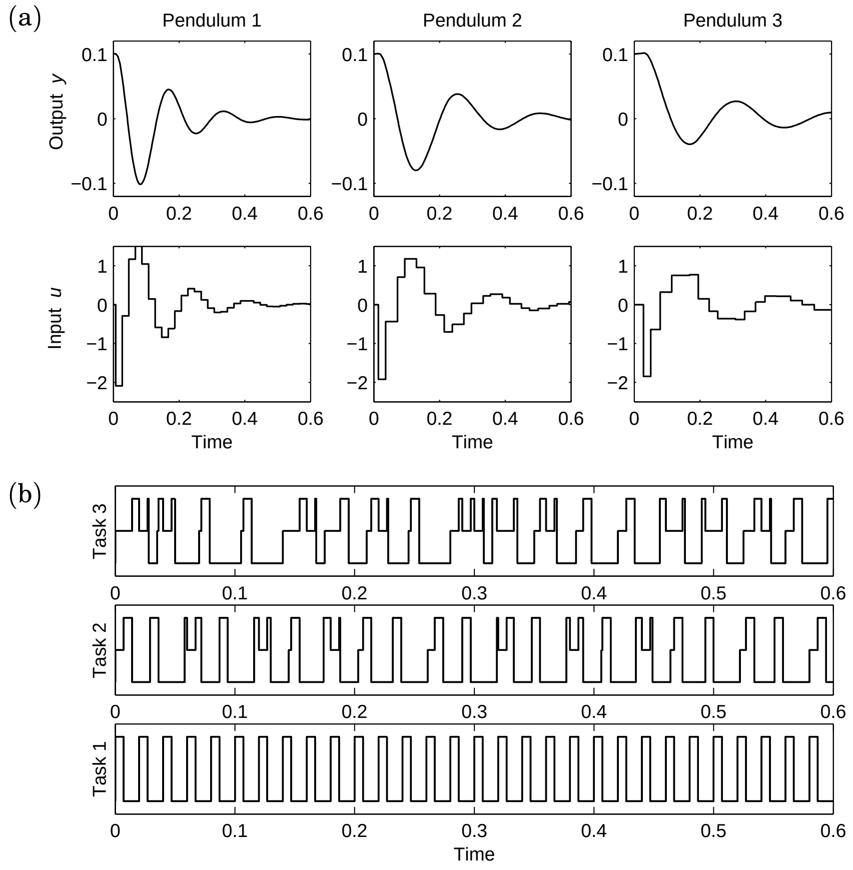


Figure 1.5 TrueTime simulation of the inverted pendulum system under rate-monotonic scheduling: (a) control performance, and (b) task schedule. The scheduling-induced latencies in the control loops deteriorate the performance. (In the schedule plots, a high level means that the task is running, a medium level that it is preempted, and a low level that it is sleeping.)

to the task period for all tasks). A simulation of the system when the tasks are released simultaneously at time zero is shown in Figure 1.5. The performance of the controllers is quite poor due to the scheduling-induced latency in the control loops. It is seen that Task 3 has a very irregular execution pattern due to preemption from Tasks 1 and 2.

From a longer simulation, it is possible to gather statistics about various relevant task timing intervals. Figure 1.6 shows the distribution of three such intervals for Task 3. The first interval, L_s, is the *sampling latency*. This is the time from the release of the task (i.e., when the task

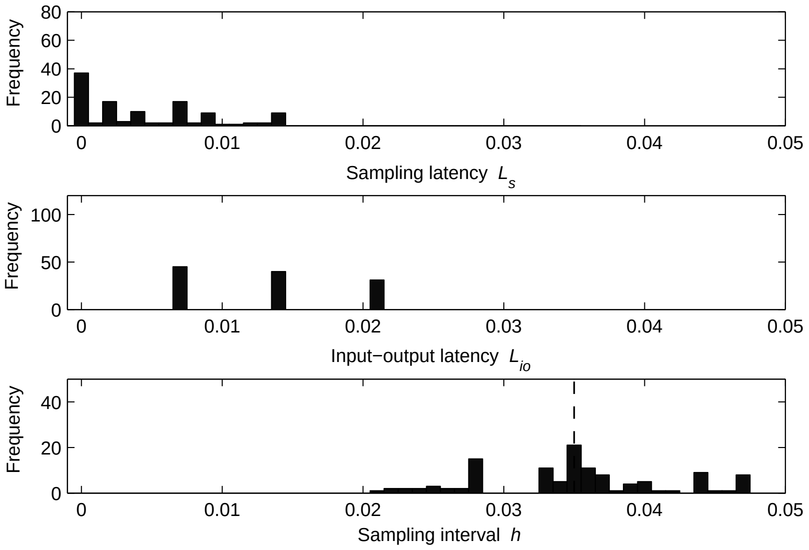


Figure 1.6 Distribution of the sampling latency, the input-output latency, and the sampling interval for Task 3 under rate-monotonic scheduling. The controller is designed for the sampling interval $h = 0.035$.

is placed in the ready queue) to the actual start of the task (i.e., when the task becomes running), at which point the plant is assumed to be sampled. The second interval, L_{io}, is the *input-output latency*. This is the time from the sampling operation to the actuation operation. The third interval, h, is the sampling interval, i.e., the time between two successive sampling operations. It is seen that the rate-monotonic scheduling algorithm introduces quite large variations in all of these intervals.

Analysis Using Jitterbug

The TrueTime simulation of the control system showed that the performance deteriorated under rate-monotonic scheduling. One way to quantify the control performance degradation is to use a *cost function*. We could for instance measure the stationary variance of the pendulum angle y,

$$J = \mathbf{E}\ y^2(t) = \lim_{T\to\infty} \frac{1}{T} \int_0^T y^2(t)\,dt, \tag{1.1}$$

assuming that the plant is disturbed by white noise. The cost function could be evaluated numerically using very long simulations in TrueTime. A better alternative is to use Jitterbug, where cost functions of the type above can be computed analytically.

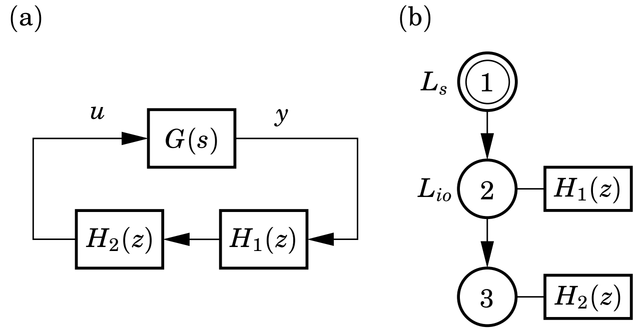


Figure 1.7 Jitterbug model of the inverted pendulum controller: (a) signal model, and (b) timing model.

A Jitterbug model the inverted pendulum controller is shown in Figure 1.7. The signal model consists of three connected linear systems. The pendulum process is described the continuous-time system $G(s)$ and is assumed to be disturbed by white input noise with unit variance. The controller is described by two discrete-time blocks, $H_1(z)$ and $H_2(z)$. The first block models the sampling operation, while the second block models the computation and actuation of the control signal. The associated timing model consists of three nodes. The first node is periodic and represents the release of the control task. There is a random delay L_s until the second node where H_1 is updated, and another random delay L_{io} until the third node where H_2 is updated.

To evaluate the performance of Task 3 in the ideal case, the delays are set to zero in the timing model ($L_s = 0$, $L_{io} = 0$). In this case, Jitterbug evaluates the cost function (1.1) to $J = 0.37$. Under rate-monotonic scheduling, we assume the probability distributions of L_s and L_{io} from Figure 1.6. In this case, the cost is computed to $J = 0.68$, i.e., the variance of the pendulum angle is considerably larger.

Jitterbug can also be used for frequency-domain analysis. In Figure 1.8, the spectral density of the output of Pendulum 3 has been computed in the ideal case and the rate-monotonic scheduling case. The resonance peak in the rate-monotonic case agrees with the oscillatory behavior seen in Figure 1.5.

In most cases, exact knowledge of timing distributions will not be available at early design stages. Instead, Jitterbug can be used to estimate the timing sensitivity of a controller by sweeping various timing parameters and plotting the results. The resulting diagrams can be used as a guide in the later real-time design. In Figure 1.9, the cost function (1.1) for Task 3 has been calculated for various amounts of input-output latency

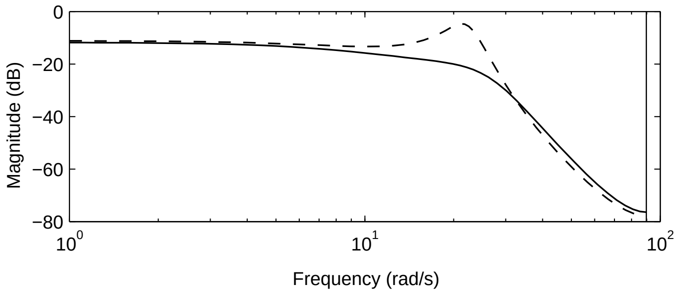


Figure 1.8 Spectral density of the output of Pendulum 3 in the ideal case (full) and in the rate-monotonic scheduling case (dashed). The resonance peak in the rate-monotonic case agrees with the oscillatory behavior seen in Figure 1.5.

and input-output jitter. In the timing model, we have assumed zero sampling latency ($L_s = 0$) and a uniformly distributed input-output latency L_{io} between $\bar{L}_{io} - J_{io}/2$ and $\bar{L}_{io} + J_{io}/2$, where $\bar{L}_{io}$ denotes the average latency and J_{io} denotes the jitter. It is seen that the controller is quite sensitive to input-output latency but not very sensitive to jitter.

Subtask Scheduling

The example so far has shown that schedulability of task does not guarantee any level of control performance. From a control perspective, it is

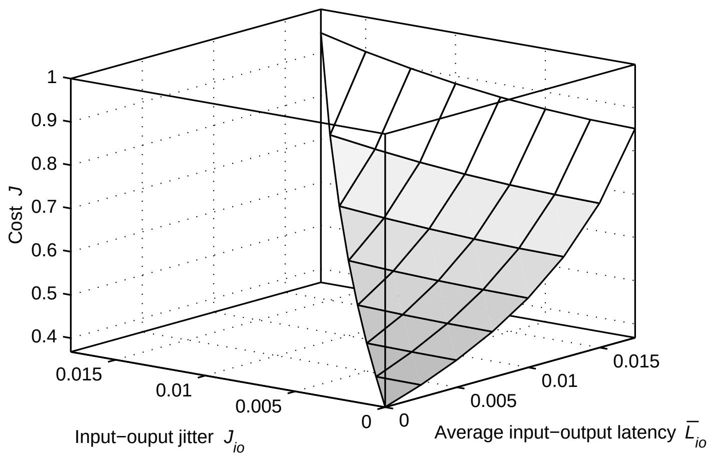


Figure 1.9 Cost as a function of the average input-output latency and the input-output jitter for Task 3. The cost increases more rapidly in the latency direction than the jitter direction.

not only important to finish the computations before the end of the period. It is also of importance that the sampling and actuation operations are performed regularly and with a short input-output latency. For this purpose, more detailed scheduling models can be used, where the different parts of the control algorithm are scheduled individually.

A typical control loop can be implemented as shown in Listing 1.1. Here, the control algorithm has been split into two parts: Calculate Output and Update State. Calculate Output contains only the operations needed to produce a control signal, while the rest of the computations are postponed to Update State. The idea of subtask scheduling is to assign different priorities to Calculate Output (which is the time-critical part) and Update State (which only has to finish before the end of the period).

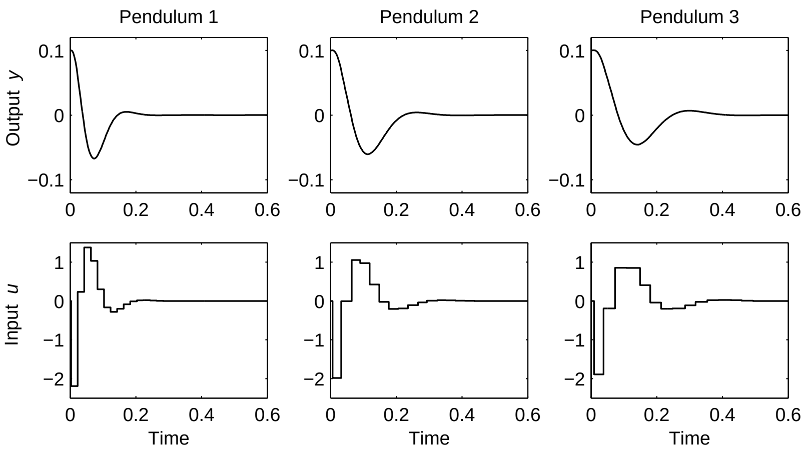
To continue the example, we assume that the execution time of the first part is 3 ms and the execution time of the second part is 4 ms. A subtask scheduling analysis shows that, in this case, it is possible to assign the highest priorities to the Calculate Output parts, while the Update State parts are given lower priorities. A new simulation of the inverted pendulum system is shown in Figure 1.10. It can be seen that the control performance is considerably improved. This is due to the reduction of latency and jitter in the control loops. The improvement is particularly large for Task 3, which was the lowest-priority controller under rate-monotonic scheduling. The new distributions of L_s, L_{io} and h for Task 3 are shown in Figure 1.11.

The improved level of performance for Task 3 can also be verified by a new Jitterbug calculation. Inserting the distributions from Figure 1.11 into the timing model gives a cost of $J = 0.41$, which is quite close to the cost in the ideal case.

Listing 1.1 Typical implementation of a control loop. The control algorithm is split into two parts: Calculate Output and Update State.

```
LOOP
  ReadInput;
  CalculateOutput;
  WriteOutput;
  UpdateState;
  WaitForNextPeriod;
END;
```

(a)



(b)

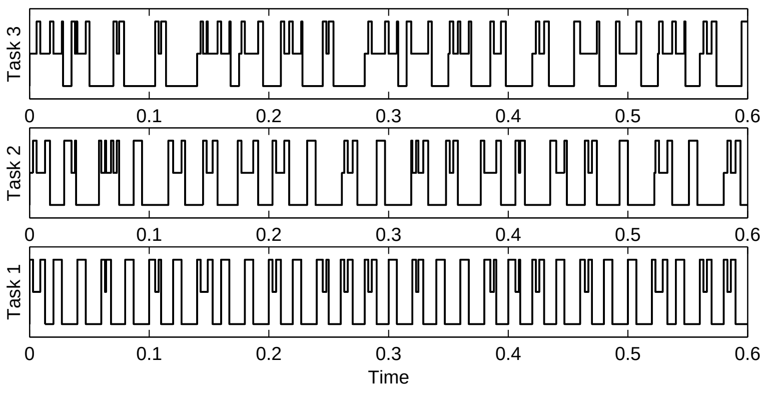


Figure 1.10 TrueTime simulation of the inverted pendulum system under subtask scheduling: (a) control performance, and (b) task schedule. The performance is very close to the ideal case (see Figure 1.3). The scheduling strategy increases the number of context switches compared to rate-monotonic scheduling (see Figure 1.5).

Feedback Scheduling

Scheduling analysis is normally applied off-line, assuming that values for all scheduling parameters (periods, deadlines, and worst-case computation times) are known. This is the appropriate approach for *hard* real-time systems, where it must be guaranteed, a priori, that all deadlines are met. For many control systems, this is an overly rigid approach. A single missed deadline does not mean system failure. This fact can be exploited to create more dynamic real-time control systems, where the resources are better utilized. However, in such systems, transient CPU overloads may occur, and the scheduler must keep the overload intervals

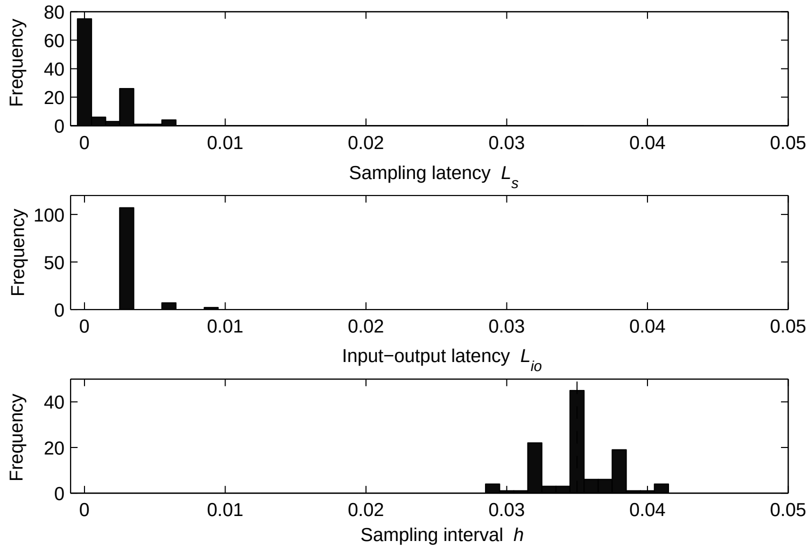


Figure 1.11 Distribution of the sampling latency, the input-output latency, and the sampling interval for Task 3 under subtask scheduling. Compared with the rate-monotonic scheduling case (Figure 1.6), the latencies are shorter and the sampling interval is more closely centered around the nominal interval $h = 0.035$.

as short as possible.

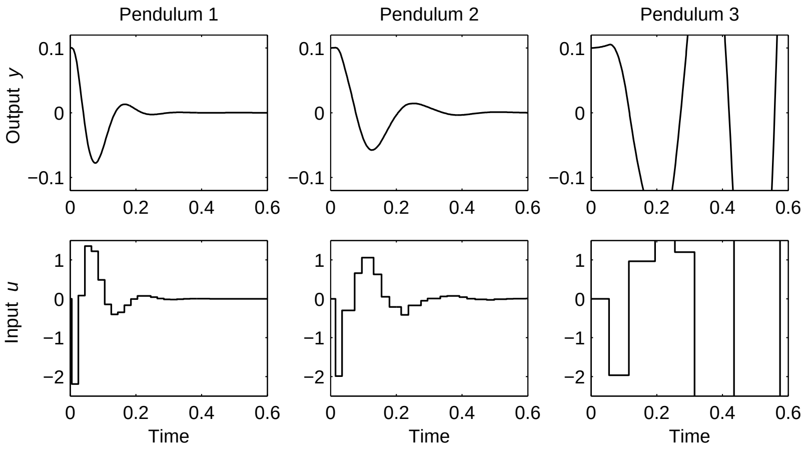
To continue the example, suppose that the execution time of the control algorithm has been underestimated. The designer believes that the execution time is 7 ms, while the true execution time is 10 ms. Using the same sampling periods as before, the CPU utilization is now

$$U = \sum \frac{C}{h_i} = 1.13,$$

i.e., the processor is overloaded. A simulation of the system under rate-monotonic scheduling is shown in Figure 1.12. Due to the overload, Task 3 is blocked most of the time, causing the control loop to be destabilized.

Next, a feedback scheduler is introduced. The scheduler attempts to control the CPU utilization to 80% by rescaling the sampling periods of the controllers. The feedback scheduler is implemented as a high-priority task that regularly collects execution-time measurements from the control tasks and estimates the CPU load. Based on the load estimate, new sampling periods are communicated to the control tasks. A simulation of the inverted pendulum system under feedback scheduling is shown in Figure 1.13. Task 3 is blocked during an initial transient until the feed-

(a)



(b)

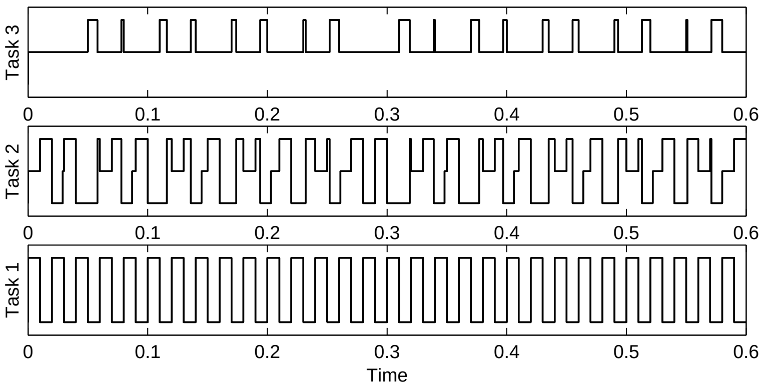
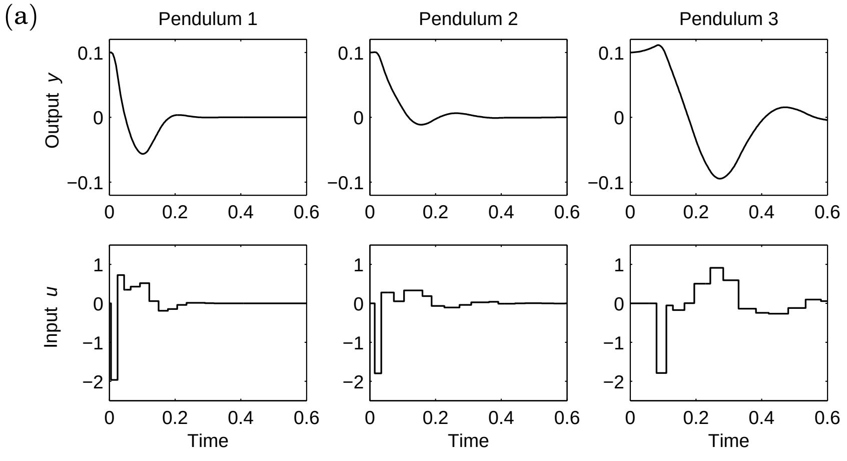


Figure 1.12 TrueTime simulation of the inverted pendulum system under rate-monotonic scheduling and an overloaded CPU: (a) control performance, and (b) task schedule. Task 3 is preempted most of the time, causing the control loop to be destabilized.

back scheduler has detected the overload and controlled the utilization to below 100%.

The Control Server

The example so far has assumed rate-monotonic scheduling of the control tasks. We have seen that this type of scheduling introduces unpredictable latency patterns in the control loops (although the magnitude of the latencies can be reduced using subtask scheduling). This problem becomes even more pronounced during CPU overloads, where a low-priority task can become completely blocked.



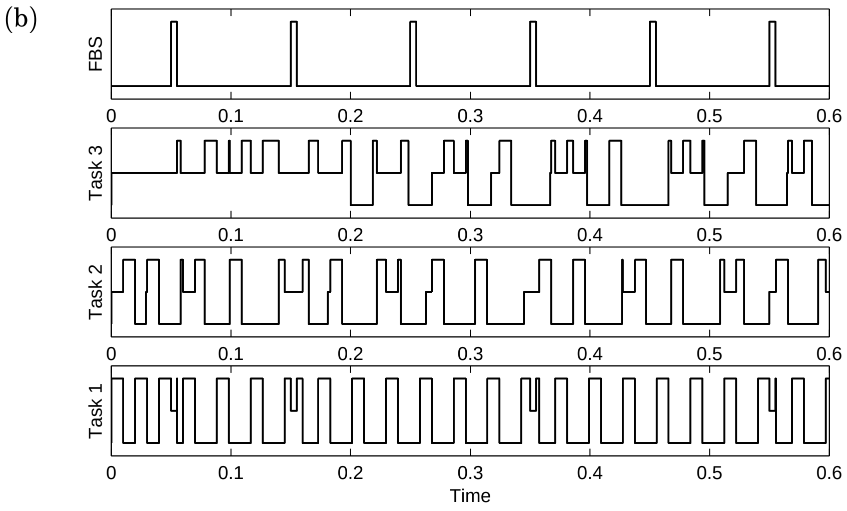
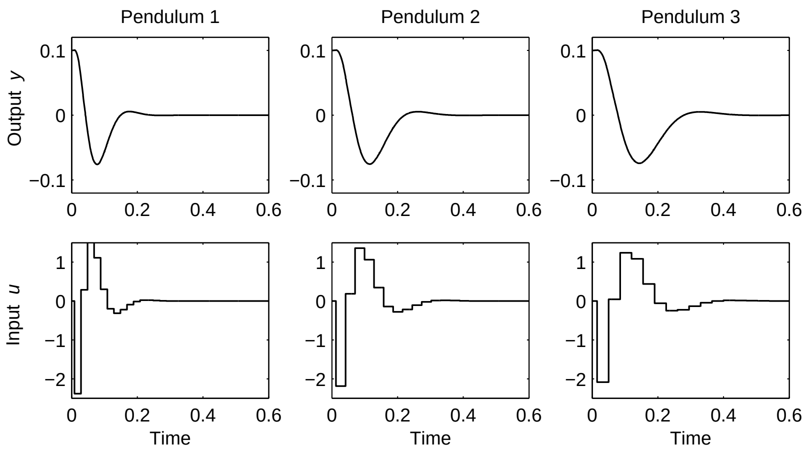


Figure 1.13 TrueTime simulation of the inverted pendulum system under feedback scheduling: (a) control performance, and (b) task schedule. The feedback scheduler (FBS) regularly adjusts the sampling periods of the controllers, resolving the initial overload situation.

Under the Control Server (CS) model, each task is *guaranteed* a certain fraction of the CPU time, using server-based scheduling. Furthermore, the sampling and actuation instants are statically scheduled (i.e., handled by the kernel) in order to minimize the jitter. Combined, these two properties give a CS task a completely predictable behavior.

Since the Control Server is based on EDF, the CPU can be fully uti-

(a)



(b)

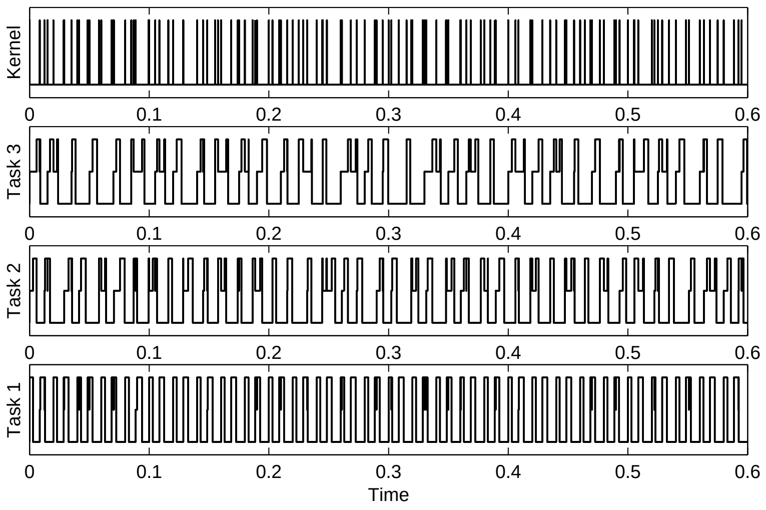


Figure 1.14 TrueTime simulation of the inverted pendulum system under the Control Server model: (a) control performance, and (b) task schedule. The performance is very close to the ideal case (see Figure 1.3). The kernel handles the I/O operations of all the control tasks.

lized. In the inverted pendulum example, this means that shorter sampling periods can be used. Each controller will have a constant input-output latency (shorter than the period), which can be compensated for in the controller design. A simulation of the inverted pendulum system under the Control Server model is shown in Figure 1.14. The performance is very close to the ideal case (see Figure 1.3).

1.5 Outline and Publications

The outline of the thesis is given below, together with references to related publications.

Chapter 2: Background

An introduction to real-time scheduling theory is given, followed by an overview of control loop timing. A brief survey of existing control and scheduling codesign approaches is given.

Publications

Årzén, K.-E., B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson, and L. Sha (1999): "Integrated control and scheduling." Technical Report ISRN LUTFD2/TFRT--7586--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Årzén, K.-E., A. Cervin, J. Eker, and L. Sha (2000): "An introduction to control and scheduling co-design." In *Proceedings of the 39th IEEE Conference on Decision and Control*. Sydney, Australia.

Cervin, A. (2000): "Towards the integration of control and real-time scheduling design." Licentiate Thesis ISRN LUTFD2/TFRT--3226--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

The Bluetooth jitter compensation example appears in

Eker, J., A. Cervin, and A. Hörjel (2001): "Distributed wireless control using Bluetooth." In *Proceedings of the IFAC Conference on New Technologies for Computer Control*. Hong Kong, P.R. China.

Chapter 3: Subtask Scheduling

This chapter considers scheduling of simple controllers where the control algorithm has been split into two parts: Calculate Output and Update State. The goal is to reduce the input-output latencies in the control loops. Subtask deadline assignment under fixed-priority and earliest-deadline first scheduling is treated. The control performance improvements are verified in an inverted pendulum example.

Publications

Cervin, A. (1999): "Improved scheduling of control tasks." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*. York, UK.

Chapter 4: Feedback Scheduling

A scheduling architecture is proposed, where the CPU load is controlled by adjusting the sampling intervals of a set of controllers. The goal is to maintain high utilization and good control performance in spite of large variations in execution time. Feedforward from mode changes is used to further improve the regulation. A heuristic approach is discussed first, where simple rescaling of the nominal sampling periods to reach the utilization setpoint is used. The approach is exemplified on a set of hybrid controllers. It is later shown that simple period rescaling is in fact optimal for controllers with certain cost functions. Overloaded open-loop EDF scheduling is also discussed, and it is shown that overloaded EDF is equivalent to linear period rescaling.

Publications

Cervin, A. and J. Eker (2000): “Feedback scheduling of control tasks.” In *Proceedings of the 39th IEEE Conference on Decision and Control*. Sydney, Australia.

Persson, P., A. Cervin, and J. Eker (2000): “Execution-time properties of a hybrid controller.” Technical Report ISRN LUTFD2/TFRT--7591--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Cervin, A., J. Eker, B. Bernhardsson, and K.-E. Årzén (2002): “Feedback-feedforward scheduling of control tasks.” *Real-Time Systems*, **23:1/2**.

A preliminary study of feedback scheduling of model predictive controllers is presented in

Henriksson, D., A. Cervin, J. Åkesson, and K.-E. Årzén (2002): “Feedback scheduling of model predictive controllers.” In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*. San Jose, CA.

Chapter 5: The Control Server

A new computational model for real-time control tasks is presented, with the primary goal of simplifying the control and scheduling codesign problem. The model combines time-triggered I/O and inter-task communication with dynamic, reservation-based task scheduling. To facilitate short input-output latencies, a task may be divided into several segments. Jitter is reduced by allowing communication only at the beginning and at the end of a segment. A key property of the model is that both schedulability and control performance of a control task will depend on the reserved utilization factor only. This enables controllers to be treated as scalable real-time components. The model has been implemented in a public domain real-time kernel and validated in control experiments.

Publications

Cervin, A. and J. Eker (2003): "The Control Server: A computational model for real-time control tasks." In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*. Porto, Portugal. (To appear in June 2003.)

Chapter 6: Analysis Using Jitterbug

The MATLAB-based toolbox Jitterbug is presented. The tool allows the user to compute a quadratic performance index for a control loop under various timing conditions. The control system is built from a number of continuous- and discrete-time linear systems, and the execution of the controller is described by a stochastic timing model. The toolbox is also capable of computing the spectral densities of the signals in the system. Using the tool, it is easy to investigate the impact of delays, jitter, lost samples, aperiodic execution, etc., on the control performance. A number of examples are given.

Publications

Lincoln, B. and A. Cervin (2002): "Jitterbug: A tool for analysis of real-time control performance." In *Proceedings of the 41st IEEE Conference on Decision and Control*. Sydney, Australia.

Cervin, A. and B. Lincoln (2003): "Jitterbug 1.1—Reference manual." Technical Report ISRN LUTFD2/TFRT--7604--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

(There are also two publications that describe both Jitterbug and TrueTime, see below.)

Chapter 7: Simulation Using TrueTime

The MATLAB/Simulink-based simulator TrueTime is presented. The simulator allows detailed co-simulation of continuous plant dynamics, real-time scheduling, control task execution, and message transmission in distributed real-time control systems. The TrueTime Kernel block is described in detail, and a number of examples are given.

Publications

Henriksson, D., A. Cervin, and K.-E. Årzén (2002): "TrueTime: Simulation of control loops under shared computer resources." In *Proceedings* *of the 15th IFAC World Congress on Automatic Control*. Barcelona, Spain.

Henriksson, D. and A. Cervin (2003): “TrueTime 1.1—Reference manual.” Technical Report ISRN LUTFD2/TFRT--7605--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Both Jitterbug and TrueTime are described in

Cervin, A., D. Henriksson, B. Lincoln, and K.-E. Årzén (2002): “Jitterbug and TrueTime: Analysis tools for real-time control systems.” In *Proceedings of the 2nd Workshop on Real-Time Tools*. Copenhagen, Denmark.

Cervin, A., D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén (2003): “How does control timing affect performance?” *IEEE Control Systems Magazine*. (To appear in June 2003.)

An old version of the simulator is described in

Eker, J. and A. Cervin (1999): “A Matlab toolbox for real-time and control systems co-design.” In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*. Hong Kong, P.R. China.

Cervin, A. (2000): “The real-time control systems simulator—Reference manual.” Technical Report ISRN LUTFD2/TFRT--7592--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Chapter 8: Conclusion

The contents of the thesis are summarized and suggestions for future work are given.

2

Background

2.1 Introduction

In textbooks on real-time systems, e.g., [Burns and Wellings, 2001; Liu, 2000; Buttazzo, 1997; Krishna and Shin, 1997], control systems are used as the prime example of *hard real-time systems*. In a hard real-time system, the computer must respond to events within specified deadlines—otherwise, the system will fail. In the case of a control system, the computer must respond to incoming measurement signals, producing new control signals fast enough to keep the plant within its operational limits.

The controller is often just one component among many in an embedded computer system. There may be several more controllers executing on the same unit, together with communication tasks, operator interfaces, tasks for data logging, etc. The concurrent activities are typically implemented on a microprocessor using a sequential programming language such as C together with a real-time operating system, or using a more modern language such as Ada or Java that has direct support for concurrent programming.

In an embedded system, the processor (CPU) time is a shared resource for which the various tasks compete. To guarantee, a priori, that all tasks will meet their deadlines, static (off-line) schedulability analysis must be performed. The analysis assumes a certain task model and that the relevant task attributes are known. The traditional view in the hard real-time scheduling community is that a digital controller can be modeled as a task that has

- a fixed period,
- a hard deadline equal to the period, and
- a known worst-case execution time (WCET).

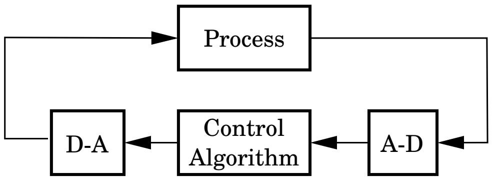


Figure 2.1 A computer-controlled system. The control task consists of three distinct parts: input data collection (A-D), control algorithm computation, and output signal transmission (D-A).

While this simple model is appropriate for some control applications, there are also a number of cases where it is either too simplistic or too rigid.

First, the model is only concerned with the scheduling of the pure *control computations*. However, a control task generally consists of three distinct operations: input data collection, control algorithm computation, and output signal transmission, see Figure 2.1. The timing of the input and output operations are crucial to the performance of the controller. A delay (or *latency*) between the input and the output decreases the stability margin of the control loop. Jitter in the inputs and outputs also causes the performance to degrade. Since the input and output operations are neglected in the task model, there are no means of controlling the latency and the jitter in the scheduling design.

Second, the model assumes that worst-case estimates of the controller execution times are available. In practice, it is very difficult to obtain a tight bound on the WCET. Very few analytical tools are available, and the ones that exist cannot handle all the features of modern computing hardware, such as multi-level caches, pipelines, and speculative branching. The net result is that WCET estimates tend to be very pessimistic. Furthermore, some control algorithms have varying execution-time demands. One example is hybrid controllers that switch between different internal modes. Another example is model predictive controllers (MPCs) that solve an optimization problem in each sample. Basing the analysis on the WCETs of these controllers may lead to very low average utilization of the processor.

Third, the model assumes that all controller deadlines are hard. This this is only a simplifying assumption, however. The sampling rate of a controller is typically chosen to be several times faster than what is dictated by Shannon's sampling theorem and pure stability concerns. A single missed deadline only means that the current sampling interval becomes somewhat longer. The net effect can be interpreted as a small disturbance acting on the process. Only in the case of longer CPU overloads will the stability of the plant be endangered.

In the following chapters of the thesis, modifications to the simple task model that remedy some of the problems above will be given. More detailed task models allow better control of latency and jitter in the control loops. Relaxing the requirements on known WCETs and hard deadlines allows the computing resources to be used more efficiently. Accepting some degree of non-determinism in real-time control systems also permits the use of COTS hardware and software. This can lead to lower development costs for embedded systems.

The rest of this chapter contains background material on real-time scheduling theory, control loop timing, and related work in the area of control and real-time scheduling codesign.

2.2 Real-Time Scheduling Theory

Real-time scheduling theory is used to predict whether the tasks in a real-time system will meet their individual timing requirements. Given a task model and a scheduling algorithm, off-line analysis is performed to check, for instance, whether all deadlines will be met during runtime.

Two main design approaches exist: *static scheduling* and *dynamic scheduling*. Static scheduling is an off-line approach that uses optimization-based algorithms to generate a cyclic executive. An execution table states the order in which the different tasks should execute and for how long they should execute. An advantage of the cyclic executive is that it is very simple to implement. The approach also has many drawbacks [Locke, 1992]. First, there is the difficulty of constructing the schedule itself. Second, it is hard to incorporate sporadic and aperiodic tasks. Third, tasks with long execution times may have to be split in many small pieces, making the code error-prone and difficult to read. Fourth, very long tables may be needed if the schedule incorporates tasks with period times that are relative prime. Due to these drawbacks, we will mainly consider dynamic scheduling in the sequel.

There exist a large number of dynamic scheduling policies. Here we will focus on the *fixed-priority* (FP) and the *earliest-deadline-first* (EDF) scheduling policies, both introduced in the seminal paper [Liu and Layland, 1973]. In the scheduling analysis, a basic task model is assumed, where each task τ_i is described by

- a period (or minimum inter-arrival time) T_i,
- a relative deadline D_i, and
- a worst-case execution time C_i.

Furthermore, in the simplest case, it is assumed that the tasks are independent (i.e., they do not communicate or share other resources than the CPU), and that there is no kernel overhead.

Fixed-Priority Scheduling

Fixed-priority (FP) scheduling is the most common scheduling policy and is supported by most commercial real-time operating systems. Under FP scheduling, each task τ_i is assigned a fixed priority P_i. If several tasks are ready to run at the same time, the task with the highest priority gets access to the CPU. If a task with higher priority than the running task should become ready, the running task is preempted by the other task.

Priority Assignment. Fixed-priority scheduling design consists of assigning priorities to all tasks before runtime. In [Liu and Layland, 1973] it was shown that the *rate-monotonic* (RM) priority assignment is optimal when $D_i = T_i$ for all tasks. Each task is assigned a priority based on its period: the shorter the period, the higher the priority. The scheme is optimal in the sense that, if the task set is not schedulable under the rate-monotonic priority assignment, it will not be schedulable under any other fixed-priority assignment. By *rate-monotonic scheduling*, we mean fixed-priority scheduling where the tasks have been assigned rate-monotonic priorities.

In many cases, it is desirable to specify deadlines that are shorter than the period. In the case where $D_i \leq T_i$ for all tasks, the *deadline-monotonic* (DM) priority assignment scheme is optimal (in the same sense as above) [Leung and Whitehead, 1982]. Each task is assigned a priority based on its relative deadline: the shorter the deadline, the higher the priority. Note that the RM priority assignment is merely a special case of the DM priority assignment.

Schedulability Analysis. Given a task set with known attributes (periods, deadlines, execution times, and priorities), a number of different tests can be applied to check whether all tasks will meet their deadlines. Assuming the rate-monotonic priority assignment, a sufficient (but not necessary) schedulability test is obtained by considering the utilization of the task set [Liu and Layland, 1973]. Assuming a set of n tasks, all tasks will meet their deadlines if

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1). \tag{2.1}$$

As the number of tasks becomes large, the utilization bound approaches $\ln 2 \approx 0.693$.

An exact schedulability test under FP scheduling is performed by computing the *worst-case response time* R_i of each task [Joseph and Pandya, 1986]. The response time of a task is defined as the time from its release to its completion. The maximum response time of a task occurs when all other tasks are released simultaneously. The worst-case response time of a task τ_i is given by the recursive equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.2}$$

where $hp(i)$ is the set of tasks with higher priority than τ_i and $\lceil x \rceil$ denotes the ceiling function. The task set is schedulable if and only if $R_i \leq D_i$ for all tasks.

Extensions. Many extensions to the theory of FP scheduling exist, e.g., [Klein *et al.*, 1993]. The analysis has been extended to handle for instance common resources, release jitter, tick scheduling, nonzero context-switching times, and clock interrupts.

The analysis behind the schedulability conditions is based on the notion of the *critical instant*. This is the situation when all tasks are released simultaneously. If the task set is schedulable for this worst case, it will be schedulable also for all other cases. In many cases, this assumption is unnecessarily restrictive. Tasks may have precedence constraints that make it impossible for them to arrive at the same time. For independent tasks it is sometimes possible to introduce *release offsets*, to avoid the simultaneous releases. If simultaneous releases can be avoided, the schedulability of the task set may increase [Audsley *et al.*, 1993]. Formulas for exact response-time calculations for tasks with static release offsets is given in [Redell and Törngren, 2002]. Schedulability analysis for tasks with dynamic offsets is discussed in [Gutierrez and Harbour, 1998]. A number of alternative scheduling models based on serialization of task executions in different ways have been suggested. These include the multi-frame model [Baruah *et al.*, 1999b] and the serially executed subtask model [Harbour *et al.*, 1994].

Recently, formulas for the *best-case response time* of tasks under fixed-priority scheduling have been derived [Redell and Sanfridson, 2002]. Knowing both the worst-case and the best-case response time of a task gives a measure of the response-time jitter. In its simplest form, the best-case response time R_i^b of task τ_i is given by the recursive equation

$$R_i^b = C_i^b + \sum_{j \in hp(i)} \left\lceil \frac{R_i^b - T_j}{T_j} \right\rceil C_j^b \tag{2.3}$$

where C_i^b denotes the *best-case execution time* of task τ_i.

Earliest-Deadline-First Scheduling

Under EDF scheduling, the task with the shortest time to its deadline is chosen for execution. The absolute deadline of a task can hence be interpreted as a dynamic priority. Because of its more dynamic nature, EDF can schedule a larger set of task sets than the FP scheduling policy can. Despite its theoretical advantages, EDF has so far mainly been used in experimental real-time operating systems. A reason for this may be that EDF also has a number of potential drawbacks compared to FP scheduling [Burns and Wellings, 2001]:

- The implementation of EDF is slightly more complex; the dynamic priority incurs a larger runtime overhead and requires more storage.
- It may be difficult to assign artificial deadlines to tasks that have no explicit deadlines.
- During overloads, all tasks tend to miss their deadlines (this is known as the *domino effect*).

The arguments above should not be taken too seriously. It can be argued that it is more difficult to assign meaningful priorities than deadlines in a real-time system. Furthermore, control tasks running under EDF scheduling actually tend to behave better in overload situations than under FP scheduling (see Chapter 4).

Schedulability Analysis. In the case where $D_i = T_i$ for all tasks, the schedulability of a task set under EDF is exactly determined by the processor utilization [Liu and Layland, 1973]. All tasks will meet their deadlines if and only if

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{2.4}$$

Note that this test is much simpler than the exact schedulability test under FP scheduling (which requires that the response times are computed). Also note that the processor can be fully utilized under EDF.

For the case $D_i \leq T_i$, the analysis becomes more difficult. A very general schedulability test under EDF involves computing the *loading factor* of the tasks (see [Stankovic *et al.*, 1998]). The analysis is performed as follows. Given an arbitrary set of tasks, let each job (task instance) j_k be described by a computation time C_k, a release time r_k, and an absolute deadline d_k. Define the *processor demand* of the task set on a time interval $[t_1, t_2]$ as

$$h[t_1, t_2] = \sum_{r_k \geq t_1 \wedge d_k \leq t_2} C_k. \tag{2.5}$$

Next, define the *loading factor* u of the tasks as

$$u = \max_{0 \le t_1 < t_2} \frac{h[t_1, t_2]}{t_2 - t_1}, \qquad (2.6)$$

where the maximization is performed over all possible time intervals. The task set is schedulable under EDF if and only if $u \le 1$.

Response-Time Analysis. Worst-case response-time analysis under EDF scheduling is more difficult than under FP scheduling. The main problem is that there is no well-defined critical instance at which the task will experience its maximum interference. Nevertheless, formulas have been derived for response-time calculations under EDF, see [Stankovic *et al.*, 1998].

Server-Based Scheduling

Many types of tasks do not fit the simple periodic task model. These include *aperiodic* and *soft* real-time tasks. To retain the guarantees of the hard real-time tasks, these types of tasks can be incorporated in the real-time system using *servers*. The main idea of the server-based scheduling is to have a special task, the server, for scheduling the pending aperiodic workload (emanating from one or several aperiodic tasks). The server has a budget that is used to schedule and execute the pending jobs. The aperiodic tasks may execute until they finish or until the budget has been exhausted. Several servers have been proposed. The *priority-exchange server* and the *deferrable server* were proposed in [Lehoczky *et al.*, 1987]. The *sporadic server* was introduced in [Sprunt *et al.*, 1989]. The main difference between the servers concerns the way the budget is replenished and the maximum capacity of the server.

The above servers have been developed for the fixed-priority case. Similar techniques also exist for the dynamic-priority case (i.e., EDF), see, e.g., [Spuri and Buttazzo, 1996]. An EDF-based server with especially interesting properties is the *constant bandwidth server* (CBS) [Abeni and Buttazzo, 1998]. Since the server will be used in Chapter 5, it is described in detail below.

The Constant Bandwidth Server. A CBS creates the abstraction of a virtual CPU with a given capacity (or *bandwidth*) U_s. Tasks executing within the CBS cannot consume more than the reserved capacity. Hence, from the outside, the CBS will appear as an ordinary EDF task with a maximum utilization of U_s. The time granularity of the virtual CPU abstraction is determined by the *server period* T_s.

Associated with the server are two dynamic attributes: the server budget c_s and the server deadline d_s. Jobs that arrive to the server are placed

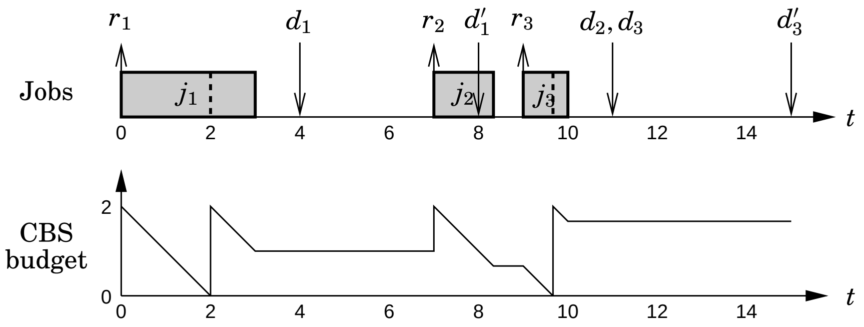


Figure 2.2 Example of a constant bandwidth server (CBS) with the bandwidth $U_s = 0.5$ and the period $T_s = 4$ serving aperiodically arriving jobs. The up arrows indicate job arrivals, and the down arrows indicate deadlines.

in a queue and are served on first-come, first-served basis. The first job in the queue is always eligible for execution (as an ordinary EDF task), using the current server deadline d_s. The server is initialized with $c_s := 0$ and $d_s := 0$. The rules for updating the server are as follows:

1. During the execution of a job, the budget c_s is decreased at unit rate.
2. Whenever $c_s = 0$, the budget is recharged to $c_s := U_sT_s$ and the deadline is postponed one server period: $d_s := d_s + T_s$.
3. If a job arrives at a time r when the server queue is empty, *and* if $c_s \geq (d_s - r)U_s$, then the budget is recharged to $c_s := U_sT_s$ and the deadline is set to $d_s := r + T_s$.

The above rules limit the server processor demand in any time interval $[t_1, t_2]$ to $U_s(t_2 - t_1)$. The third rule is used to “reset” the server after a sufficiently long idle interval. Note that postponing the deadline corresponds to lowering the dynamic priority of the server.

An example of CBS scheduling is given in Figure 2.2. The server is assumed to have the bandwidth $U_s = 0.5$ and the period $T_s = 4$. At $t = 0$, the server is empty and job j_1 arrives. The budget is charged to $U_sT_s = 2$ and the job is served with the deadline $d_1 = r_1 + T_s = 4$ (rule 3). At $t = 2$, the budget is exhausted. The budget is recharged to $U_sT_s = 2$, and the remainder of j_1 (one time unit) is served using the postponed deadline $d'_1 = d_1 + T_s = 8$ (rule 2). At $t = 7$, job j_2 arrives. Rule 3 is in effect, causing the budget to be recharged and the deadline to be set to $d_2 = r_2 + T_s = 11$. At $t = 9$, job j_3 arrives. Rule 3 is *not* in effect, so the job is served with the old server deadline $d_3 = d_2 = 11$. At $t = 9.7$, the budget is once again exhausted. The budget is recharged, and the remainder of the job (0.3 units) is served using the postponed deadline $d'_3 = d_3 + T_s = 15$.

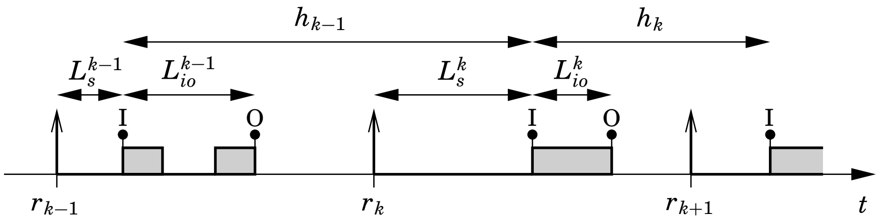


Figure 2.3 Controller timing.

2.3 Control Loop Timing

A control loop consists of three main parts: input data collection, control algorithm computation, and output signal transmission. In the simplest case the input and output operations consist of calls to an external I/O interface, e.g., A-D and D-A converters or a field-bus interface. In a more complex setting the input data may be received from other computational blocks, such as noise filters, and the output signal may be sent to other computational blocks, e.g., other control loops in the case of set-point control. The complexity of the control algorithm may range from a few lines of code implementing a PID (proportional-integral-derivative) controller to the iterative solution of a quadratic optimization problem in the case of model predictive control (MPC). In most cases the control is executed periodically with a sampling interval that is determined by the dynamics of the process that is controlled and the requirements on the closed-loop performance. A typical rule-of-thumb [Åström and Wittenmark, 1997] is that the sampling interval h should be selected such that

$$\omega_c h = 0.2\text{–}0.6, \tag{2.7}$$

where ω_c is the bandwidth of the closed-loop system. A real-time system where the product $\omega_c h$ is small for all control tasks will be less sensitive to scheduling-induced latencies and jitter, but it will also consume more CPU resources.

Timing Parameters

Ideally, the control algorithm should be executed with perfect periodicity, and there should be zero delay between the reading of the inputs and the writing of the outputs. This will not be the case in a real implementation, where the execution and scheduling of tasks introduce latencies. The basic timing parameters of a control task are shown in Figure 2.3. It is assumed that the control task is *released* (i.e., inserted into the ready queue of the real-time operating system) periodically at times given by $r_k = hk$, where h is the *nominal sampling interval* of the controller. Due to preemption from other tasks in the system, the actual *start* of the task may be delayed for some time L_s. This is called the *sampling latency* of the controller. A dynamic scheduling policy will introduce variations in this interval. The *sampling jitter* is quantified by the difference between the maximum and minimum sampling latencies in all task instances,

$$J_s \overset{\text{def}}{=} L_s^{max} - L_s^{min}. \tag{2.8}$$

Normally, it can be assumed that the minimum sampling latency of a task is zero, in which case we have $J_s = L_s^{max}$. Jitter in the sampling latency will of course also introduce jitter in the sampling interval h. From the figure, it is seen that the actual sampling interval in period k is given by

$$h_k = h - L_s^{k-1} + L_s^k. \tag{2.9}$$

The *sampling interval jitter* is quantified by

$$J_h \overset{\text{def}}{=} h^{max} - h^{min}. \tag{2.10}$$

We can see that the sampling interval jitter is upper bounded by

$$J_h \leq 2J_s. \tag{2.11}$$

After some computation time and possibly further preemption from other tasks, the controller will actuate the control signal. The delay from the sampling to the actuation is the *input-output latency*, denoted L_{io}. Varying execution times or task scheduling will introduce variations in this interval. The *input-output jitter* is quantified by

$$J_{io} \overset{\text{def}}{=} L_{io}^{max} - L_{io}^{min}. \tag{2.12}$$

The impact of input-output latency, input-output jitter, and sampling jitter on the control design and the scheduling design is discussed below.

Input-Output Latency

Control Design. It is well known that a constant input-output latency decreases the phase margin of the control system, and that it introduces a fundamental limitation on the achievable closed-loop performance. The resulting sampled-data system is time-invariant and of finite order, which allows standard linear time-invariant (LTI) analysis to be used (see e.g., [Åström and Wittenmark, 1997]). For a given value of the latency, it is easy to predict the performance degradation due to the delay. Furthermore, it is straightforward to account for a constant latency in most control design methods. From this perspective, a constant input-output latency is preferable over a varying latency.

Scheduling Design. The scheduling-induced input-output latency of a single control task can be reduced by assigning it a higher priority (or, alternatively, under EDF scheduling, a shorter deadline). This approach will of course not work for the whole task set.

Another option is to use non-preemptive scheduling. This will guarantee that, once the task has started its execution, it will continue uninterrupted until the end. The disadvantages of this approach are that the scheduling analysis for non-preemptive scheduling is quite complicated (e.g., [Klein *et al.*, 1993; Stankovic *et al.*, 1998]), and that the schedulability of other the tasks may be compromised.

A standard way to achieve a short input-output latency in a control task is to separate the algorithm calculations in two parts: *Calculate Output* and *Update State*. Calculate Output contains only the parts of the algorithm that make use of the current sample information. Update State contains the update of the controller states and precalculations for the next period. Update State can therefore be executed after the output signal transmission, hence, reducing the input-output latency. Further improvements can be obtained by scheduling the two parts as subtasks. This is the topic of the next chapter of the thesis.

Input-Output Jitter

Control Design. A control system with a time-varying input-output latency is quite difficult to analyze, since the standard tools for LTI systems cannot be used. If the statistical properties of the latency are known, then theory from jump linear systems can be used to evaluate the stability and performance of the system (in the mean sense), see [Nilsson, 1998a]. A similar approach is taken in the Jitterbug toolbox (see Chapter 6), where the latency is described by a random variable that is assumed to be independent from sample to sample.

Often, it is not possible to have exact knowledge of the input-output latency distribution. A simple, sufficient stability test for systems where only the range of the latency is known is given in [Lincoln, 2002b]. Assuming zero sampling jitter, the test can guarantee stability for *any* input-output latencies in a given interval (whether they are time-varying, dependent, etc.).

One approach to deal with jitter in the control design is to explicitly

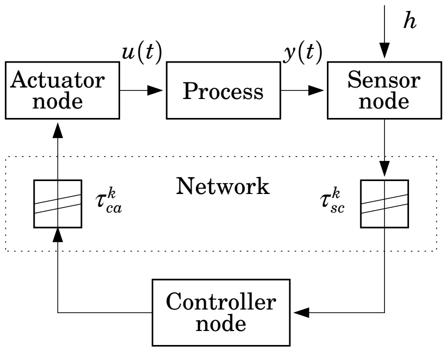


Figure 2.4 Distributed digital control system with network communication delays τ_k^{sc} and τ_k^{ca}. From [Nilsson, 1998a].

design the controller to be robust, i.e., the delay is treated as a parametric uncertainty. Many robust design methods can be used, such as H_∞, quantitative feedback theory (QFT) and μ-design.

Another approach is to let the controller actively compensate for the delay in each sample. An optimal, jitter-compensating controller was developed in [Nilsson, 1998a]. The controller compensates for time-varying delays in a control loop, which is closed over a communication network. The setup is shown in Figure 2.4. The sensor node samples the process periodically, sending the measurements over the network to the controller node. The controller node is event-driven and computes a new control signal as soon as a measurement arrives. The control signal is sent to the event-driven actuator node, which outputs the signal to the process. The LQ (linear-quadratic) state feedback control law has the form

$$u(k) = -L(\tau_{sc}^k) \begin{pmatrix} x(k) \\ u(k-1) \end{pmatrix}, \tag{2.13}$$

where the feedback gain L depends on the sensor-to-controller delay τ_{sc}^k in the current sample. The computation of the gain vector L is quite involved and requires that the probability distributions of τ_{sc} and τ_{ca} are known.

The above approach cannot be directly applied to scheduling-induced delays. The problem is that the delay in the current sample (i.e., the current input-output latency) will not be known until the task has finished (and by then it is too late to compensate). A simple scheme that compensates for delay in the previous sample is presented in [Lincoln, 2002a]. The compensator has the same basic structure as the well-known Smith predictor, but allows for a time-varying delay.

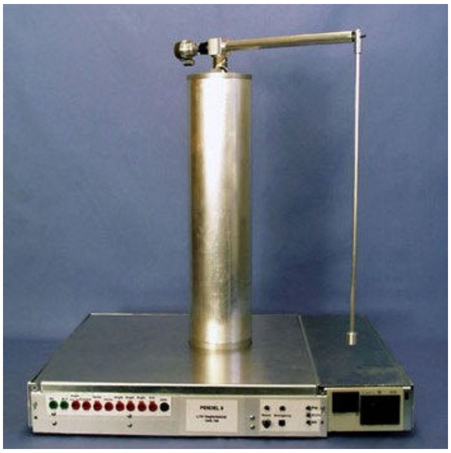


Figure 2.5 The inverted pendulum used in the Bluetooth network example. The pendulum is attached the end of a rotating arm.

Many other heuristic jitter compensation schemes have been suggested, e.g., [Hägglund, 1992; Albertos and Crespo, 1999; Marti *et al.*, 2001]. Yet another scheme (based on [Nilsson, 1998a]) is given in the following example.

EXAMPLE 2.1—INPUT-OUTPUT JITTER IN A BLUETOOTH NETWORK

Consider a distributed control system where a rotating inverted pendulum (see Figure 2.5) should be controlled over a Bluetooth network. The example is taken from [Eker *et al.*, 2001], where experiments on a laboratory setup were reported.

The objective of the control is to stabilize the pendulum in the upright position by applying a torque to the rotating arm. The control design is based on a linearized, fourth-order model of the process. The full state vector (the arm and pendulum angles and their derivatives) is measurable on the process.

The control system is configured according to the setup in Figure 2.4. Due to disturbances and retransmissions in the wireless Bluetooth network, long random delays are introduced in the control loop. The total network delay $\tau = \tau_{sc} + \tau_{sa}$ is assumed to have the probability distribution shown in Figure 2.6.

It can be noted that delay compensation in the distributed system works better the shorter the controller-to-actuator delay is compared to the sensor-to-controller delay. If the controller is located *at* the actuator, the delay in the current sample can be known exactly and compensated

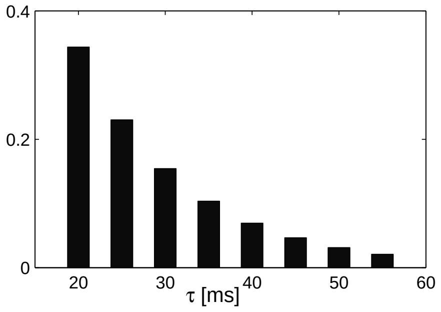


Figure 2.6 The input-output latency distribution in the Bluetooth example.

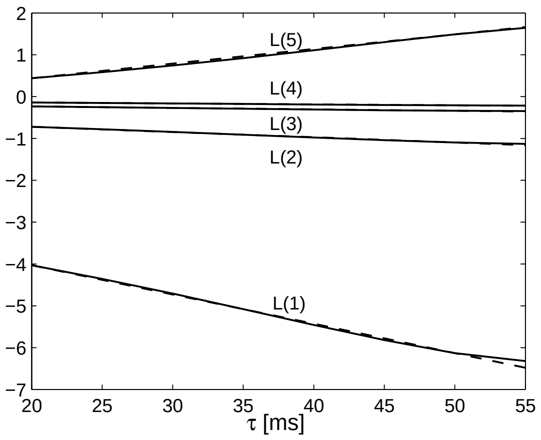


Figure 2.7 Optimal state feedback gains (full lines) and affine approximations (dashed lines).

for accordingly. Modifying the setup is of course not a real solution, but using an intelligent I/O node (containing both the sensor node and the actuator node) with only very limited computational resources, we will see that ideal case can be closely emulated.

Assuming a sampling interval of $h = 60$ ms, a state feedback controller on the form (2.13) is derived. The resulting optimal gain vector for different values of the total delay τ is shown in Figure 2.7. It is noted that the optimal gain vector can be closely approximated by an affine function of τ. A Taylor approximation of (2.13) around a nominal delay τ_0 gives

$$u(k) \approx -\left(L(\tau_0) + \frac{\partial}{\partial \tau}L(\tau_0)(\tau - \tau_0)\right)\begin{pmatrix} x(k) \\ u(k-1) \end{pmatrix}. \tag{2.14}$$

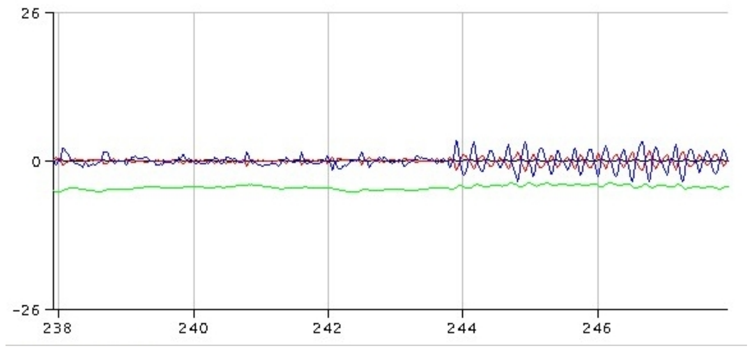


Figure 2.8 Experiment on the Bluetooth-pendulum setup with random communication delays. At $t = 244$, the jitter compensation is turned off, causing a large increase in the state variance.

In the controller node, $x(k)$ and $u(k-1)$ are known, but the value of τ is still unknown. However, the controller can precompute

$$\hat{u}(k) = -L_0 \begin{pmatrix} x(k) \\ u(k-1) \end{pmatrix}, \tag{2.15}$$

and

$$\alpha(k) = -\frac{\partial}{\partial \tau} L(\tau_0) \begin{pmatrix} x(k) \\ u(k-1) \end{pmatrix}. \tag{2.16}$$

These two scalars are then sent to the actuator node. Assuming that the I/O keeps track of the round-trip delay τ, it can do the simple adjustment

$$u(k) = \hat{u}(k) + \alpha(k)(\tau - \tau_0), \tag{2.17}$$

before applying the control signal to the process.

The compensation scheme was tested on a laboratory Bluetooth-pendulum setup. The random delays in the experiments were not due to actual retransmissions in the network but were injected in the control loop on purpose. The result of an experiment with and without jitter compensation in shown in Figure 2.8. It is seen that jitter compensation reduces the variance of the measured states considerably.

□

Scheduling Design. One way to minimize scheduling-induced input-output jitter is to introduce a dedicated, high-priority task (or interrupt handler) for the output operation. The task is released in such a way that the latency between the input and the output is always constant. The approach has been suggested in various settings, e.g., [Locke, 1992; Klein *et al.*, 1993; Halang, 1993]. Disadvantages of the approach include a more complex implementation and more run-time overhead. Also, reducing jitter means increasing the average input-output latency. In [Balbastre *et al.*, 2000], a design procedure that minimizes input-output jitter using high-priority input and output tasks is presented. Task attribute assignment under both FP and EDF scheduling is considered.

Another option to reduce input-output jitter is, again, to use non-preemptive scheduling. Given that the control algorithm has a constant execution time, this will make the input-output latency constant. The drawback is that the scheduling design becomes more complicated.

In [Baruah *et al.*, 1999a], minimization of output jitter under EDF scheduling is discussed. A list of properties that a "true" jitter minimization algorithm should fulfill is given: no new tasks should be introduced, outputs should not be suspended, etc. An algorithm is then suggested where the utilizations of some tasks are increased in order to reduce the jitter.

Sampling Jitter

Control Design. As a rule of thumb, relative variations of the sampling interval that are smaller than ten percent of the nominal sampling interval need not be compensated for. The simplest sampling jitter compensation methods are ad-hoc, but seem to work quite well, e.g., [Wittenmark and Åström, 1980; Albertos and Crespo, 1999; Marti *et al.*, 2001]. If the controller is based on a continuous-to-discrete approximation, the approximation can be carried out in each sample, based on the previous sampling interval. The technique is illustrated on a PD (proportional-derivative) controller below.

EXAMPLE 2.2—SAMPLING JITTER

Consider PD control of a DC servo. Let the servo be described by the continuous-time transfer function

$$G(s) = \frac{1000}{s(s+1)}. \tag{2.18}$$

A good discrete-time implementation of the PD controller, which includes filtering of the derivative part, is

$$\begin{aligned} P(k) &= K(r(k) - y(k)), \\ D(k) &= a_d D(k-1) + b_d(y(k-1) - y(k)), \\ u(k) &= P(k) + D(k), \end{aligned} \tag{2.19}$$

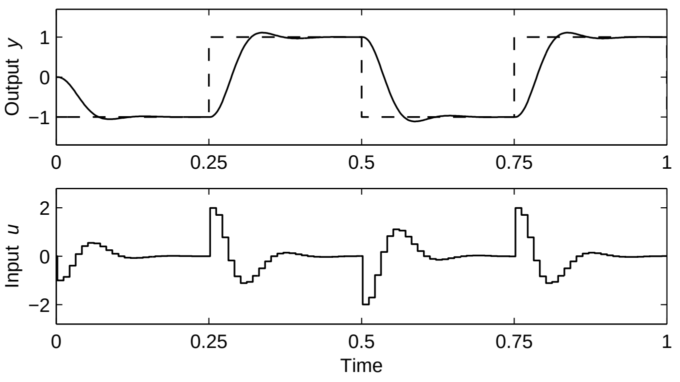


Figure 2.9 When no sampling jitter is present, the control performance is good.

where $a_d = \frac{T_d}{Nh+T_d}$ and $b_d = \frac{NKT_d}{Nh+T_d}$.

A sampling period of $h = 10$ ms is chosen, and the PD controller is tuned to give a fast and well-damped response to setpoint changes. The obtained parameters are $K = 1$, $T_d = 0.04$, and $N = 30$. The parameters a_d and b_d are normally precalculated, assuming that the sampling interval is constant.

A first simulation of the closed-loop system, where there is no sampling jitter, is shown in Figure 2.9. The controller behaves as expected, and the performance is good.

A second simulation, where the actual sampling interval varies randomly in the interval $h_{min} = 2$ ms to $h_{max} = 18$ ms, is shown Figure 2.10. The discrepancy between the nominal and the actual sampling interval causes the controller to repeatedly take either too small or too large actions. The resulting performance is quite poor. This is especially visible in the control signal.

Finally, in a third simulation, the controller is redesigned to compensate for the varying sampling interval. This is done by measuring the actual length of the sampling interval and recalculating the parameters a_d and b_d accordingly. Figure 2.11 shows that this version of the controller handles the sampling jitter well. The performance is almost as good as in Figure 2.9.

$\square$

The technique of sampling interval modification must be altered for higher order controllers, see [Wittenmark and Åström, 1980].

Another approach to sampling jitter compensation is to introduce a time-varying Kalman filter. The optimal time-varying Kalman filter is however quite computationally demanding, see [Nilsson, 1998a].

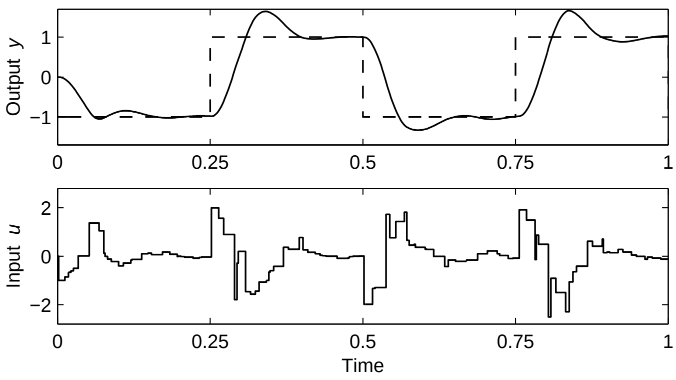


Figure 2.10 Sampling jitter causes the control performance to degrade.

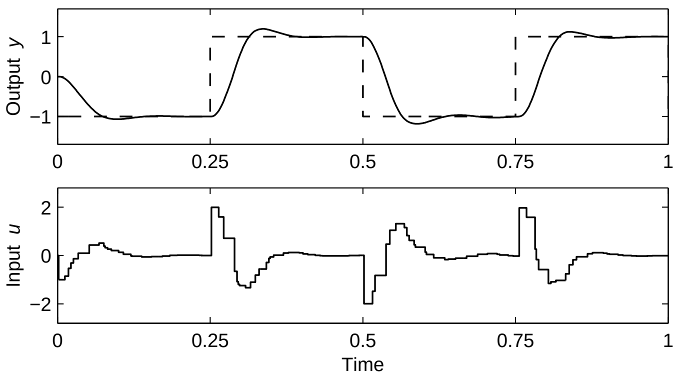


Figure 2.11 When compensating for the sampling jitter, the control performance is good again.

2.4 Control and Scheduling Codesign

In recent years, a number of research efforts have been devoted to control and scheduling codesign of real-time control systems. One of the very first papers that addressed the issue of joint controller and scheduler design was [Seto *et al.*, 1996]. The paper considers optimal sampling period selection for a set of digital controllers. The performance of each controller is described by a cost function (a performance index), which is approximated by an exponential function of the sampling frequency. An optimization problem is formulated, where the sum of the cost functions should be minimized subject to the schedulability constraint. Both fixed-priority and earliest-deadline-first scheduling is considered.

The previous paper does not consider the input-output latency in the controllers. An approach to joint optimization of sampling period and input-output latency subject to performance specifications and schedulability constraints is presented in [Ryu *et al.*, 1997; Ryu and Hong, 1998]. The control performance is specified in terms of steady state error, overshoot, rise time, and settling time. These performance parameters are expressed as functions of the sampling period and the input-output latency. The goal of the optimization is to minimize the utilization of the task set. A heuristic iterative algorithm is proposed for the optimization of these parameters subject to schedulability constraints. The algorithm is based on the period calibration method (PCM) [Gerber *et al.*, 1995] for determining the task attributes.

Optimal sampling period selection for a set of linear-quadratic (LQ) controllers is treated in [Eker *et al.*, 2000]. Similar to [Seto *et al.*, 1996], an optimization problem is formulated where the total performance of the controllers should be optimized subject to a schedulability constant. Analytical expressions for optimal LQ cost as a function of the sampling interval are derived.

The integration of static cyclic scheduling and optimal (LQ) control is the topic of [Rehbinder and Sanfridson, 2000]. A number of processes should be controlled by a set of state-feedback controllers. The performance of each controller is measured by a quadratic cost function. A combinatorial optimization problem is formulated that, given a schedule length, attempts to minimize the sum of the control costs. The solution contains the periodic task schedule as well as the state feedback gains of the controllers. A similar codesign problem is formulated in [Lincoln and Bernhardsson, 2000]. There, the optimal switching sequence (i.e., schedule) for a set of controllers should be found. Unlike the previous paper, however, the obtained schedule is not necessarily periodic.

In [Palopoli *et al.*, 2002], sampling periods for a set of controllers should be chosen such that a certain robustness measure is maximized. An optimization procedure is used to derive the gains of the controllers together with the sampling periods, subject to a schedulability constraint. The method is limited in its applicability in that it only handles first-order plants.

A codesign method that assumes run-time scheduling decisions based on current plant state information is presented in [Zhao and Zheng, 1999]. There, the controller associated with the plant with the largest current control error is chosen for execution, while the other plants are fed zeros. An algorithm is given that determines the (static) state feedback laws to meet a given performance specification.

3

Subtask Scheduling

3.1 Introduction

In the introductory chapter, it was shown that a naive implementation of multiple controllers on the same platform can introduce large amounts of latency and jitter in the control loops. Listing 3.1 shows a straightforward implementation of a periodic control task. Each period, the input is read, the control algorithm is executed, and the output is written. In the best case, the controller has been designed to compensate for the computational delay in the control algorithm. When executing in the real-time system, additional, time-varying latency will be introduced by the task scheduling. Taken together, the latency and jitter degrade the control performance. In this chapter, we will show how the input-output latency can be reduced by more detailed scheduling analysis.

It should be pointed out that input-output latency reduction is of out-

Listing 3.1 Naive implementation of a control loop. (The listings in this chapter are written in Modula-2 pseudo code.)

```
t := CurrentTime;
LOOP
  ReadInput;
  ControlAlgorithm;
  WriteOutput;
  t := t + h;
  SleepUntil(t);
END;
```

most importance. Even if the latency is fixed and known, delay compensation can only recover part of the performance loss. This fact is illustrated by the following example:

EXAMPLE 3.1—MINIMUM-VARIANCE CONTROL OF AN INTEGRATOR
Consider minimum-variance control an integrator,

$$\frac{dx(t)}{dt} = u(t) + v_c(t), \tag{3.1}$$

where x is the state, u is the control signal, and v_c is a continuous-time white noise process with zero mean and unit variance. A discrete-time controller is designed to minimize the continuous-time cost function

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T x^2(t)\,dt. \tag{3.2}$$

Assuming an input-output latency of L, the cost for the optimal, delay-compensating controller is

$$J = \frac{3+\sqrt{3}}{6}h + L \ (\approx 0.79h + L). \tag{3.3}$$

(For details, see Appendix.) Note that the cost is very much affected by the input-output latency. In this case, a one-sample latency ($L = h$) more than doubles the value of the cost function compared to a controller with zero latency. To put it another way, a controller with a one-sample latency needs to execute more than twice as often as a controller with zero latency to achieve the same performance. □

Division into Calculate Output and Update State

A standard way to reduce the latency in a controller is to split the control algorithm into two parts: Calculate Output and Update State (e.g., [Åström and Wittenmark, 1997]), see Listing 3.2. Calculate Output contains only the operations immediately necessary to produce a control signal, while the rest of the calculations are postponed to Update State. Consider for instance the implementation of a general linear controller written in state-space form:

$$\begin{aligned} x(k+1) &= \Phi x(k) + \Gamma y(k), \\ u(k) &= Cx(k) + Dy(k). \end{aligned} \tag{3.4}$$

Here, only the calculation of $Dy(k)$ (a scalar operation in the single-input, single-output case) and one addition needs to be performed in Calculate

Listing 3.2 Implementation where the control algorithm has been divided into two parts, Calculate Output and Update State.

```
t := CurrentTime;
LOOP
  ReadInput;
  CalculateOutput;
  WriteOutput;
  UpdateState;
  t := t + h;
  SleepUntil(t);
END;
```

Output; the remaining matrix calculations can be done in Update State. Another example is an adaptive controller where the recursive parameter estimation and controller design can carried out in the Update State part.

In a real implementation, the control algorithm itself only makes up a small portion of the code executed in each sample. There are often a number of other functions that need to be performed, such as reading of user commands, reference trajectory generation, safety checks, logging of data, plotting, etc. This implies that the Calculate Output part can be made quite small compared to the Update State part in many controllers.

Related Work

Subtask scheduling of (for instance) controllers is considered in [Gerber and Hong, 1993; Gerber and Hong, 1997]. There, the objective is not to achieve better control performance but to enhance the schedulability of the task set under fixed-priority (FP) scheduling. An automatic program slicing procedure is suggested where tasks are decomposed into subtasks based on program dependence graphs.

A schedulability motive is also given in [Burns *et al.*, 1994], which deals with FP scheduling of tasks that consists of two parts: one that produces an output, and one that does not. Only the first part needs to complete before the deadline. The analysis is not applicable to controllers, however, since the Update State part of a control algorithm does produce an output (i.e., the new state).

In [Crespo *et al.*, 1999; Albertos *et al.*, 2000; Balbastre *et al.*, 2000], subtask scheduling of control tasks is considered. Each task is decomposed into three parts with different priorities: the input operation (medium priority), the control computation (low priority), and the output operation (high priority). The goal of the scheduling design is to minimize the input-output jitter (here termed “the control action interval”). As a side effect, the average input-output latency increases (despite the somewhat misleading title of [Balbastre *et al.*, 2000]).

3.2 Task Models

In this section, we will consider different task models for periodic control loops. Both fixed-priority (FP) and earliest-deadline-first (EDF) scheduling will be treated. Let each control task τ consist of the two subtasks τ_{CO} (Calculate Output) and τ_{US} (Update State). The worst-case execution time of the subtasks are assumed to be known and equal to C_{CO} and C_{US} respectively. For simplicity, it will be assumed that the execution time for reading inputs and writing outputs can be neglected, or, alternatively, be included in C_{CO}.

Priority-Constrained Scheduling

The first model assumes that the subtasks are executed in sequence using a priority constraint, see Figure 3.1. In the model, both subtasks are released at the beginning of the period. To guarantee the correct execution order, τ_{CO} must have higher priority (dynamic or static, depending on the scheduling policy) than τ_{US}. This kind of priority constraint is natural to enforce, since τ_{CO} is the time-critical part of the control algorithm. The model is simple to analyze under FP scheduling, since the subtasks are released at the same instant (the critical instant in the response-time analysis).

Offset Scheduling

The second model assumes that τ_{US} is released with a fixed *offset* ϕ_{US} compared to the release of τ_{CO}, see Figure 3.2. The priorities and the offset must be chosen such that τ_{CO} is guaranteed to have finished its execution

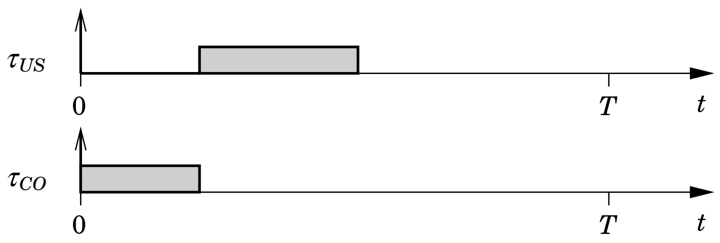


Figure 3.1 The controller subtasks are scheduled using a priority constraint.

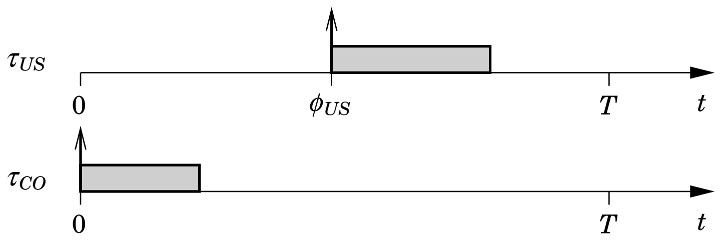


Figure 3.2 The controller subtasks are scheduled using an offset.

before τ_{US} starts. The introduction of the offset can potentially increase the schedulability of the task set, since the subtasks no longer share a critical instant. The offsets makes the schedulability analysis under FP is more difficult [Audsley *et al.*, 1993; Redell and Törngren, 2002]; but, as we will see, it enables quite simple analysis under EDF scheduling.

Other Task Models

There are of course many other possible task models. One option would be to release τ_{US} immediately after τ_{CO} has finished. From a schedulability point of view, this model would differ from the priority-constrained model only when τ_{US} has higher priority than τ_{CO}, which seems counter-intuitive. Also, because of the dynamic release offset, the analysis becomes more difficult, see [Harbour *et al.*, 1994; Gutierrez and Harbour, 1998].

Another option for more detailed analysis is to also treat the input and output actions as subtasks [Crespo *et al.*, 1999]. This gives better control of the sampling jitter and the input-output jitter. At the same time, the scheduling analysis becomes more complicated.

3.3 Deadline Assignment

In the models presented above, a number of task attributes need to be assigned. In the priority-constrained case, relative deadlines must be assigned to all Calculate Output parts. Runability constraints dictate that the deadline is chosen somewhere in the interval

$$C_{CO} \leq D_{CO} \leq T - C_{US}. \tag{3.5}$$

The deadline of the Update State part is always equal to the period,

$$D_{US} = T. \tag{3.6}$$

In the offset case, more attributes must be assigned. First, D_{CO} must be selected according to (3.5). The offset ϕ_{US} should be assigned somewhere in the interval

$$D_{CO} \leq \phi_{US} \leq T - C_{US}. \tag{3.7}$$

The relative deadline of Update State follows according to

$$D_{US} = T - \phi_{US}. \tag{3.8}$$

Deadline Assignment under FP Scheduling

Both models above would in theory be applicable under both FP and EDF scheduling. However, from a designer's perspective, we would like to use the simplest schedulability analysis possible. For this reason, we will assume the priority-constrained model under FP scheduling and the offset model under EDF scheduling.

The ideal case under FP scheduling is when all Calculate Output parts have higher priorities than all Update State parts. Unfortunately, such a priority assignment might render the task set unschedulable. In cases where this approach does not work, the following iterative algorithm can be used. Given a schedulable original task set, the algorithm attempts to minimize the deadlines of the Calculate Output parts while maintaining schedulability.

ALGORITHM 3.1

1. Start by assigning $D_{US_i} := T_i$ and $D_{CO_i} := T_i - C_{US_i}$.
2. Assign deadline-monotonic (DM) priorities P_i to all subtasks.
3. Calculate the worst-case response times, R_{CO_i}, of the Calculate Output parts (see Section 2.2).
4. Assign new deadlines to the Calculate Output parts according to $D_{CO_i} := R_{CO_i}$.
5. Repeat from 2 until no further improvement in response times is obtained.

□

Some properties of the algorithm are formalized in the following theorem:

THEOREM 3.1

Given a set a control tasks that is rate-monotonic (RM) schedulable, Algorithm 3.1 preserves schedulability in each iteration. Furthermore, the algorithm terminates in a finite number of steps.

Proof. Schedulability in the first iteration: Since the original task set is RM schedulable, each task must finish before its deadline $D_i = T_i$. This implies that each Calculate Output parts must finish at least before $T_i - C_{US_i}$ (otherwise the Update State part would not have time to execute). Since the subtasks are schedulable under the RM deadline assignment (i.e., both subtasks execute in sequence and with the same priority), they must also be schedulable under the deadline-monotonic (DM) priority assignment, since that policy is optimal.

Schedulability in the following iterations: After assigning new deadlines, the subtasks are obviously still schedulable with the old priorities (in fact, we will have $R_{CO_i} = D_{CO_i}$). The subtasks will be schedulable also after the DM priority assignment since, again, that policy is optimal.

Termination: After each iteration, the relative deadline of a Calculate Output part can either decrease or remain the same. The relative priority ordering of the Calculate Output parts will never change, since a lower priority always implies a longer response time. Hence, the priority of a Calculate Output part can only increase, and the priority of an Update State part can only decrease. Since there are a finite number of priority orderings, the priorities (and hence also the deadlines and the response times) must converge after a finite number of steps. □

Deadline Assignment under EDF

The priority-constrained model and the deadline assignment algorithm given above would also work under EDF scheduling. Unfortunately, response-time analysis is quite difficult under EDF. A simpler option is to use the offset model. With this model, deadlines can be assigned to the subtasks such that the processor demand in each interval is equal to the utilization of the original task. The deadlines and the offsets are assigned according to

$$D_{CO_i} := \frac{C_{CO_i}}{C_{CO_i} + C_{US_i}} T_i, \tag{3.9}$$

and

$$D_{US_i} := \frac{C_{US_i}}{C_{CO_i} + C_{US_i}} T_i, \tag{3.10}$$

and the offset is chosen as $\phi_{US_i} = D_{CO_i}$.

Schedulability of the task set can be checked using the processor demand approach (see Section 2.2). In the interval $[0, \phi_{US_i}]$ the loading factor of the task is

$$\frac{C_{CO_i}}{\phi_{US_i}} = \frac{C_{CO_i} + C_{US_i}}{T_i} = \frac{C_i}{T_i}, \tag{3.11}$$

and in the interval $[\phi_{US_i}, T_i]$ the loading factor is

$$\frac{C_{US_i}}{T_i - \phi_{US_i}} = \frac{C_{CO_i} + C_{US_i}}{T_i} = \frac{C_i}{T_i}. \tag{3.12}$$

The loading factor (or utilization) of the original task was $U_i = \frac{C_i}{T_i}$. The subtasks are thus schedulable if and only if the original task was schedulable.

3.4 Latency Analysis

Once the subtask attribute assignment has been performed, response-time analysis can be applied to verify that the latencies in the control loops have been reduced.

Latency Analysis under FP scheduling

Under fixed-priority scheduling, it is straightforward to derive bounds for the sampling latency and the input-output latency.

Sampling Latency. Assuming that the sampling operation takes place at the very start of the Calculate Output part, the maximum sampling latency can be found by computing the worst-case response time of a hypothetical sampling task with a very short execution time and the same priority as the original Calculate Output part. According to standard response-time analysis (see Section 2.2), the maximum sampling latency for task i is given by the smallest number $L_{s_i}^{max} > 0$ that fulfills

$$L_{s_i}^{max} = \sum_{j \in hp(i)} \left\lceil \frac{L_{s_i}^{max}}{T_j} \right\rceil C_j. \tag{3.13}$$

Here, $hp(i)$ is the set of all subtasks or tasks with higher priority than the task under investigation.

Input-Output Latency. The maximum input-output latency occurs when the task is released just prior to all other tasks in the system. The task has time to sample the plant but is then immediately preempted by all higher-priority tasks. The maximum input-output latency of a controller is hence equal to the worst-case response time of the Calculate Output part:

$$L_{io_i}^{max} = C_{CO_i} + \sum_{j \in hp(i)} \left\lceil \frac{L_{io_i}^{max}}{T_j} \right\rceil C_j. \tag{3.14}$$

Similarly, the minimum input-output latency of a controller is equal to the best-case response time of the Calculate Output part. In the best case, the start of the task (i.e., the sampling operation) coincides with the release of the task [Redell and Sanfridson, 2002]. According to the best-case response time analysis (see Section 2.2), the minimum input-output latency is thus given by

$$L_{io_i}^{min} = C_{CO_i}^b + \sum_{j \in hp(i)} \left\lceil \frac{L_{io_i}^{min} - T_j}{T_j} \right\rceil C_j^b, \tag{3.15}$$

where C_i^b denotes the best-case execution time of task τ_i.

Latency Analysis under EDF scheduling

Latency analysis under EDF can be performed using similar techniques as those above. There are, however, no known formulas for the *best-case* response times under EDF scheduling. An alternative to analysis is task simulation.

Regarding the input-output latency, an interesting observation can be made when comparing ordinary rate-monotonic scheduling to earliest-deadline-first scheduling (assuming $D_i = T_i$ for all tasks):

THEOREM 3.2
Consider a set of control tasks implemented according to Listing 3.1 or Listing 3.2. For each task, the maximum input-output latency under EDF scheduling is shorter than or equal to the corresponding maximum latency under RM scheduling.

Proof. Under RM scheduling, the maximum latency is given by (3.14). Now consider EDF scheduling. Assume that task τ_i is released and then, possibly after some preemption, starts its execution. Once the task has started, it can only be preempted by tasks with a shorter period. (Tasks with a longer period that arrive after τ_i has started will have a later deadline than what τ_i has.) Furthermore, τ_i can only be preempted by task instances that have an earlier deadline than what τ_i has. This gives the following upper bound on the input-output latency:

$$L_{io_i}^{max} = C_{CO_i} + \sum_{j \in sp(i)} \left\lceil \frac{\min\left(L_{io_i}^{max},\, T_i - T_j\right)}{T_j} \right\rceil C_j. \tag{3.16}$$

Here, $sp(i)$ is the set of tasks with shorter period than the task under investigation. But this set is identical to the set $hp(i)$ under rate-monotonic scheduling. Because of the min$(\cdot)$ operation, it follows that $L_{io_i}^{max}$ in (3.16) is less than or equal to $L_{io_i}^{max}$ in (3.14). $\square$

3.5 Implementation

In the analysis, we have modeled Calculate Output and Update State as two separate tasks. They do not necessarily have to be implemented as such, however. If the real-time operating system supports dynamic changes of priorities, only simple modifications of the code in Listing 3.2 is needed to produce the desired result. This makes for a more efficient implementation with fewer tasks and less need for inter-task communication.

Implementation under FP Scheduling

Under FP scheduling, we simply insert `SetPriority` commands in the code when entering a new subtask, see Listing 3.3. Note that the priority changes may introduce additional context switches, and this might degrade the performance in a real system.

Implementation under EDF Scheduling

Under EDF scheduling, we must both insert `SetAbsDeadline` commands and ensure that the Update State part does not start until the correct offset. This is done by inserting an extra `SleepUntil` statement in the code, see Listing 3.4.

Remark: Ideally, the EDF real-time kernel should provide a primitive called `SetAbsDeadlineAndSleepUntil` in order to minimize the number of unnecessary context switches.

Listing 3.3 Implementation of subtask scheduling under fixed-priority scheduling.

```
t := CurrentTime;
SetPriority(P_CO);
LOOP
  ReadInput;
  CalculateOutput;
  WriteOutput;
  SetPriority(P_US);
  UpdateState;
  t := t + h;
  SetPriority(P_CO);
  SleepUntil(t);
END;
```

Listing 3.4 Implementation of subtask scheduling under earliest-deadline-first scheduling.

```
t := CurrentTime;
SetAbsDeadline(t + D_CO);
LOOP
  ReadInput;
  CalculateOutput;
  WriteOutput;
  SetAbsDeadline(t + h);
  SleepUntil(t + D_CO);
  UpdateState;
  t := t + h;
  SetAbsDeadline(t + D_CO);
  SleepUntil(t);
END;
```

Jitter Reduction

In the subtask analysis above, we did not specifically deal with jitter minimization. Since the jitter depends on the maximum latencies, Eqs. (2.8) and (2.12), it is often reduced as a side effect of the subtask scheduling (see the example below). It is possible to reduce the jitter further by raising the priority temporarily when performing the input and the output. Listing 3.5 shows an example of how this can be done under FP scheduling. Note that this scheme increases the average input-output latency of the controller.

Assuming that the `ReadInput` and `WriteOutput` operations takes zero time, the subtask analysis is still valid. However, the priority changes will introduce even more context switches, and, again, this might degrade the performance in the real system.

3.6 An Example

As an example, we will revisit the inverted pendulum control problem introduced in Section 1.4. The control performance under various scheduling policies will be evaluated by schedule simulations using TrueTime (see Chapter 7) followed by cost computations using Jitterbug (see Chapter 6).

Listing 3.5 Implementation of subtask scheduling with jitter reduction under fixed-priority scheduling.

```
t := CurrentTime;
SetPriority(HIGH);
LOOP
  ReadInput;
  SetPriority(P_CO);
  CalculateOutput;
  SetPriority(HIGH);
  SleepUntil(t + D_CO);
  WriteOutput;
  SetPriority(P_US);
  UpdateState;
  t := t + h;
  SetPriority(HIGH);
  SleepUntil(t);
END;
```

Controller Design

Let each inverted pendulum be described by a linear transfer function,

$$G(s) = \frac{\omega_0^2}{s^2 - \omega_0^2}, \tag{3.17}$$

where ω_0 is the natural frequency of the pendulum. The different pendulum lengths, $l = 0.1, 0.2, 0.3$ m, correspond to different natural frequencies, $\omega_0 = 9.9, 7.0, 5.7$ rad/s. It is assumed that the process is disturbed by a continuous-time white noise process with unit variance, and that the measurements are disturbed by discrete-time noise with unit variance. An LQG controller is designed to minimize the continuous-time cost function

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt, \tag{3.18}$$

where the weighting matrix Q is chosen as

$$Q = \begin{pmatrix} 1 & 0 \\ 0 & 0.002 \end{pmatrix}.$$

The controller is designed (using the Jitterbug command `lqgdesign`) assuming a constant input-output latency equal to the minimum input-output latency. The cost function (3.18) will also be used to evaluate the performance of the controllers.

Table 3.1 First iteration of the deadline assignment algorithm.

	T	D	C	P	R
τ_{CO1}	20	17	3	6	3
τ_{US1}	20	20	4	5	7
τ_{CO2}	29	26	3	4	10
τ_{US2}	29	29	4	3	14
τ_{CO3}	35	32	3	2	17
τ_{US3}	35	35	4	1	28

Table 3.2 Third iteration of the deadline assignment algorithm.

	T	D	C	P	R
τ_{CO1}	20	3	3	6	3
τ_{US1}	20	20	4	3	13
τ_{CO2}	29	6	3	5	6
τ_{US2}	29	29	4	2	17
τ_{CO3}	35	9	3	4	9
τ_{US3}	35	35	4	1	28

Deadline Assignment

It is assumed that the execution time of the control algorithm is 7 ms. The Calculate Output part takes 3 ms, and the Update State part takes 4 ms.

Fixed-Priority Scheduling Under FP scheduling, Algorithm 2.1 is used for assigning deadlines to the Calculate Output parts. In the first iteration of the algorithm, we have the task set shown in Table 3.1. In the table, P denotes the priority of a subtask and R is the worst-case response time (calculated according to (2.2)). After two more iterations, the values have converged, see Table 3.2. In this case, the end result is that all Calculate Output parts have higher priority than all the Update State parts.

Earliest-Deadline-First Scheduling Under EDF scheduling, the subtasks are assigned relative deadlines according (3.9) and (3.10).

Table 3.3 Summary of scheduling-induced latencies. (Compare the values for Task 3 under FP scheduling (naive implementation and subtask scheduling) with the histograms in Figures 1.6 and 1.11.)

	Task 1			Task 2			Task 3		
	L_s^{max}	L_{io}^{min}	L_{io}^{max}	L_s^{max}	L_{io}^{min}	L_{io}^{max}	L_s^{max}	L_{io}^{min}	L_{io}^{max}
Ideal	0	0	0	0	0	0	0	0	0
Sample delay	0	20	20	0	29	29	0	35	35
Naive, FP	0	7	7	7	7	14	14	7	28
Naive, EDF	1	7	7	7	7	14	14	7	21
Subtask, FP	0	3	3	3	3	6	6	3	9
Subtask, EDF	0	3	3	3.6	3	6	6.4	3	10

Table 3.4 Summary of costs.

	J_1	J_2	J_3	$\sum J$
Ideal	1.00	1.00	1.00	3.00
Sample delay	2.28	2.20	1.75	6.23
Naive, FP	1.36	1.45	1.75	4.56
Naive, EDF	1.36	1.47	1.58	4.41
Subtask, FP	1.14	1.11	1.10	3.36
Subtask, EDF	1.14	1.13	1.11	3.38

Results

For each scheduling scenario below, the performance has been evaluated in two steps. First, the scheduling algorithm has been simulated in True-Time to find the distributions of the sampling latency, L_s, and the input-output latency, L_{io} for each controller. Then, these distributions have been used in Jitterbug to calculate the control cost (3.18). The maximum and minimum latencies in the different cases have been summarized in Table 3.3 and the resulting costs are given in Table 3.4.

Ideal Case. The ideal case is used as a reference in the performance evaluation. This is the performance resulting from executing the controllers in an infinitely fast computer. The noise acting on each process has been scaled such that the cost for each controller is unity.

Sample delay. In this implementation, there is a constant input-output latency of one sample for each controller. The input and output actions are assumed to be performed in hardware, thus eliminating the jitter. Despite the exact delay compensation in the control design, the resulting performance is poor.

Naive Implementation. In the naive implementation, the entire algorithm is computed before the output is written. This introduces a minimum input-output latency of 7 ms in all loops. The performance is comparable under FP and EDF scheduling, but slightly better for Task 3 under EDF scheduling. This is probably due to the shorter maximum input-output latency of Task 3 under EDF scheduling (compare with Theorem 3.2).

Subtask Scheduling. Dividing the algorithm into Calculate Output and Update state, the minimum input-output latency is decreased from 7 to 3 ms. Under both FP and EDF scheduling, the maximum input-output latencies have been significantly reduced. The resulting control performance is quite close to the ideal case.

3.7 Conclusion

It has been shown how the performance of digital controllers can be improved by more detailed timing analysis. By scheduling the main parts of a control algorithm as subtasks, it is possible to reduce the latencies in a set of control loops. The scheduling technique is simple to implement in ordinary real-time operating systems.

Subtask scheduling under EDF as suggested in this chapter is one of the ideas behind the Control Server (see Chapter 5).

One drawback with subtask scheduling is that more context switches are introduced (see Figure 1.10). In the example above, the potentially negative effects of this were neglected. In Chapter 7, a more detailed simulation is performed where the overhead due to context switches is accounted for.

4

Feedback Scheduling

4.1 Introduction

In some control applications, the computational workload can change dramatically over time. As an example, consider an unmanned vehicle that should operate in different types of terrain. Depending on the environment, different algorithms might be needed for planning, navigation, and steering. The execution time of the individual tasks may also be highly varying.

In systems where the workload displays large fluctuations, a traditional real-time design, based on worst-case assumptions, may be infeasible. Assuming that all tasks execute at their maximum rates and according to their worst-case execution times (WCETs), the developer may be forced to choose a very powerful CPU, which will be under-utilized most of the time. On the other hand, basing the design on average-case conditions can lead to temporary CPU overloads that degrade the control performance.

Another problem with traditional real-time design is the assumption that the WCETs of the tasks are known. In practice, WCET estimation is very difficult. An under-estimation of the WCET might lead to CPU overloads, while an over-estimation will mean that resources are wasted.

In this chapter, we present a feedback scheduler for control tasks that attempts to keep the CPU utilization at a high level, avoid overload, and distribute the available computing resources among the control tasks. While we want to keep the number of missed deadlines as low as possible, control performance is our primary objective. Control tasks, thus, in our view, fall in a category somewhere between hard and soft real-time tasks. The known-WCET assumption is relaxed by the use of feedback from execution-time measurements. This assumes that the operating system supports measurements of task execution times. We also introduce feedforward to further improve the regulation of the utilization.

Two simulation case studies are presented. First, we study a set of hybrid controllers that switch mode during setpoint transitions. The computational resources are distributed among the tasks in an ad-hoc manner. In the second case study we consider scheduling of linear controllers that can be turned on and off. The CPU time is divided among the controllers in order to maximize a global cost function.

Related Work

In recent years, much research has been devoted to quality-of-service (QoS) aware real-time software, where the resource allocation in a system is adjusted on-line in order to maximize the performance in some respect. In [Li and Nahrstedt, 1998] a general framework is proposed for controlling the application requests for system resources using the amount of allocated resources for feedback. It is shown that a PID controller can be used to bound the resource usage in a stable and fair way. In [Abeni and Buttazzo, 1999], task models suitable for multimedia applications are defined. Two of these use PI control feedback to adjust the reserved fraction of CPU bandwidth. The resource allocation scheme Q-RAM is presented in [Rajkumar *et al.*, 1997]. Several tasks are competing for finite resources, and each task is associated with a utility value, which is a function of the assigned resources. The system distributes the resources between the tasks to maximize the total utility of the system. In [Abdelzaher *et al.*, 2000] a QoS renegotiation scheme is proposed as a way to allow graceful degradation in cases of overload, failures or violation of pre-runtime assumptions. The mechanism permits clients to express, in their service requests, a range of QoS levels they can accept from the provider, and the perceived utility of receiving service at each of these levels. The approach is demonstrated in an automated flight-control system.

Many scheduling techniques that allow QoS adaptation have been developed. An interesting mechanism for workload adjustments is given in [Buttazzo *et al.*, 1998], where an elastic task model for periodic tasks is presented. The relative sensitivities of tasks to period rescaling are expressed in terms of elasticity coefficients. Schedulability analysis of the system under EDF scheduling is given. In [Gill *et al.*, 1998], a mixed static/dynamic-priority scheduling approach for avionics systems is presented. Each task is associated with a criticality parameter. In overload situations, tasks at the highest criticality level are allowed to execute before other tasks. Similar ideas are used within the broader area of *value-based scheduling*, e.g., [Burns *et al.*, 2000].

The idea of using feedback in scheduling has to some extent been previously used in general purpose operating systems, in the form of multi-level feedback queue scheduling [Kleinrock, 1970; Blevins and Ramamoorthy, 1976; Potier *et al.*, 1976]. However, this has mostly been done in an ad-hoc way. A more control-theoretical approach is taken in [Stankovic *et al.*, 1999; Lu *et al.*, 1999] that present a scheduling algorithm called Feedback Control EDF (FC-EDF). A PID controller regulates the deadline miss-ratio for a set of soft real-time tasks with varying execution times, by adjusting their CPU utilization. It is assumed that tasks can change their CPU consumption by executing different versions of the same algorithm. An admission controller is used to accommodate larger changes in the workload. In [Lu *et al.*, 2000] the approach is extended. An additional PID controller is added that instead controls the CPU utilization. The two controllers are combined using a min-approach. The resulting hybrid controller scheme, named FC-EDF^2, gives good performance both during steady-state and under transient conditions. The framework is further generalized in [Lu *et al.*, 2002], where the feedback scheduler is broken down in three parts: the monitor that measures the miss ratio and/or the utilization, the control algorithm, and the QoS actuator that contains a QoS optimization algorithm to maximize the system value. In [Abeni *et al.*, 2002], the problem of dynamically assigning bandwidths to a set of constant bandwidth servers [Abeni and Buttazzo, 1998] is analyzed. The servers are modeled as discrete switched systems, and a feedback scheduler that adjusts the server bandwidths is derived using hybrid control theory.

On-line adjustment of sampling periods in order to avoid CPU overloads is the topic of [Beccari *et al.*, 1999]. A number of different rate-modulation algorithms suitable for rate-monotonic scheduling are given. A more control-theoretic approach to rate modulation is taken in [Shin and Meissner, 1999], where each controller is associated with a cost function. During overloads, tasks may migrate between different CPUs in a multiprocessor in order to maximize the overall control performance.

Adjustment of sampling rates is not the only way to achieve graceful degradation in overloaded real-time control systems. Controllers that are based on *anytime algorithms* allow a direct trade-off between the consumed CPU time and the quality of control. A good example is model-predictive control (MPC), where the control signal is found by solving an optimization problem in each sample. In the case of an overload, the optimization may be terminated early and still produce acceptable results. In [Henriksson *et al.*, 2002], scheduling of multiple MPC controllers is considered. Scheduling of anytime algorithms in avionics applications is discussed in [Agrawal *et al.*, 2003], where the resources are dynamically distributed among the tasks in order to maximize the global QoS.

Part of the work presented in this chapter is a continuation of the work in [Eker *et al.*, 2000], where a feedback scheduler for the special

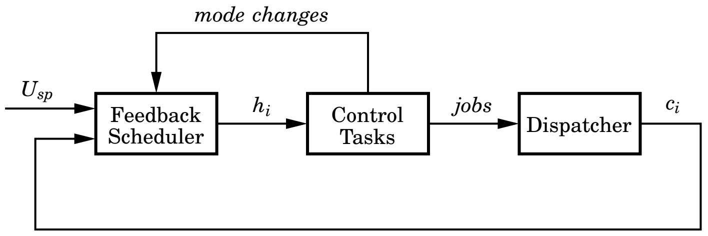


Figure 4.1 Block diagram of the proposed feedback scheduling structure. The scheduler controls the CPU utilization to a given setpoint, based on feedback from the task execution times. The control is exercised through adjustment of the controller sampling periods. Feedforward is used to notify the scheduler about controller mode changes.

case of linear-quadratic (LQ) controllers was presented. Formulas for the LQ cost function and its derivative with respect to the sampling period were given. The resource distribution problem was formulated as a recursive optimization problem based on the exact formulas for the derivatives of the cost functions. Due to the high computational costs involved, an approximate version of the scheduler was also developed. The cost functions were approximated by quadratic functions of the sampling periods, and explicit expressions for the optimal sampling periods were derived.

4.2 A Feedback Scheduling Architecture

In this section, a feedback scheduling architecture for control tasks is developed. It is assumed that each controller can operate in different modes that require different amounts of computations. In each mode, the controller has a *nominal sampling period*, denoted $h_{nom\,i}$. The computation time within each mode may vary randomly due to data dependencies, hardware effects, etc. It is assumed that the execution time of each task instance (job) can be measured by the real-time kernel.

The proposed structure of the feedback scheduler is shown in Figure 4.1. The control tasks generate jobs that are fed to a run-time dispatcher. The scheduler receives feedback information about the actual execution time, c_i, of each job. It also receives feedforward information from control tasks that are about to switch mode. In this way, the scheduler can pro-act rather than react to sudden changes in the workload. The scheduler attempts to keep the CPU utilization, U, as close as possible to a utilization setpoint, U_{sp}. This is done by manipulation of the actual sampling periods, h_i.

Design Considerations

A number of design considerations exist. The utilization setpoint, U_{sp}, must be chosen. The choice will depend on the scheduling policy of the dispatcher and on the sensitivity of the controllers to missed deadlines. A too low setpoint will give low resource utilization and poor control performance. A too high setpoint, on the other hand, may cause tasks to suffer from temporary overruns due to the varying execution times. Notice that the well-known, guaranteed utilization bounds of 100% for deadline-driven scheduling and 69% for priority-driven scheduling [Liu and Layland, 1973] are not valid in this context, since the assumptions about known, fixed WCETs and fixed periods are violated.

The feedback scheduler itself will execute as a periodic task in the system, and its period, h_{FBS}, must be chosen. A shorter period will give good control of the utilization but also consume much of the available resources. A longer period will consume less resources but will make the scheduler respond slower to CPU load disturbances.

The parameters in a digital controller typically depend on the sampling interval. Hence, the controller must be aware of the current period and adjust its parameters accordingly. On-line recalculations are often too costly. Instead, parameters for a range of sampling periods can be calculated off-line and stored in a table. Furthermore, the controller should be realized in a suitable form, such that the sampling period adjustments do not introduce transfer bumps.

Controlling the Utilization

The utilization control mechanism of the feedback scheduler consists of two parts: a utilization observer that estimates the current CPU utilization, and a resource allocator that divides the available resources among the tasks. In our architecture, the scheduler obtains execution-time estimates $\hat{C}_i$ from filtered job execution-time measurements,

$$\hat{C}_i(k) = \lambda \hat{C}_i(k-1) + (1-\lambda)c_i. \tag{4.1}$$

Here, λ is a forgetting factor. Setting λ close to one results in a smooth, but slow estimate. A λ close to zero gives a faster, but more noisy execution-time estimate. Combining the execution-time estimates with the nominal sampling periods, the *nominal utilization* $\hat{U}$ can be estimated as

$$\hat{U} = \sum_{i=1}^{n} \frac{\hat{C}_i}{h_{nom\,i}}. \tag{4.2}$$

If $\hat{U} \leq U_{sp}$, all tasks are allowed to execute with their nominal periods. If not, the feedback scheduler assigns new sampling periods such that the utilization setpoint is reached. This can be done in different ways. One option is to apply simple linear rescaling of the task periods. Another option is to derive optimal sampling periods for the controllers using control-theoretical arguments. Both options will be considered in the follwing sections.

Feedforward Information

The role of the feedforward information from the controllers to the feedback scheduler is three-fold. First, the scheduler can react to sudden changes in the workload by executing an extra time in connection with a mode change. The controller, which typically executes more frequently than the feedback scheduler, is responsible for signaling the scheduler as soon as a mode switch condition has been detected. If the mode change is likely to increase the workload, and if the switching time itself is not critical, the controller could delay the switch one or several sampling periods while the scheduler recalculates the periods. Such a non-critical mode switch could for instance be the result of an operator entering a new setpoint for the controller.

Second, the scheduler may need to keep track of the modes of the controllers in order to assign suitable sampling periods. The controller could communicate the new nominal sampling period at the mode switch, or the scheduler could keep a table of the nominal sampling periods for all the controller modes.

Third, the mode information allows the scheduler to run separate execution-time estimators in the different modes. The forgetting factor λ can then be chosen according to the execution-time variability within each mode. At a mode change, the scheduler can immediately switch to the current estimate in the new mode. This further improves the regulation of the utilization at the mode changes.

4.3 Case Study 1: Hybrid Controllers

As a first example of feedback scheduling, we will study the problem of scheduling a set of hybrid controllers that switch between different internal modes. The resource allocation is performed by simple rescaling of the nominal sampling periods.

A Hybrid Controller

A hybrid controller for the double-tank process, see Figure 4.2, was designed and implemented in [Eker and Malmborg, 1999]. The goal is to control the level of the lower tank to a desired setpoint. The measurement

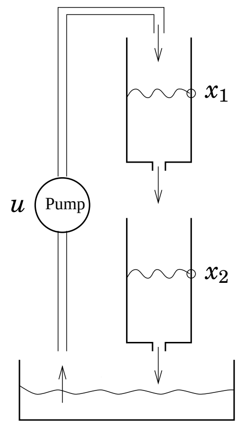


Figure 4.2 The double-tank process.

signals are the levels of both tanks, and the control signal is the inflow to the upper tank. Choosing state variables x_1 for the upper tank level and x_2 for the lower tank level, we get a nonlinear state-space description in the form

$$\begin{aligned} \dot{x}_1(t) &= -\alpha\sqrt{x_1(t)} + \beta u(t), \\ \dot{x}_2(t) &= \alpha\sqrt{x_1(t)} - \alpha\sqrt{x_2(t)}. \end{aligned} \tag{4.3}$$

The process constants α and β depend on the cross-sections of the tanks, the outlet areas, and the capacity of the pump. The control signal u is limited to the interval [0,1].

Traditionally, there is a trade-off in design objectives when choosing controller parameters. It is usually hard to achieve the desired step-change response and at the same time get the wanted steady-state behavior. An example of contradictory design criteria is tuning a PID controller to achieve both fast response to setpoint changes, fast disturbance rejection, and a small overshoot. In process control it is common practice to use PI control for steady state regulation and to use manual control for large setpoint changes. To approach taken here is to use a hybrid controller consisting of two subcontrollers, one PID controller and one time-optimal controller, together with a switching scheme. The time-optimal controller is used when the states are far away from the setpoint. Coming close, the PID controller is switched in to replace the time-optimal controller. The design of the controllers was reported in [Eker and Malmborg, 1999] and is summarized below.

PID Controller. Linearizing (4.3) around the equilibrium state corresponding to the tank level setpoint y_{sp}, the PID parameters K, T_i, and T_d

are calculated to give the closed-loop characteristic polynomial

$$(s+\omega_0)(s^2+2\zeta\omega_0 s+\omega_0^2), \tag{4.4}$$

where $\omega_0 = 6$ and $\zeta = 0.7$ are chosen to give fast rejection of load disturbances. The following discrete-time implementation, which includes low-pass filtering of the derivative part ($N = 10$), is used:

$$\begin{aligned} P(k) &= K(y_{sp}(k) - y(k)), \\ I(k) &= I(k-1) + \tfrac{Kh}{T_i}(y_{sp}(k) - y(k)), \\ D(k) &= \tfrac{T_d}{Nh+T_d}D(k-1) + \tfrac{NKT_d}{Nh+T_d}(y(k-1) - y(k)), \\ u(k) &= P(k) + I(k) + D(k). \end{aligned} \tag{4.5}$$

Time-Optimal Controller. For the linearized system, the time-optimal control signal is of bang-bang type. Solving the optimal control problem, it is possible to derive the switching curve

$$x_2(x_1) = \frac{1}{a}\left((ax_1 - b\bar{u})\left(1 + \ln\frac{ay_{sp} - b\bar{u}}{ax_1 - b\bar{u}}\right) + b\bar{u}\right), \tag{4.6}$$

where a and b are process constants, y_{sp} is the setpoint, and $\bar{u}$ takes values in $\{0, 1\}$. The optimal control signal is $u = 0$ above the switching curve and $u = 1$ below. A closeness criterion on the form

$$V_{close} = \begin{pmatrix} y_{sp} - x_1 \\ y_{sp} - x_2 \end{pmatrix}^T P(\theta,\gamma) \begin{pmatrix} y_{sp} - x_1 \\ y_{sp} - x_2 \end{pmatrix}, \tag{4.7}$$

where

$$P(\theta,\gamma) = \begin{pmatrix} \cos^2\theta + \gamma\sin^2\theta & (1-\gamma)\sin\theta\cos\theta \\ (1-\gamma)\sin\theta\cos\theta & \sin^2\theta + \gamma\cos^2\theta \end{pmatrix}, \tag{4.8}$$

is evaluated at each sample, to determine whether the controller should switch to PID mode. The parameters θ and γ determine the size and shape of the PID catching region.

Real-Time Properties

The execution-time properties of the hybrid double-tank controller were investigated in [Persson *et al.*, 2000]. The controller code and the implementation platform were instrumented to enable execution-time measurements. It was found that the time-optimal mode had considerably longer execution time than the PID mode. In each mode, the execution time was close to the best case most of the time, but it also exhibited random bursts.

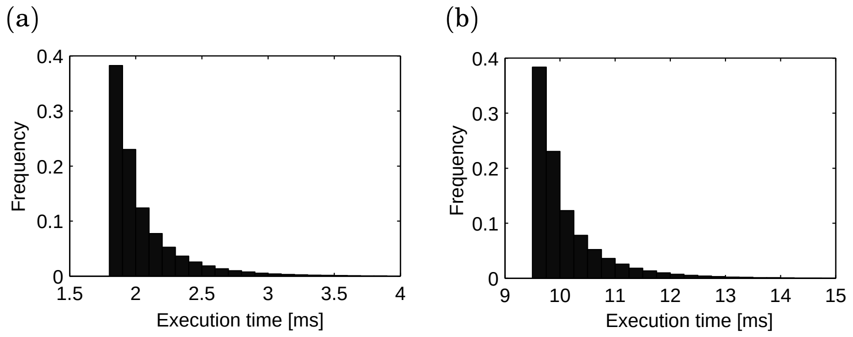


Figure 4.3 Assumed execution-time probability distributions in the hybrid controller: (a) PID mode, and (b) time-optimal mode.

For purposes of illustration, we here assume that the average execution time in the PID mode is $C_{\rm PID} = 2$ ms and the average execution time in the time-optimal mode is $C_{\rm Opt} = 10$ ms. The execution times of the algorithms are assumed to vary randomly from invocation to invocation according to the probability distributions shown in Figure 4.3.

Experiments

Assume that three hybrid double-tank controllers should execute on the same CPU. The double tanks have different physical parameters, resulting in different rise times, T_{r1}, T_{r2}, T_{r3} = 210, 180, 150 ms. Based on the rise times, the controllers are assigned the nominal sampling intervals h_{nom1}, h_{nom2}, h_{nom3} = 21, 18, 15 ms. A controller has the same nominal sampling period in both modes. In the experiments, rate-monotonic scheduling is assumed, i.e., the task with the shortest period is assigned the highest priority.

A simulation of Task 1 executing in isolation is shown in Figure 4.4. The process is disturbed by both input noise and measurement noise. The controller displays good setpoint response and steady-state regulation. It is seen that the CPU utilization is very low in PID mode, on average $C_{\rm PID}/h_{nom1} = 0.09$. In the time-optimal mode, it is significantly higher, on average $U = C_{\rm Opt}/h_{nom1} = 0.45$.

Various scheduling approaches in the multiple-process case are evaluated by cosimulation of the scheduling algorithm and the process dynamics using TrueTime (see Chapter 7). First, ordinary open-loop rate-monotonic scheduling is attempted. Next, a feedback scheduler is added to the system. Finally, feedforward is introduced in the scheduler. A 4-second simulation cycle is constructed as follows. At time $t = 0$, all controllers start in the PID mode. At $t = 0.5$, the worst-case scenario appears: all

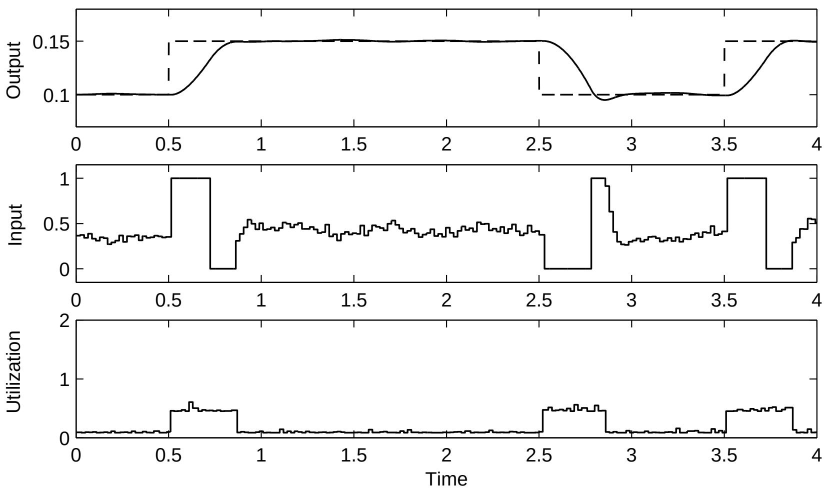


Figure 4.4 Performance of Task 1 when running in isolation. The controller displays good setpoint response and steady-state regulation. The CPU never becomes overloaded.

controllers receive new setpoints and should switch to time-optimal mode. Following this, the controllers get new setpoints pairwise, and then one by one. In each simulation, the behavior of Task 1, which is the lowest-priority controller, is plotted. Also plotted is the total CPU utilization, $U = \sum_i c_i/h_i$, where c_i is the current execution time of task i, and h_i is the current period of task i.

It is important to specify the behavior of the periodic tasks in the case of missed deadlines. No matter when a task finishes, the next release time is set to the current release time plus the assigned period of the task. Thus, a task that has missed many deadlines may have a release time that is far back in time compared to the actual starting time of the task. This standard implementation of periodic tasks, e.g., [Liu, 2000], penalizes especially the low-priority tasks in the case of an overload.

Simulation Results

The simulation results under various scheduling strategies are presented and discussed below.

Open-Loop Scheduling. We first consider open-loop scheduling, where the controllers are implemented as tasks with fixed periods equal to their nominal sampling intervals. The simulation results are shown in Figures 4.5 and 4.6.

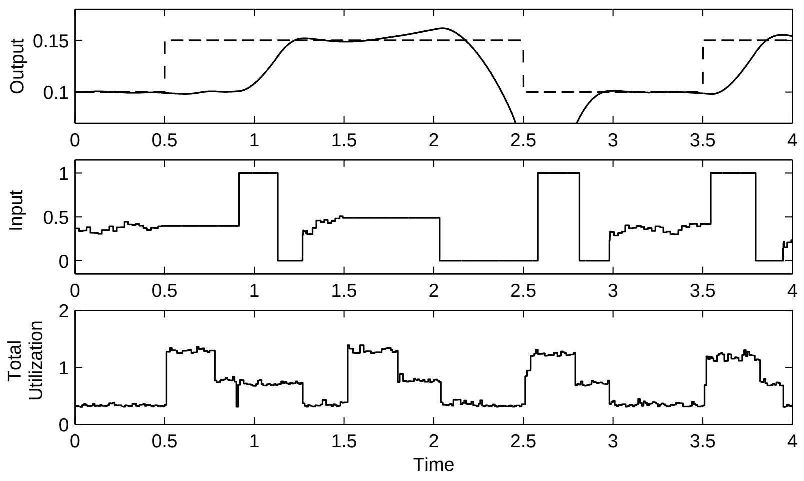


Figure 4.5 Performance of Task 1 under open-loop scheduling. The CPU is overloaded during long intervals, and the controller cannot update its control signal very often. The result is poor control performance.

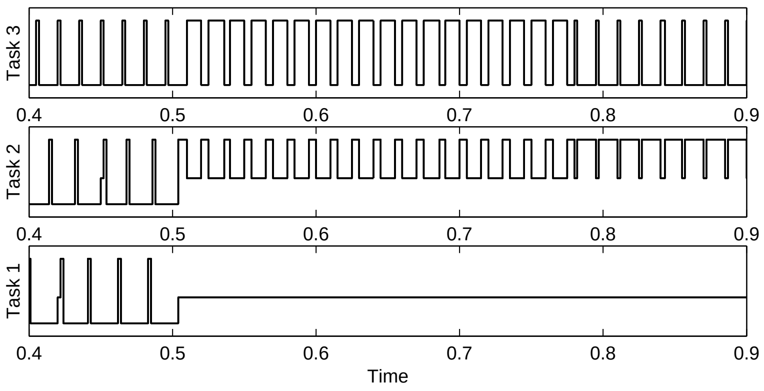


Figure 4.6 Close-up of the schedule under open-loop scheduling. At $t = 0.5$, Task 2 and 3 switch to time-optimal mode, and the CPU gets overloaded. As a result, Task 1 is preempted in a long interval.

The system easily becomes overloaded, since in the worst case, $U = \sum_i C_{\text{Opt}}/h_{nom\,i} = 1.7$. As seen in the schedule plot, Task 1 is completely blocked in the interval $t = [0.5, 0.9]$ because of preemption. The result is very poor control performance.

Feedback Scheduling. Next, a feedback scheduler is introduced. The scheduler is implemented as a high-priority task with a period of $T_{\mathrm{FBS}} = 100$ ms. The utilization setpoint is set to $U_{sp} = 0.8$. At each invocation, the feedback scheduler estimates the nominal utilization of the tasks by computing $\hat{U} = \sum_i \hat{C}_i / h_{nom\,i}$. The λ factor in the execution-time estimation is set to $\lambda = 0.9$. If $\hat{U} > U_{sp}$, task periods are assigned according to the linear rescaling

$$h_i = h_{nom\,i} \hat{U} / U_{sp}, \tag{4.9}$$

otherwise, the nominal sampling periods are used. The execution time of the feedback scheduler is assumed to be 2 ms. The simulation results are shown in Figures 4.7 and 4.8. The scheduler tries to keep the workload close to 0.8. However, there is a delay from a change in the utilization until it is detected by the feedback scheduler. This results in CPU overload peaks at the mode change instants. For instance, Task 1 is blocked in the interval $t = [0.5, 0.58]$.

Feedback and Feedforward Scheduling. To increase the responsiveness of the feedback scheduler, a feedforward mechanism is added. When a task in PID mode detects a new setpoint, it notifies the feedback scheduler, which is released immediately. The task periods are adjusted before the notifying task is allowed to continue to execute in the time-optimal mode. The execution-time estimation is improved by running separate estimators in the different modes. A forgetting factor of $\lambda = 0.9$ is chosen to give smooth estimates in both modes. The result is a more responsive and accurate feedback scheduler. The simulation results are shown in Figures 4.9 and 4.10. It is seen that the delay for Task 1 at the mode switches has been reduced, and that the control performance is slightly better.

Performance Evaluation and Summary. The performance of the controllers under different scheduling policies is evaluated using the criterion

$$J_i = \int_0^{T_{sim}} (y_{ideal\,i}(t) - y_i(t))^2 dt, \tag{4.10}$$

where $y_{ideal\,i}$ is the process output when Task i is allowed to run unpreempted at its nominal sampling interval and without any disturbances, while y_i is the actual process output when Task i is running in the multitasking real-time system. Ten simulation cycles ($T_{sim} = 40$) are simulated and the final accumulated costs for the controllers are summarized in Table 4.1.

Under open-loop scheduling, Task 3 has a only very small cost, due to the disturbances acting on the process. Task 2 suffers from some preemption from Task 3, which gives a small cost, while Task 1 is preempted during long intervals, which gives a very large final cost.

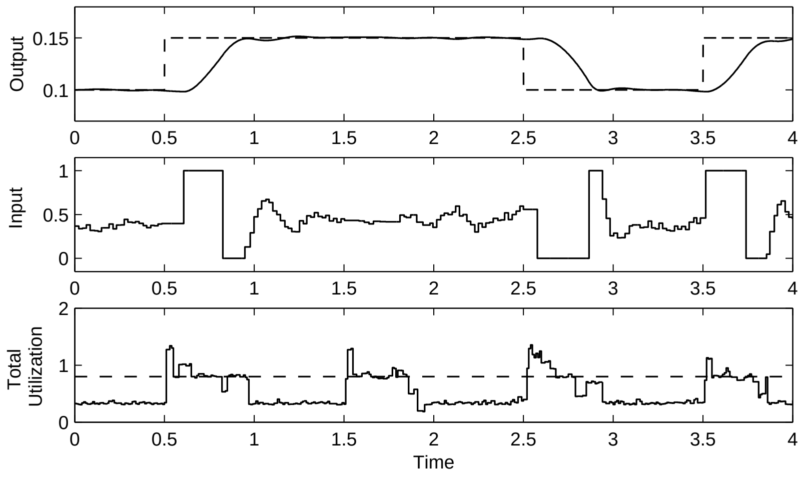


Figure 4.7 Performance of Task 1 under feedback scheduling. The CPU is overloaded in shorter intervals and the performance is better than under open-loop scheduling, cf. Figure 4.5.

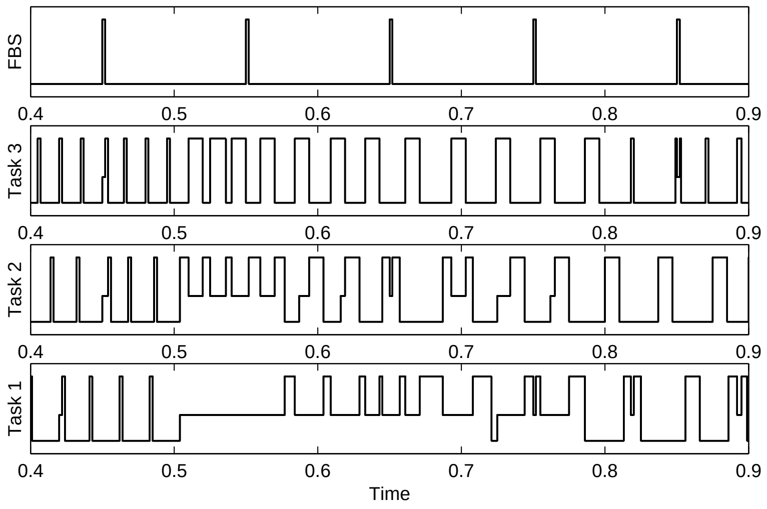


Figure 4.8 Close-up of the schedule under feedback scheduling. At $t = 0.5$, Task 3 switches to time-optimal mode, and the CPU gets overloaded. At $t = 0.55$, the feedback scheduler rescales the task periods. But this allows Task 2 to switch to time-optimal mode, and the CPU gets overloaded again.

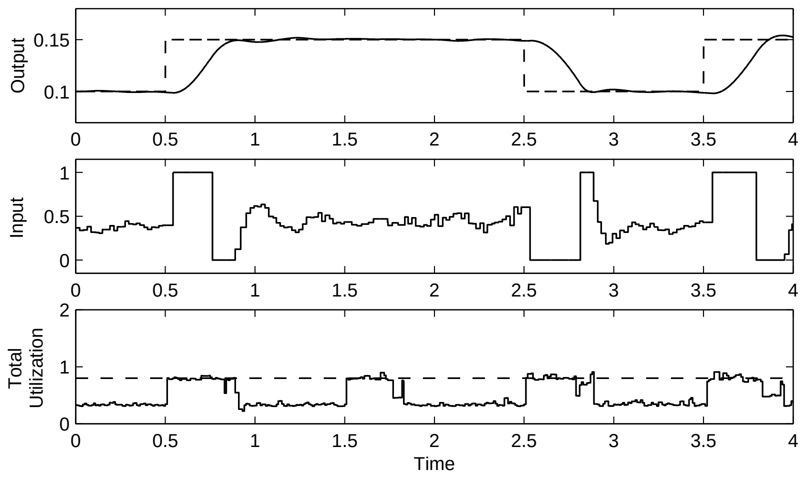


Figure 4.9 Performance of Task 1 under feedback and feedforward scheduling. The CPU is never overloaded, which results is better control performance.

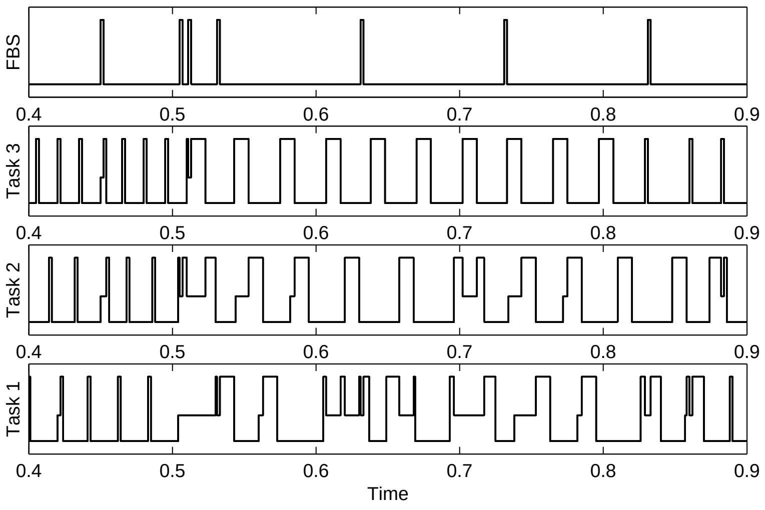


Figure 4.10 Close-up of the schedule under feedback and feedforward scheduling. The periods are immediately rescaled as each controller switches to time-optimal mode.

Table 4.1 Final accumulated costs for the three hybrid controllers under different scheduling strategies.

	J_1	J_2	J_3	$\sum J_i$
Open-loop scheduling	150	1.9	0.7	153
Feedback scheduling	4.6	1.1	1.2	6.9
Feedback-feedforward scheduling	2.1	1.3	1.1	4.5

Under feedback scheduling, the cost is much smaller for Task 1, due to the drastically reduced amount of preemption from Tasks 2 and 3. The performance is also improved for Task 2. Because of the period rescaling, Task 3 has a slight increase in its cost.

Under feedback and feedforward scheduling, Tasks 1 decreases its cost even further. The total cost is small, and it is more evenly distributed among the controllers.

4.4 Optimal Resource Distribution

In the previous example, linear rescaling of the nominal sampling periods was used as a simple mechanism to adjust the load in the real-time system. In this section, we will study the problem of optimal sampling period assignment for a set of linear controllers. It will be shown that, under certain assumptions, linear rescaling of the nominal sampling periods to meet the utilization setpoint is indeed optimal with respect to the overall control performance.

Cost Functions

As a quality-of-service measure, each sub-controller is associated with a *cost function* $J(h)$, which measures the performance of the controller as a function of the sampling period h. Typically, a longer sampling period implies a higher cost, although this is not always true (see [Eker *et al.*, 2000] for a counter-example).

Assuming a linear plant and a linear controller, a quadratic cost function can be computed using Jitterbug (see Chapter 6). Hence, we let each plant be described by a continuous-time linear system,

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + v(t), \\ y(t) &= Cx(t) + e(t), \end{aligned} \tag{4.11}$$

where v is a continuous-time white noise process with variance R_1 and e is a discrete-time white noise process with variance R_2. Each controller is described by a discrete-time linear system,

$$\begin{aligned} x_d(k+1) &= \Phi x_d(k) + \Gamma y(k), \\ u(k) &= C x_d(k) + D y(k), \end{aligned} \tag{4.12}$$

where the controller parameters Φ, Γ, C, and D are typically functions of the sampling interval. The performance of the controller is measured by a stationary, continuous-time quadratic cost function,

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} x(t) \\ u(t) \end{pmatrix} dt, \tag{4.13}$$

where Q is a positive semi-definite matrix. Ideally, the controller should be designed to minimize the cost function at each possible sampling period (i.e., the optimal LQG controller is derived for each value of h). As an alternative, a continuous-time controller could be discretized assuming different values of h.

Performance Optimization

The feedback scheduler should control the workload of the processor by adjusting the sampling periods of the controllers. At the same time, it should optimize the overall control performance. This is stated as the following optimization problem (originally formulated in [Seto *et al.*, 1996]): Given n control tasks with constant execution times $C_1, \ldots, C_n$ and sampling periods $h_1, \ldots, h_n$, the feedback scheduler should solve the problem

$$\begin{aligned} \min_{h_1,\ldots,h_n} \; & J_{tot} = \sum_{i=1}^{n} J_i(h_i), \\ \text{subject to} \; & \sum_{i=1}^{n} C_i/h_i \le U_{sp}, \end{aligned} \tag{4.14}$$

where U_{sp} is the desired processor utilization level. This problem has nonlinear constraints. To get linear constraints, the costs are recast as functions of the sampling frequencies f_i:

$$V_i(f_i) = J_i(1/h_i). \tag{4.15}$$

The problem is now written as

$$\begin{aligned} \min_{f_1,\ldots,f_n} \; & V_{tot} = \sum_{i=1}^{n} V_i(f_i), \\ \text{subject to} \; & \sum_{i=1}^{n} C_i f_i \le U_{sp}. \end{aligned} \tag{4.16}$$

Assuming that the functions $V_1(f_1), \ldots, V_n(f_n)$ are decreasing and convex, the optimal frequencies $f_1^\star, \ldots, f_n^\star$ fulfill the Kuhn-Tucker conditions

$$\begin{aligned}\frac{\partial}{\partial f_i} V_i(f_i^\star) + \lambda C_i &= 0,\\ \lambda\Big(U_{sp} - \sum_{i=1}^{n} C_i f_i^\star\Big) &= 0,\\ \lambda &\geq 0,\end{aligned} \tag{4.17}$$

where λ is the Lagrange multiplier (see, e.g., [Fletcher, 1987]).

Solving the optimization problem exactly can be very time-consuming. Evaluating a cost function for a single sampling frequency involves a large amount of computations. If the resource allocation problem is to be solved by an on-line optimizer, the cost functions for the plants must be computed off-line and then approximated by simpler functions. A quadratic approximation was suggested in [Eker *et al.*, 2000]. Here, we also present a linear approximation. It is shown that the solution to the approximated problem can in both cases be interpreted as a simple linear rescaling of the nominal sampling periods.

Quadratic Approximation. Assume that the cost functions can be approximated by

$$J_i(h_i) = \alpha_i + \beta_i h_i^2, \tag{4.18}$$

or, equivalently,

$$V_i(f_i) = \alpha_i + \beta_i (1/f_i)^2. \tag{4.19}$$

Applying the Kuhn-Tucker conditions (4.17) yields the explicit solution

$$f_i^\star = \left(\frac{\beta_i}{C_i}\right)^{1/3} \frac{U_{sp}}{\sum_{j=1}^{n} C_j^{2/3} \beta_j^{1/3}}. \tag{4.20}$$

Notice that the constants α_i can be disregarded, i.e., it is sufficient to estimate the curvature of the cost functions.

Linear Approximation. Assume that the cost functions can be approximated by

$$J_i(h_i) = \alpha_i + \gamma_i h_i, \tag{4.21}$$

or, equivalently,

$$V_i(f_i) = \alpha_i + \gamma_i / f_i. \tag{4.22}$$

This often seems to be a better approximation than (4.19). Note that the cost of the integrator process in Example 3.1 is described exactly by (4.21). Applying the Kuhn-Tucker conditions (4.17) yields the explicit solution

$$f_i^\star = \left(\frac{\gamma_i}{C_i}\right)^{1/2} \frac{U_{sp}}{\sum_{j=1}^n (C_j\gamma_j)^{1/2}}. \tag{4.23}$$

Notice that the constants α_i can be disregarded, i.e., it is sufficient to estimate the slope of the cost functions.

Interpretation as Simple Rescaling. Both the quadratic and the linear cost function approximations yield quite simple, explicit formulas for optimal task frequency assignment, that could be used on-line in a feedback scheduler. If a new task arrives, or if the execution time of a controller suddenly changes, new sampling periods could be calculated using (4.20) or (4.23).

However, not even that amount of calculations is really needed. In both cases, it can be noted that each task receives a share of the CPU that is proportional to a task constant. In the quadratic case, the proportionality constant is $(\beta_i/C_i)^{1/3}$, and in the linear case it is $(\gamma_i/C_i)^{1/2}$. The ratios between the optimal sampling periods are thus constant and do not depend on the available resources or the number of tasks in the systems. This implies that, if the nominal sampling periods have been chosen wisely, optimal feedback scheduling can be performed by simple rescaling of the task periods. This is formulated in the following theorem:

THEOREM 4.1
If the cost functions of the tasks in the system can be described by either (a) quadratic functions of the sampling period, Eq. (4.18), or by (b) linear functions of the sampling period, Eq. (4.21), and if nominal sampling frequencies $f_{nom\,1}, \ldots, f_{nom\,n}$ are chosen in proportion to (a) $(\beta_i/C_i)^{1/3}$, or (b) $(\gamma_i/C_i)^{1/2}$, then simple rescaling of the nominal frequencies to meet the utilization constraint is optimal with respect to the total control performance.

Proof. Follows from the proportionality argument above. □

Additional Constraints. It is possible to add more constraints to the approximate optimization problem and still retain a simple solution.

First, one can let the nominal sampling periods $f_{nom\,i}$ be minimal sampling periods. If $\sum_{i=1}^n C_i f_i \leq U_{sp}$, then the nominal periods are used, otherwise they are rescaled. This constraint prevents the CPU from being fully loaded when it is not necessary from a control performance point of view.

Second, one can impose maximum sampling periods to some tasks. This leads to an iterative solution (linear programming), where the remaining tasks are rescaled until all constraints are met.

4.5 Case Study 2: Linear Controllers

As a second example we study the problem of simultaneously stabilizing four inverted pendulums. Again, different scheduling approaches are evaluated by cosimulation of the scheduler, the control tasks, and the pendulums. By simulating the execution of the tasks, the effects of latency and jitter (due to varying execution times and scheduling) on the control performance are also captured in the results.

Plants and Controllers

Similar to the example in Section 3.6, each inverted pendulum is described by the system

$$G(s) = \frac{\omega_0^2}{s^2 - \omega_0^2}, \tag{4.24}$$

where ω_0 is the natural frequency of the pendulum. The pendulums have different lengths that correspond to different natural frequencies, $\omega_0 = 10, 13.3, 16.6, 20$ rad/s. The processes are disturbed by continuous-time input noise with the variance $R_1 = 1/\omega_0$ and discrete-time measurement noise with the variance $R_2 = 10^{-4}$.

Discrete-time LQG-controllers for the plants are designed to minimize the continuous-time cost function

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \left(y^2(t) + u^2(t)\right) dt. \tag{4.25}$$

The costs functions for the four pendulums as functions of the sampling period are computed using Jitterbug (see Chapter 6) and are shown in Figure 4.11. It is seen that the cost functions can be reasonably well approximated by linear functions,

$$J_i(h) = \alpha_i + \gamma_i h. \tag{4.26}$$

The estimated slopes of the cost functions are $\gamma = 43, 67, 95, 127$. It can be noted that, the higher the natural frequency of the pendulum is, the more sensitive the controller is towards an increase in the sampling period. This is quite intuitive.

Nominal sampling periods are chosen according to the solution of the performance optimization problem and according to the rule of thumb

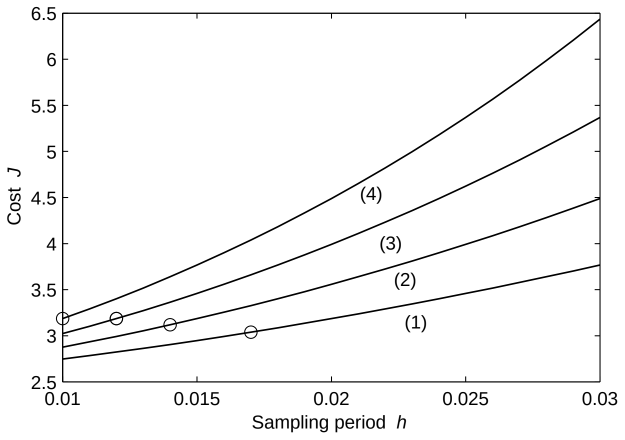


Figure 4.11 Cost functions for the four inverted pendulums. The circles indicate the nominal sampling periods.

[Åström and Wittenmark, 1997] that states that the sampling period should be chosen such that $0.2 < \omega_0 h < 0.6$. The resulting nominal periods are $h_{nom} = 17, 14, 12, 10$. These periods have been indicated in Figure 4.11. The optimal costs associated with these sampling periods are $J_0 = 3.04, 3.12, 3.19, 3.19$. These are the expected costs if the controllers could really execute at their nominal sampling periods, with zero latency and zero jitter. Implemented in a computer, the controllers will suffer from various amounts of sampling jitter, input-output latency, and input-output jitter, and the actual cost will be higher.

To allow for fast changes between different sampling periods during run-time, the LQG controller parameters are calculated off-line for a range of different sampling periods for each pendulum controller and stored in a table.

The Experiments

Each of the controllers has two different modes: on and off. When on, the average task execution time is $C = 5.5$ ms. The control tasks are implemented according to Listing 3.2. The execution time of the parts are assumed to be $C_{CO} = 2$ ms (Calculate Output) and $C_{US} = 3.5$ ms (Update State). The total execution time of the task is assumed to vary according to the probability distribution shown in Figure 4.12. When the controller is turned off, the execution time is zero.

At the start of the experiment, $t = 0$, Tasks 1 and 2 are on, while Tasks 3 and 4 are off. At $t = 2$, Task 3 switches on, and at $t = 4$, Task 4 also switches on. The four controllers run in parallel until $t = 6$.

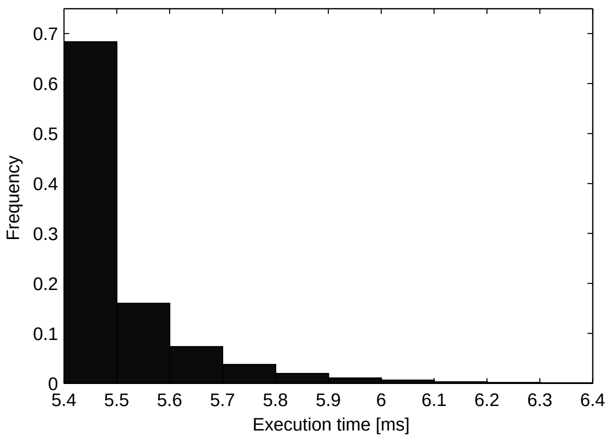


Figure 4.12 Assumed execution-time probability distribution of the LQG controller.

It is initially assumed that the feedback scheduler and the tasks are executing under FP scheduling. The feedback scheduler is given the highest priority while the control tasks are assigned rate-monotonic priorities. In this case, Task 1 will be given the lowest priority and will thus suffer the most during an overload.

It is assumed that the execution time of the feedback scheduling task is $C_{FBS} = 2$ ms. Its period is chosen as $h_{FBS} = 200$ ms and the utilization setpoint is chosen as $U_{sp} = 0.85$ to yield good control performance and not too many missed deadlines. The execution-time estimation forgetting factor is chosen as $\lambda = 0.9$. This gives smooth estimates but will cause the scheduler to react slowly to mode switches if the feedforward action is not used.

The experiment is repeated for different scheduling approaches. First, open-loop scheduling is attempted. Then, feedback scheduling without the feedforward mechanism is tried. Then, feedback-feedforward scheduling is studied. In the end, open-loop EDF scheduling is also studied.

To measure the performance of a controller, the accumulated cost is recorded,

$$J_i = \int_0^{T_{sim}} \left(y_i^2(t) + u_i^2(t)\right) dt. \tag{4.27}$$

If the pendulum falls down, the accumulated cost is set to infinity. The pendulums are subjected to identical sequences of process noise and measurement noise in all simulations. The execution times also consist of identical random sequences in all cases.

During each experiment, the schedule, i.e., the execution trace, is also recorded, together with the current utilization, $U = \sum_{i=1}^{n} \frac{c_i}{h_i}$, where c_i is the execution time of the latest invocation of task i and h_i is the sampling period currently assigned to task i. Proper regulation of the utilization should keep this quantity approximately equal to or less than U_{sp}.

Simulation Results

The simulation results in the four different scheduling cases are presented and discussed below.

Open-Loop Scheduling. Under open-loop scheduling, the controllers attempt to execute at their nominal sampling periods. The accumulated costs for the pendulums and the requested processor utilization are shown in Figure 4.13, and a close-up of schedule at $t = 4$ is shown in Figure 4.14.

Starting at $t = 0$, the average utilization is $U = C/h_{nom1} + C/h_{nom2} = 0.72$ and the control performance is good.

At $t = 2$, Task 3 starts to execute. The utilization of Tasks 2 and 3 is $U = C/h_{nom1} + C/h_{nom2} = 0.85$, which leaves only 15% of the CPU to Task 1, which is the lowest-priority task. The resulting average period is $h_1 = C/0.15 = 37$ ms, which is actually sufficient to stabilize the pendulum, although the cost increases more rapidly.

At $t = 4$, Task 4 is turned on. Tasks 3 and 4 taken together have the utilization $T = C/h_{nom3} + C/h_{nom4} = 1.01$, and this blocks Tasks 1 and 2 completely. The result is that both pendulums fall down.

Feedback Scheduling. Under feedback scheduling, the tasks starts to execute at their nominal sampling periods when turned on. The feedback scheduler then adjusts the periods every 200 ms. The accumulated costs for the pendulums and the requested processor utilization are shown in Figure 4.15, and a close-up of schedule at $t = 4$ is shown in Figure 4.16.

At $t = 2$, Task 3 is turned on and the requested utilization raises above the utilization setpoint of 0.85. At $t = 2.1$, the feedback scheduler adjusts the periods, and the overload condition is eventually removed. The same thing is repeated at $t = 4$ when Task 4 is turned on. The transient overload causes the performance of Task 1 to degrade, but the situation is soon stabilized. The slow response is due to the forgetting factor $\lambda = 0.99$. A lower value would correct the overload situation faster, but it would also increase the variance of the utilization in stationarity.

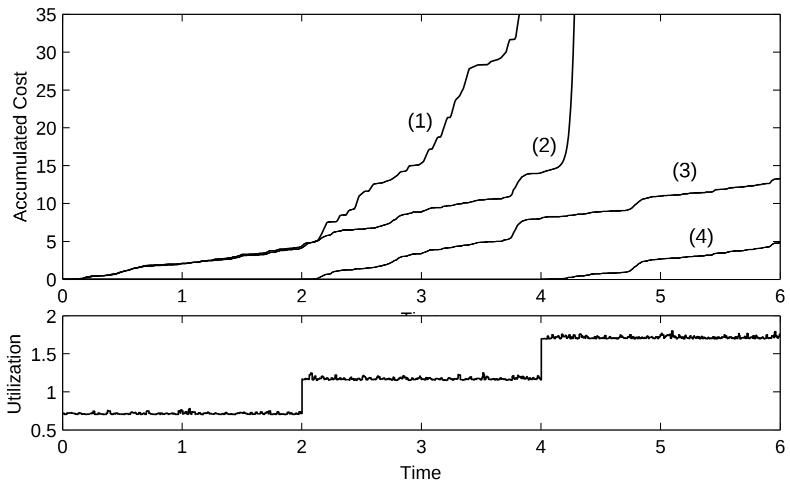


Figure 4.13 Accumulated costs and requested utilization under open-loop scheduling. At $t = 2$, the CPU becomes overloaded, which degrades the performance of Controller 1. At $t = 4$, Controller 2 also becomes unstable.

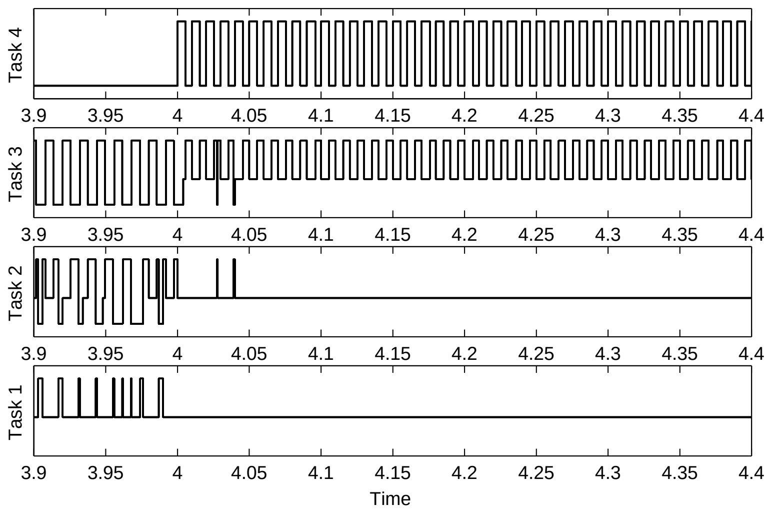


Figure 4.14 Close-up of schedule at $t = 4$ under open-loop scheduling. Tasks 2 and 3 become completely blocked when Task 4 starts to execute.

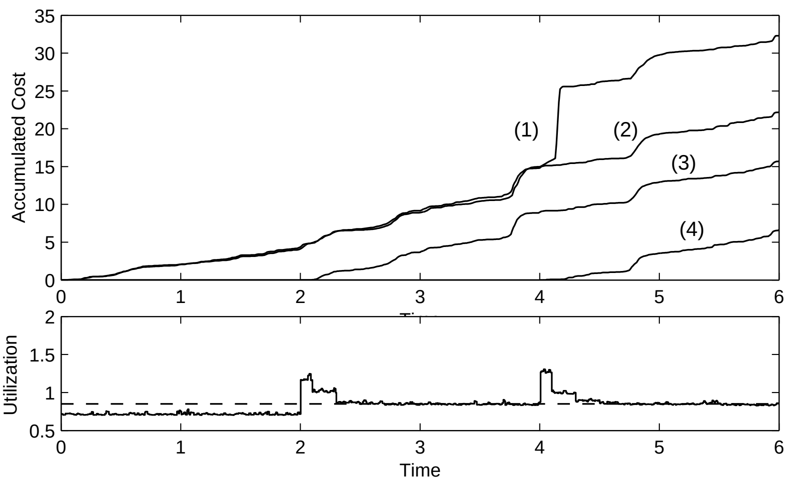


Figure 4.15 Accumulated costs and requested utilization under feedback scheduling. The overloads at time $t = 2$ and $t = 4$ are handled by the feedback scheduler.

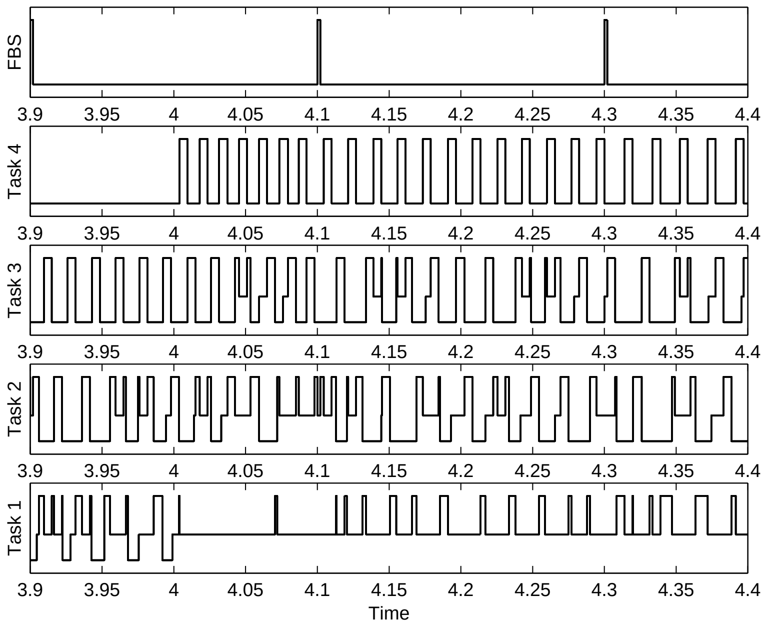


Figure 4.16 Close-up of schedule at $t = 4$ under feedback scheduling. Task 1 becomes blocked when Task 4 starts to execute. Eventually, the feedback scheduler rescales the sampling periods such that all tasks will have time to execute.

Feedback-Feedforward Scheduling. Under feedback-feedforward scheduling, tasks that switch modes also immediately activate the feedback scheduler. The accumulated costs for the pendulums and the requested processor utilization are shown in Figure 4.17, and a close-up of schedule at $t = 4$ is shown in Figure 4.18.

The results are similar to those under feedback scheduling, except at the mode changes. Here, the overloads are avoided due to the immediate rescaling of the task periods at $t = 2$ and $t = 4$. The transients are avoided, and the performance of all the controllers is good throughout.

Open-Loop EDF Scheduling. Under open-loop EDF scheduling, all tasks attempt to execute at their nominal periods. The relative deadline of each task is set to the period. The accumulated costs for the pendulums are shown in Figure 4.19 and a close-up of schedule at $t = 4$ is shown in Figure 4.20. The processor utilization is identical to the one under open-loop rate-monotonic scheduling, shown in Figure 4.13.

Although the system is overloaded from $t = 2$, the performance of Tasks 1–3 is good throughout. The reason is that ordinary EDF scheduling acts as a natural period rescaling mechanism in overload situations. This property is discussed further in Section 4.6.

Task 4 experiences some problems, however. When it is released at $t = 4$, the system has been overloaded for two seconds. This means that the absolute deadlines of Tasks 1–3 lie somewhere backwards in time, and they will have priority over Task 4, which initially has the absolute deadline $4 + h_{nom4} = 4.010$. The result is that Task 4 is blocked until around $t = 4.35$, before which the pendulum has fallen down.

Summary of Results and Discussion. The final accumulated costs for the four pendulums in the different cases are summarized in Table 4.2. In the ideal case, the execution time of the control algorithm has been set to 0 s. Of the other scenarios, feedback-feedforward scheduling gives the best overall control performance, although open-loop EDF scheduling does a quite good job at scheduling Tasks 1–3. The reason that the control performance is slightly better under feedback-feedforward scheduling than under open-loop EDF scheduling is that, in the EDF case, the controllers do not adjust their parameters according to the actual sampling periods.

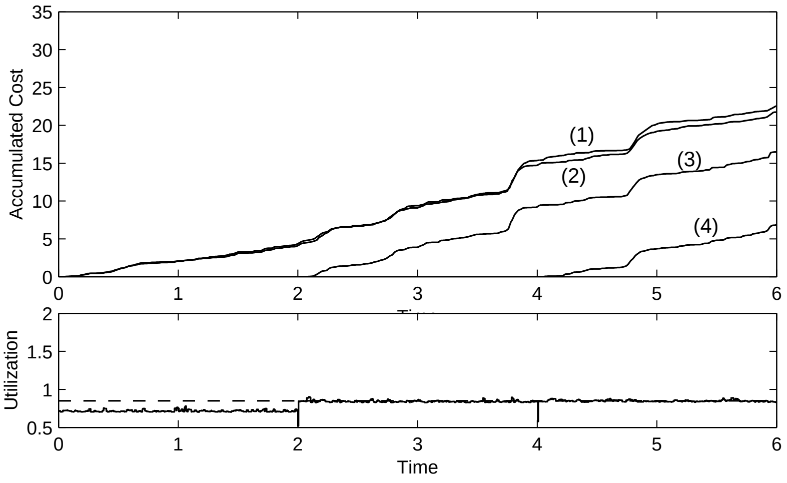


Figure 4.17 Accumulated costs and requested utilization under feedback-feedforward scheduling. The feedforward action makes it possible to avoid CPU overloads completely. The result is better control performance.

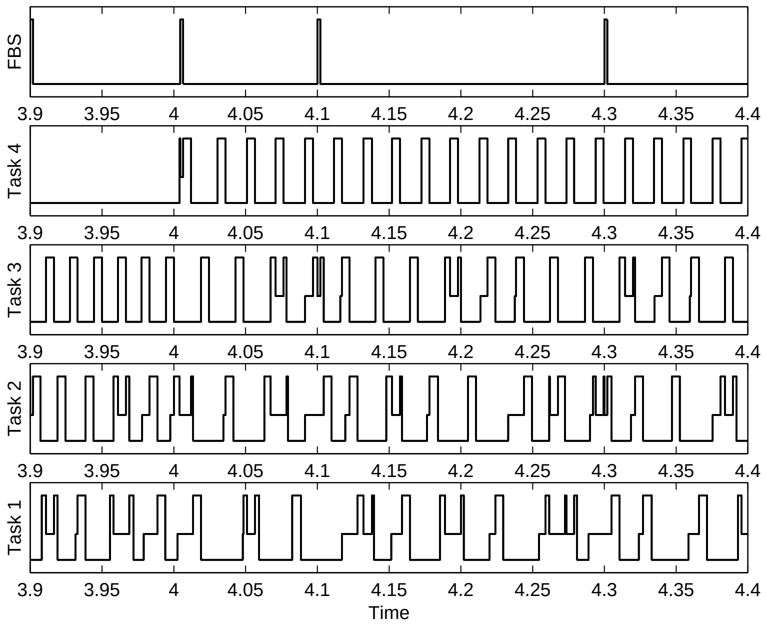


Figure 4.18 Close-up of schedule at $t = 4$ under feedback-feedforward scheduling. When Task 4 is turned on at $t = 4$, the feedback scheduler is immediately released.

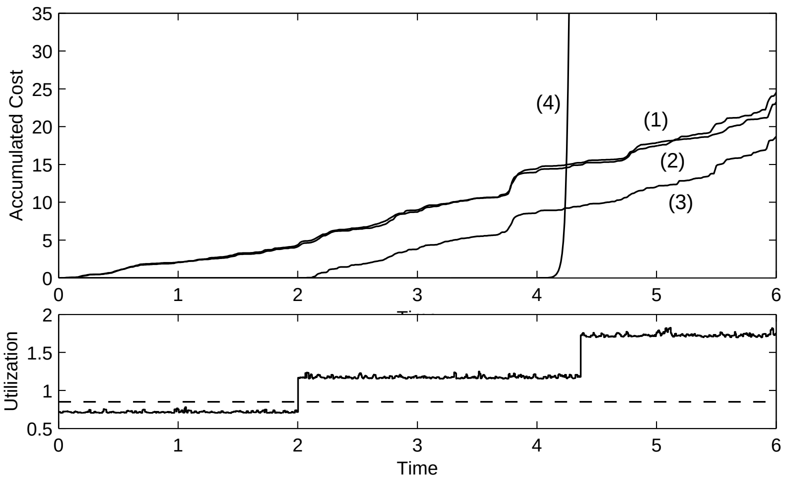


Figure 4.19 Costs and utilization under open-loop EDF scheduling. Despite the permanent overload from $t = 2$ and onwards, the performance of Controllers 1 to 3 is good throughout.

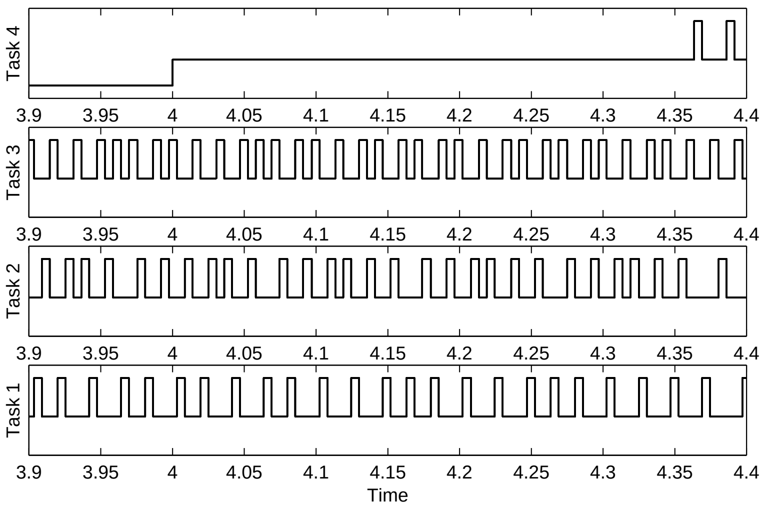


Figure 4.20 Close-up of schedule at $t = 4$ under open-loop EDF scheduling. Task 4 is blocked during a long interval due to the previous overload situation.

Table 4.2 Final accumulated costs for the four pendulum controllers under the different scheduling strategies.

	J_1	J_2	J_3	J_4	$\sum J_i$
Ideal	19	18	13	5	55
Open-loop scheduling	∞	∞	13	5	∞
Feedback scheduling	32	22	16	7	77
Feedback-feedforward scheduling	23	22	16	7	68
Open-loop EDF scheduling	23	25	19	∞	∞

4.6 EDF as a Feedback Scheduling Mechanism?

As seen in the simulations in the previous section, the performance of the controllers under open-loop EDF scheduling was quite good, despite the system being permanently overloaded and all deadlines being missed. This can be explained by the following theorem:

THEOREM 4.2
Assume a set of n periodic tasks, where each task i is described by a fixed period, T_i, a fixed execution time, C_i, a relative deadline, D_i, and a release offset, ϕ_i. The jobs of the tasks are scheduled according to their absolute deadlines (i.e., EDF scheduling). If $U = \sum_{j=1}^{n} C_j/T_j > 1$, then the average actual period of task i in stationarity, $\bar{T}_i$, is given by $\bar{T}_i = T_i U$.

Proof. See Appendix. □

Theorem 4.1 and Theorem 4.2 taken together give the following:

COROLLARY 4.1
An ordinary EDF scheduler can be interpreted, in stationarity, as an optimal feedback scheduler for control tasks that have cost functions that can be described by quadratic or linear functions of the sampling period. □

This result rests upon several assumptions, however. First, it is assumed that the controller samples the plant when it starts to execute, and not when it is released. This way, the input-output latency will be bounded, even though the response time might approach infinity.

Second, it is assumed that jitter has only negligible impact on the control performance. This may not be true during a permanent overload situation where the tasks start to execute non-preemptively. While the average period of a task is given by Theorem 4.2, the jitter may be unbounded because of the non-preemptive execution pattern. In the feedback scheduling example in the previous section this was not a problem, since all the tasks had execution times and periods of the same magnitude.

As seen in the example, problems may occur when tasks switch mode (and this is when feedback scheduling is really needed). Since tasks are scheduled using old deadlines, it will take time for a resource redistribution to have effect. One solution would be to reset the release time of all tasks to the current time immediately following a mode change.

Another problem with the open-loop EDF approach is that the period information is not communicated to the controllers. Thus, they cannot use the correct control parameters, and this degrades the performance to some degree. This could be corrected by letting the control tasks measure their own actual periods.

4.7 Conclusion

A scheduler architecture has been proposed that combines feedback and feedforward action in order to optimize control performance while maintaining high resource utilization. The feedback part relaxes the requirement on known execution-time bounds for multitasking control systems. The feedforward part allows for rapid adaptation to changing load conditions.

The control performance, or quality of control (QoC), is considered as a quality-of-service measure that should be maximized. Optimal adjustment strategies for the controller task periods have been derived for the cases when the cost function is a quadratic function of the sampling period and when it is a linear function of the sampling period. The adjustment strategy uses linear rescaling, making it computationally efficient, and hence, possible to use on-line.

The different strategies have been evaluated in simulated examples. The proposed approach gives substantially better results than what is achieved using classical open-loop scheduling methods. A new result for periodic tasks with EDF scheduling under overload conditions makes it possible, in certain situations, to interpret an ordinary EDF dispatcher as a feedback scheduler for control tasks.

5

The Control Server

5.1 Introduction

Traditional scheduling models give poor support for codesign of real-time control systems. One difficulty lies in the nonlinearity in scheduling algorithms such as fixed-priority (FP) or earliest-deadline-first (EDF) scheduling: a small change in a task parameter (period, execution time, deadline, priority, etc.) may give rise to unpredictable results in terms of latency and jitter. While latency and jitter can to some extent be addressed with more detailed control and scheduling analysis (see Chapters 2 and 3), the design problem quickly becomes extremely complicated. The issue becomes critical in feedback scheduling applications (Chapter 4), where the codesign problem should ideally be solved on-line as tasks enter or leave the system.

This chapter presents a novel computational model for control tasks, called the Control Server. The primary goal of the model is to facilitate simple codesign of flexible real-time control systems. In particular, the model should provide

(R1) isolation between unrelated tasks,

(R2) short input-output latencies,

(R3) minimal sampling jitter and input-output jitter,

(R4) a simple interface between the control design and the real-time design,

(R5) predictable control and real-time behavior, also in the case of overruns, and

(R6) the possibility to combine several tasks (components) into a new task (component) with predictable control and real-time behavior.

Requirement (R1) is fulfilled by the use of constant bandwidth servers (CBSs) [Abeni and Buttazzo, 1998]. The servers make each task appear as if it was running on a dedicated CPU with a given fraction of the original CPU speed. To facilitate short latencies (requirement (R2)), a task may be divided into a number of *segments*, which are scheduled individually. A task may only read inputs (from the environment or from other tasks) at the beginning of a segment and write outputs (to the environment or to other tasks) at the end of a segment. All communication is handled by the kernel and is hence not prone to jitter (requirement (R3)).

Requirements (R4)–(R6) are addressed by the combination of bandwidth servers and statically scheduled communication points. For periodic tasks with constant execution times, the model creates the illusion of a perfect division of the CPU, equivalent to the Generalized Processor Sharing (GPS) algorithm [Parekh and Gallager, 1993]. The model makes it possible to analyze each task in isolation, from both scheduling and control points of view. Like ordinary EDF, schedulability of the task set is simply determined by the total CPU utilization (ignoring context switches and the I/O operations performed by the kernel). The performance of a controller can also be viewed as a function of its alloted CPU share. These properties make the model very suitable for feedback scheduling applications.

Furthermore, the model makes it possible to combine two or several communicating tasks into a new task. The new task will consume a fraction of the CPU equal to the sum of the utilization of the constituting tasks. The new task will have a predictable I/O pattern, and, hence, also predictable control performance. Control tasks may thus be treated as *real-time components*, which can be combined into new components.

Related Work

Giotto [Henzinger *et al.*, 2001] is an abstract programming model for the implementation of embedded control systems. Similar to our model, I/O and communication are time-triggered and assumed to take zero time, while the computations inbetween are assumed to be scheduled in real-time. A serious drawback with the model is that a minimum of one sample input-output latency is introduced in all control loops. Also, Giotto does not address the scheduling problem.

Within the Ptolemy project [Bhattacharyya *et al.*, 2002], a computational domain called Timed Multitasking (TM) has been developed [Liu and Lee, 2003]. In the model, tasks (or *actors* in the terminology of Ptolemy) may be triggered by both periodic and aperiodic events. Inputs are read when the task is triggered and outputs are written at the specified task deadline. The computations inbetween are assumed to be scheduled by a fixed-priority dispatcher. In the case of a deadline overrun, an overrun handler may be called. Again, the scheduling problem is not explicitly addressed by the model.

In [Caccamo *et al.*, 2000b], a variant of the CBS server, called CBS$^{\mathrm{hd}}$, is used to schedule control tasks with varying execution times. In the case of an execution overrun, the current period is extended and the CBS budget is recharged in small increments until the task finishes.

The idea of jitter minimization using dedicated input and output tasks or interrupt handlers has been propsed several times, see Section 3.1.

5.2 The Model

The Control Server (CS) model assumes an underlying real-time operating system with an EDF scheduler. To guarantee isolation, all tasks in the system must belong to either one of two categories:

- CS tasks, suitable for control loops and other periodic activities with high demands for input/output timing accuracy.
- Tasks served by ordinary CBS servers, including aperiodic, soft and non-real-time tasks.

CS Tasks

A CS task τ_i is described by

- a CPU share U_i,
- a period T_i,
- a release offset ϕ_i,
- a set of $n_i \geq 1$ segments $S_i^1, S_i^2, \ldots, S_i^{n_i}$ of lengths $l_i^1, l_i^2, \ldots, l_i^{n_i}$ such that $\sum_{j=1}^{n_i} l_i^j = T_i$,
- a set of inputs I_i (associated with physical inputs or shared variables), and
- a set of outputs O_i (associated with physical outputs or shared variables).

Associated with each segment S_i^j are

- a subset of the task inputs, $I_i^j \in I_i$,
- a code function f_i^j, and
- a subset of the task outputs, $O_i^j \in O_i$,

The segments can be thought of as a static cyclic schedule for the reading of inputs, the writing of outputs, and the release of jobs. At the beginning of a segment S_i^j, i.e., when $t = \phi_i + \sum_{k=1}^{j-1} l_i^k \pmod{T_i}$, the inputs I_i^j are read and a job executing f_i^j is released. At the end of the segment, i.e., when $t = \phi_i + \sum_{k=1}^{j} l_i^k \pmod{T_i}$, the outputs O_i^j are written.

The jobs produced by a CS task τ_i are served on a first-come, first-served basis by a dedicated, slightly modified CBS with the following attributes:

- a server bandwidth equal to the CPU share U_i,
- a dynamic deadline d_i,
- a server budget c_i, and
- a segment counter m_i.

The server is initialized with $c_i = m_i = 0$ and $d_i = \phi_i$. The rules for updating the server are as follows:

1. During the execution of a job, the budget c_i is decreased at unit rate.
2. If $c_i = 0$, *or*, if a new job arrives at time r and $d_i = r$, then
   - the segment counter is updated, $m_i := \mathrm{mod}(m_i, n_i) + 1$,
   - the deadline is moved, $d_i := d_i + l_i^{m_i}$, and
   - the budget is recharged to $c_i := U_i \, l_i^{m_i}$.

The rules are somewhat simplified compared to the original CBS rules (see Section 2.2) due to the predictable pattern of release times and deadlines. The only real difference from an ordinary CBS is that here a "dynamic server period", equal to the current segment length, $l_i^{m_i}$, is used.

Figure 5.1 shows an example of a CS task with two segments executing alone. This is a typical model of a control algorithm, which has been split into Calculate Output and Update State (see Chapter 3). The lengths of the segments are 2 and 4 units respectively, and the task CPU share is $U = 0.5$. At the beginning of the first segment, an input is read, and at the end of the first segment, an output is written. The two first jobs consume less than their budgets (which are 1 and 2 units respectively), while the third job has an overrun at time 7. This causes the deadline to be moved to the end of the next segment and the budget to be recharged to 2 units (hence "borrowing" budget from the fourth job). In this example, the latency is constant and equal to 2 units (the length of the first segment) despite the variation in the job execution times.

Note that CS rules allow for budget recharging across the task period. The server deadline of a task that has constant overruns will be postponed repeatedly and eventually approach infinity.

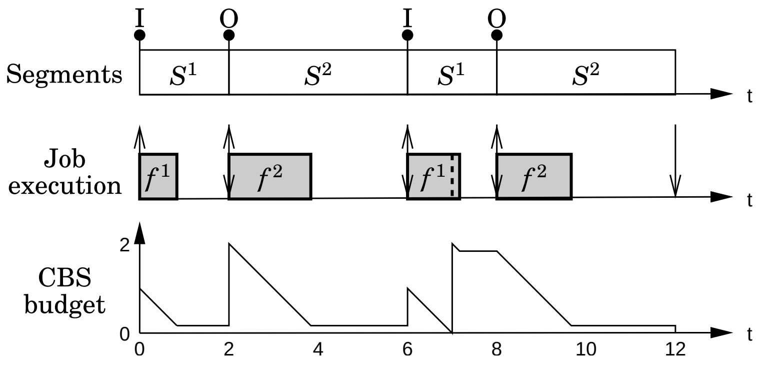


Figure 5.1 Example of a CS task executing alone. The up arrows indicate job releases and the down arrows indicate deadlines. The overrun at $t = 7$ causes the deadline to be postponed to the end of the next segment.

Communication and Synchronization

The communication between tasks and the environment requires some amount of buffering. When an input is read at the beginning of a segment, the value is stored in a buffer. The value in the buffer is then read from user code using a real-time primitive. The read operation is non-blocking and non-consuming, i.e., a value will always be present in the buffer and the same value can be read several times. Similarly, another real-time primitive is used to write a new output value. The value is stored in a buffer and is written to the output at the end of the relevant segment. The write operation is non-blocking and any old value in the buffer will be overwritten.

Communication between tasks is handled via shared variables. If an input is associated with a shared variable, the value of the variable is copied to the input buffer at the beginning of the relevant segment. Similarly, if an output is associated with a shared variable, the value in the output buffer is copied to the shared variable at the end of the relevant segment.

If two tasks should write to the same physical output or shared variable at the same time, the actual write order is undefined. More importantly, if one task writes to a shared variable and another task reads from the same variable at the same time, *the write operation takes place first*. The offsets can hence be used to line up tasks such that the output from one task is immediately read by another task, minimizing the end-to-end latency.

The use of buffers and non-blocking read and write operations allow tasks with different periods to communicate. The periods of two communicating tasks need not be harmonic, even if this makes most sense in typical applications. However, for the kernel to be able to accurately determine if a read and write operation really occurs simultaneously, the offsets, periods, and segment lengths of a set of communicating tasks need to be integer multiples of a common tick size. For this purpose, communicating tasks are gathered into *task groups*. This is described further in the implementation section.

Scheduling Properties

From a schedulability point of view, a CS task with the CPU share U_i is equivalent to a CBS server with the bandwidth U_i. In [Abeni, 1998], it is shown that a CBS can never demand more than its reserved bandwidth. The proof is based on the processor demand approach (see Section 2.2). By postponing the deadline when the budget is exhausted, the loading factor of the jobs served by the CBS can never exceed U_i. The same argument holds for the modified CBS used in the CS model. A set of CBS and CS tasks is thus schedulable if and only if

$$\sum U_i \leq 1. \tag{5.1}$$

If the segment lengths of a CS task τ_i are chosen such that

$$l_i^j = C_i^j / U_i, \tag{5.2}$$

where C_i^j denotes the worst-case execution time (WCET) of the code function f_i^j, overruns will never occur (i.e., the budget will never be exhausted before the end of the segment), and all latencies will be constant. For tasks with large variation in their execution time, it can sometimes be advantageous to assign segment lengths that are shorter than those given by Eq. (5.2). This means that some deadlines will be postponed and that the task may not always produce a new output in time, delaying the output one or more periods. An example of when this can actually give better control performance (for a given value of U_i) is given later.

5.3 Control and Scheduling Codesign

As stated in the introductory chapter of the thesis, the control and scheduling codesign problem can be formulated as follows: Given a set of processes to be controlled and a computer with limited resources, a set of controllers should be designed and scheduled as real-time tasks such that the overall control performance is optimized. With dynamic scheduling algorithms such as EDF and RM, the general design problem is extremely difficult due to the complex interaction between task parameters, control parameters, schedulability, and control performance.

With our model, the link between the scheduling design and the control design is the CPU share U. Schedulability of a task set is simply determined by the total CPU utilization. The performance (or *cost*) J of a controller executing in a real-time system can—roughly speaking—be expressed as a function of the sampling period T, the input-output latency L_{io}, the sampling jitter J_s, and the input-output jitter J_{io}:

$$J = J(T,\, L_{io},\, J_s,\, J_{io}). \tag{5.3}$$

Assuming that the first segment contains the Calculate Output part of the control algorithm, and that the segment lengths are chosen according to Eq. (5.2), execution under the Control Server implies

$$\begin{aligned} T &= \sum l^k = \sum C^k/U, \\ L_{io} &= l^1 = C^1/U, \\ J_s &= 0, \\ J_{io} &= 0. \end{aligned} \tag{5.4}$$

The only independent variable in the expressions above is U. The control performance can thus be expressed as a function of U only:

$$J = J(U). \tag{5.5}$$

Assuming a linear controller, a linear plant, and a quadratic cost function, the performance of the controller for different values of U can easily be computed using Jitterbug (see Chapter 6).

The elimination of the jitter has several advantages. First, it is easy to design a controller that compensates for a constant delay. Second, the performance degradation associated with the jitter is removed. Third, it becomes possible to accurately predict the performance of the controller. These properties are exploited in the codesign examples below.

Codesign Example 1: Optimal Period Selection

In this example we study the problem of optimal sampling period selection for a set of control loops. This type of codesign problem first appeared in [Seto *et al.*, 1996]. A similar problem was studied in conjunction with the feedback scheduler in Chapter 4. In those cases, however, the scheduling-induced latency and jitter was ignored.

Suppose for instance that we want to control three identical integrator processes,

$$\frac{dx(t)}{dt} = u(t) + v_c(t), \tag{5.6}$$

where x is the state, u is the control signal, and v_c is a continuous-time white noise process with zero mean and unit variance. A discrete-time controller is designed to minimize the continuous-time cost function

$$J = \lim_{T\to\infty} \frac{1}{T}\int_0^T x^2(t)\,dt. \tag{5.7}$$

assuming the sampling period h and a constant input-output latency L. As shown in Example 3.1, the cost of the optimal controller is given by

$$J(h,\,L) = \frac{3+\sqrt{3}}{6}h + L. \tag{5.8}$$

The assumed design goal is to select sampling periods h_1, h_2, h_3 such that a weighted sum of the cost functions,

$$J_{tot} = w_1\,J(h_1,\,L_1) + w_2\,J(h_2,\,L_2) + w_3\,J(h_3,\,L_3), \tag{5.9}$$

is minimized subject to the utilization constraint

$$U = \frac{C}{h_1} + \frac{C}{h_2} + \frac{C}{h_3} \le 1.$$

Here, C is the (constant) total execution time of the control algorithm. Assigning segment lengths proportional to the parts of the algorithm, the CS model implies the same relative latency $a = L/h$ for all controllers. Using (5.8) the objective function (5.9) can be written

$$J_{tot} = \left(\frac{3+\sqrt{3}}{6} + a\right)(w_1h_1 + w_2h_2 + w_3h_3).$$

The solution to the optimization problem is (see Eq. (4.23))

$$h_1 = b/\sqrt{w_1}, \quad h_2 = b/\sqrt{w_2}, \quad h_3 = b/\sqrt{w_3},$$

where $b = C(\sqrt{w_1} + \sqrt{w_2} + \sqrt{w_3})$. (For more general problems numerical optimization must be performed.) Contrary to [Seto *et al.*, 1996] (where RM or EDF scheduling is assumed), our model allows for the latency and the (non-existent) jitter to be accounted for in the optimization.

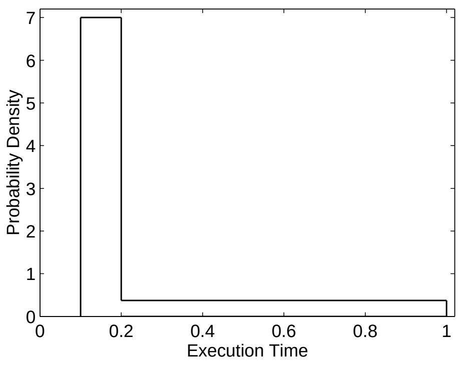


Figure 5.2 Assumed execution time probability distribution of the integrator controller.

Codesign Example 2: Allowing Overruns

For controllers with large variations in their execution time, it can sometimes be pessimistic to select task periods (and segment lengths) according to the WCETs. The intuition is that, given a task CPU share, it may be better to sample often and occasionally miss an output, than to sample seldom and always produce an output. With our model, it becomes easy to predict the worst-case effects (i.e., assuming that the rest of the CPU is fully utilized) of such task overruns.

Again consider the integrator controller. For simplicity, it is assumed that the controller is implemented as a single segment, i.e., we have $L_{io} = T$ if no overrun occurs, and that the assigned CPU share is $U = 1$. Now assume that the execution time of the controller is given by the probability distribution in Figure 5.2. Choosing a period less than the WCET means that some outputs will be missed and that the actual latency will vary randomly between T, $2T$, $3T$, etc. The resulting control performance for such a model can be computed using Jitterbug (see Chapter 6). In Figure 5.3 the cost (3.2) has been computed for different values of the task period. The optimal cost $J = 1.67$ is obtained for $T = 0.76$. For that period, overruns will occur in 9% of the periods (introducing a latency of $2T$ or more). The example shows that our model can be used to “cut off the tail” of execution time distributions with safe and predictable results.

5.4 CS Tasks as Real-Time Components

As argued in the previous section, given a control algorithm with known execution time C (divided into one or several segments), the sampling pe-

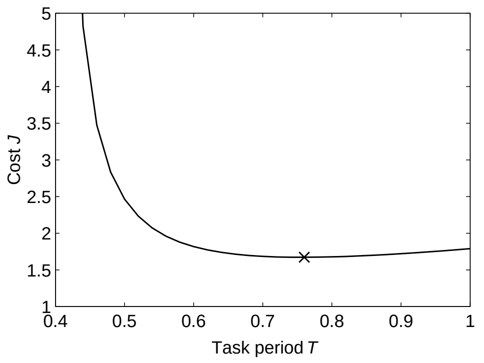


Figure 5.3 Cost as a function of the task period for the integrator controller with varying execution time. The optimal period is $T = 0.76$.

riod T, the latency L_{io}, and the control performance J can be expressed as functions of the CPU share U. The predictable control and scheduling properties allows a CS task to be viewed as a scalable real-time component.

Consider for instance the PID (proportional-integral-derivative) controller component in Figure 5.4. The controller has two inputs: the reference value r and the measurement signal y, and one output: the control signal u. The U knob determines the CPU share. An ordinary software component (see, e.g., [Crnkovic and Larsson, Eds., 2002]) would only specify the functional behavior, i.e., the PID algorithm. The specification for our real-time component includes the resource usage and the timely behavior. Assuming an implementation where the execution time of the Calculate Output part is 0.25 ms and the execution time of the Update State part is 0.75 ms, the specification could look something like this:

- Algorithm: $u = K(r - y) + \ldots$,
- Parameters: U, K, T_i, T_d, $\ldots$,
- $C = 1$ ms,

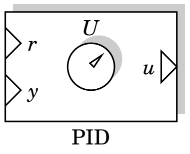


Figure 5.4 A PID controller component.

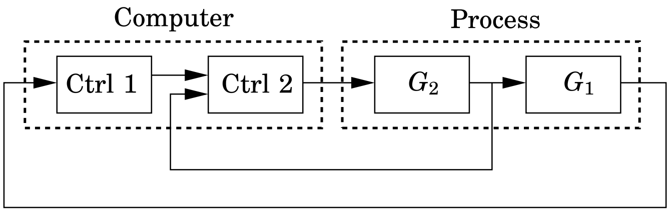


Figure 5.5 Cascaded controller structure.

- $T = C/U$,
- $L_{io} = T/4$,
- $J = J(U)$ (specified as an analytical function or a table).

Note that our model guarantees that the controller will have the specified behavior, regardless of other tasks in the system.

Next, consider the composition of two PID controllers in a cascaded controller structure, see Figure 5.5. In this very common structure, the inner controller is responsible for controlling the (typically) fast process dynamics G_2, while the outer controller handles the slower dynamics G_1. A cascaded controller component can be built from two PID components as shown in Figure 5.6. In this case, it is assumed that the inner controller should have twice the sampling frequency of the outer controller (reflecting the speed of the processes). This is achieved by assigning the shares $U/3$ to PID1 and $2U/3$ to PID2, U being the CPU share of the composite controller. The end-to-end latency in the controller can be minimized by a suitable segment layout, see Figure 5.7.

The schedulability and performance of the cascaded controller will, again, only depend on the total assigned CPU share U. The resulting controller is a multi-rate controller and its performance can be computed

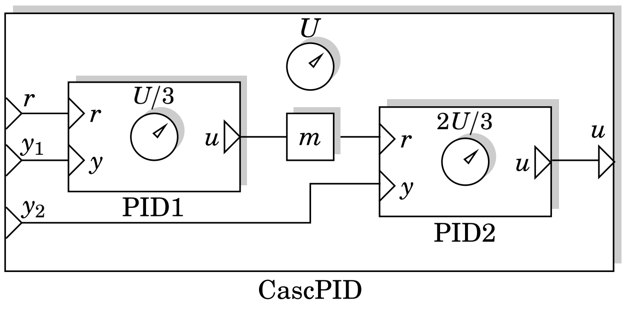


Figure 5.6 A cascaded PID controller component.

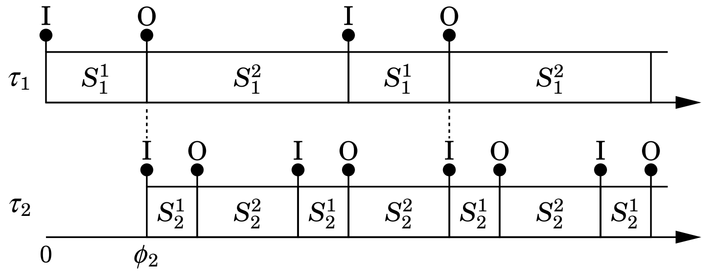


Figure 5.7 Segment layout in the cascaded PID controller. Task τ_2 is given an offset $\phi_2 = l_1^1$ such that the value written by S_1^1 is immediately read by S_2^1.

using Jitterbug (see Example 3 in Section 6.4). Note that composition is not possible with ordinary threads, i.e., two communicating threads cannot be treated as one, neither from schedulability nor control perspectives.

5.5 Feedback Scheduling

The CS model is suitable as a platform for feedback scheduling applications. The optimal resource allocation problem studied in Chapter 4 does not account for latency and jitter in the control loops. Under the CS model, however, the cost function $J(U)$ expresses the true performance of the controller executing in the real-time system. Furthermore, the use of CBS servers ensures that the control tasks will not consume more than their allocated CPU shares. This can be viewed as an inner loop in the feedback scheduling structure.

We will now revisit the feedback scheduling example from Section 4.5. Each pendulum controller is implemented as a CS task with two segments: Calculate Output (C_{CO} = 2 ms) and Update State (C_{US} = 3.5 ms). The control parameters and the cost functions are recomputed to account for the constant input-output latency. Since the CS model is based on EDF scheduling, higher CPU utilization may be achieved. Allowing for some implementation overhead, the utilization setpoint is chosen as $U_{sp} = 0.95$. A simulation of the system assuming feedback-feedforward scheduling is shown in Figure 5.8. The final accumulated costs are given in Table 5.1. The total cost is lower than under fixed-priority scheduling, cf. Table 4.2. A close-up of the schedule at time $t = 4$ is shown in Figure 5.9. It is seen that considerable execution jitter is introduced by the EDF scheduling algorithm. This does not affect the input and output operations, however, since they are handled by the kernel (schedule not shown).

Table 5.1 Final accumulated costs for the four pendulums under CS scheduling. Compare with Table 4.2.

	J_1	J_2	J_3	J_4	$\sum J_i$
CS scheduling	21	21	16	7	65

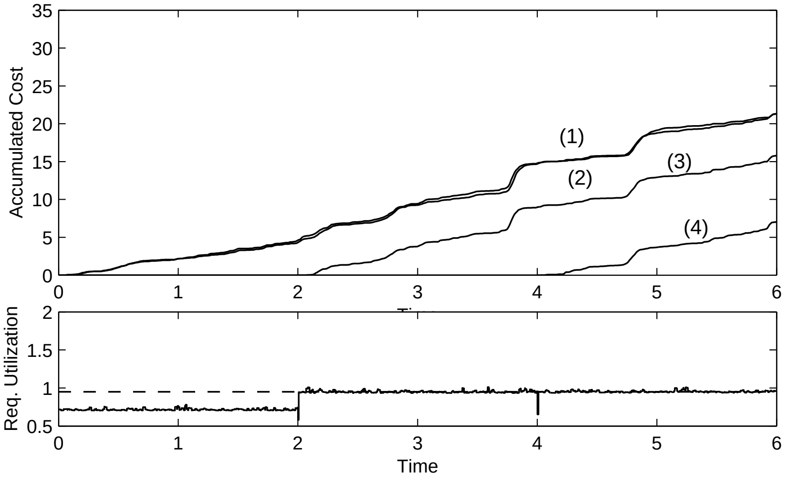


Figure 5.8 Accumulated costs and requested utilization under CS scheduling.

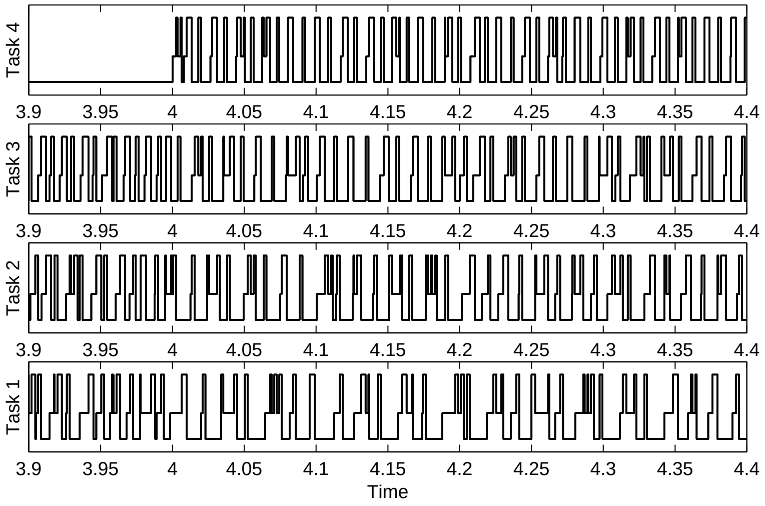


Figure 5.9 Close-up of schedule at $t = 4$ under CS scheduling.

5.6 Implementation

As a proof of concept, the computational model has been implemented in the public-domain real-time kernel STORK [Andersson and Blomdell, 1991], developed at the Department of Automatic Control, Lund Institute of Technology. The original kernel is a standard priority-preemptive real-time kernel written in Modula-2, running on multiple platforms. For this project, the Motorola PowerPC was chosen because of its high clock resolution (40 ns on a 100 MHz processor).

The kernel was modified to use EDF as the basic scheduling policy, and high-resolution timers (hardware clock interrupts that trigger user-defined handlers) were introduced. For tracing purposes, the kernel measures the execution-time of each task. An outline of the kernel code is shown in Listing 5.1.

A number of data structures for CBS servers, CS tasks, segments, inputs, and outputs, etc., were introduced, see the UML diagram in Figure 5.10. The tasks in the ready queue are sorted according to their absolute deadlines. Tasks that are associated with a CBS inherit the deadline

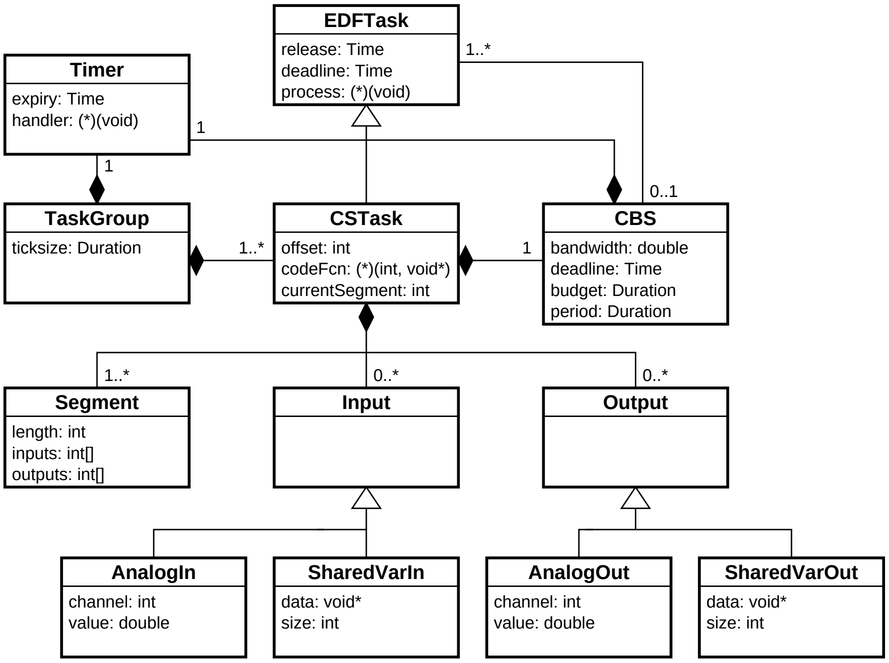


Figure 5.10 The various data structures in the implementation.

Listing 5.1 Pseudo code for the modified real-kernel.

```
void schedule() {   // Called by timer interrupt handler
  now = PowerPC.GetTB();     // Read hardware clock
  exectime = now - lastTime;
  if (task is associated with a CBS) {
    Decrease CBS budget by exectime;
    if (budget <= 0) {
      Update budget and deadline according to the rules;
    }
  }
  for (each timer in the timer queue) {
    if (now >= expiry) {
      Run handler;
    }
  }
  for (each task in the time queue) {
    if (now >= release) {
      Move task to ready queue;
      if (task is associated with a CBS) {
        Update budget and deadline according to the rules;
      }
    }
  }
  Make the first task in the ready queue the running task;
  if (task is associated with a CBS) {
    Set up CBS timer;
  }
  Determine next wake-up time (check timer and time queues);
  Set up new timer interrupt;
  lastTime = PowerPC.GetTB();   // Read hardware clock
  Record context switch in log; // For schedule traces
  Transfer control to the running task;
}
```

of the CBS. Note that several tasks may be served by the same CBS.

Each CBS is implemented using a timer. When a served task starts to execute, the expiry time of the timer is set to the time when the budget is expected to be exhausted. When the CBS is preempted or idle, the timer is disabled. A CBS that is associated with a CS task uses the segment information to determine how much the budget should be recharged and how much the deadline should be postponed.

Listing 5.2 Pseudo code for the task group timing.

```
for (each task in the task group) {
  if (current segment is finished) {
    Write outputs; // (if any)
    Increase segment counter;
  }
}
for (each task in the task group) {
  if (a new segment should begin) {
    Read inputs;  // (if any)
    Release segment job;  // signal semaphore
  }
}
Determine next interrupt time;
Set up timer;
```

Task Group Timing

For synchronization reasons, communicating CS tasks must share a common timebase and are gathered in task groups. Each task group uses a timer to trigger the reading of inputs, writing of outputs, and release of segments of tasks within the group. The structure of the task group timer interrupt handler is shown in Listing 5.2.

Associated with each CS task is a semaphore that is used to handle the release of the segment jobs. Internally, every CS task is implemented as a simple loop, see Listing 5.3.

API

The kernel provides a number of primitives for defining task groups, EDF tasks, CBS servers, CS tasks, inputs and outputs, etc. The code of a CS task is written according to a special format, here illustrated with a PID controller (written in Modula-2), see Listing 5.4. In the code, the ker-

Listing 5.3 Pseudo code for the internal implementation of a CS task.

```
while (true) {
  Increase segment counter;
  Wait on semaphore;
  Call codeFcn(segment,data);
}
```

Listing 5.4 Code function in Modula-2 representing a CS task.

```
PROCEDURE PIDTask(segment: CARDINAL; data: PIDData);
VAR r, y, u: LONGREAL;
BEGIN
   CASE segment OF
    1: r := ReadInput(1);
       y := ReadInput(2);
       u := PID.CalculateOutput(data, r, y);
       WriteOutput(1, u);
       |
    2: PID.UpdateState(data);
  END;
END PIDTask;
```

nel primitives `ReadInput` and `WriteOutput` are used to access the inputs and outputs associated with the segment. Note that structure of the code function is very similar to the code functions used in TrueTime (see Chapter 7).

5.7 Control Experiments

Some control experiments were performed on the ball and beam process, see Figure 5.11. The control objective is to move the ball to a given position on the beam. The input to the process is the beam motor voltage, and the

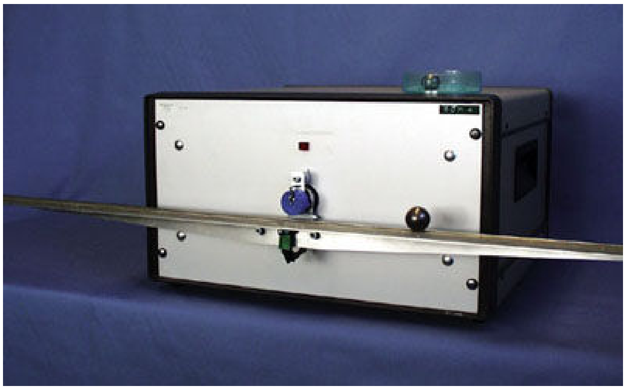


Figure 5.11 The ball and beam process.

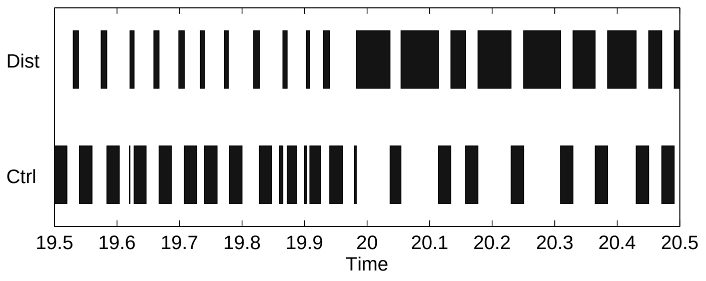


Figure 5.12 Execution trace under EDF scheduling. At $t = 20$, the disturbance task starts to misbehave, causing the controller task to miss its deadlines.

outputs are voltages representing the beam angle and the ball position. The process is controlled with a cascaded PID controller structure. To keep the example simple, the controller is implemented as a single task.

The controller is designed with the sampling interval $T_1 = 40$ ms and the assumed execution time is $C_1 = 20$ ms, thus consuming $U_1 = 0.5$ of the CPU (to generate a high CPU load, busy cycles were inserted in the task code). The code is divided into two segments: Calculate Output (5 ms) and Update State (15 ms).

Also executing in the system is a sporadic task with a minimum inter-arrival time of $T_2 = 20$ ms and an assumed WCET of $C_2 = 10$ ms. Between time 0 and 20, the actual execution time varies randomly between 5 and 10 ms. At time $t = 20$, the disturbance task starts to misbehave and has an execution time that varies randomly between 5 and 50 ms.

The behavior of the real-time control system under ordinary EDF scheduling and under CS scheduling was compared in different experiments. In each experiment, the execution trace (i.e, the schedule) was logged, together with the process measurements.

A trace of the task execution under ordinary EDF scheduling is shown in Figure 5.12. The sporadic task introduces jitter in the control task, and after time $t = 20$ the sampling interval is much longer due to the interference. The corresponding control performance is shown in Figure 5.13. The jitter causes a slight performance degradation, and after time $t = 20$ the performance deteriorates further (although the system is still stable).

Next, running the tasks under CS scheduling, both tasks were assigned a CPU share of 50%. The controller execution is no longer disturbed by the misbehaving sporadic task (see Figure 5.14), and (not visible in the trace) there is no longer any I/O jitter. The resulting improved control performance is shown in Figure 5.15. The performance is identical both before and after time $t = 20$.

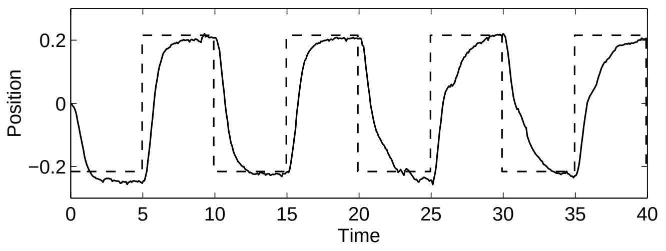


Figure 5.13 Control performance under EDF scheduling. After $t = 20$, the controller cannot execute as often as it should, causing the performance to degrade.

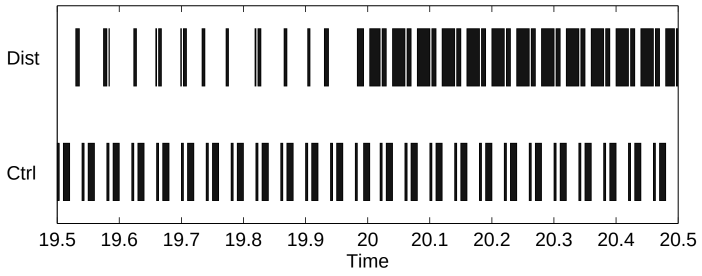


Figure 5.14 Execution trace under CS scheduling. The execution of the controller is unaffected by the disturbance task, thanks to the CBS servers used.

5.8 Conclusion

This chapter has presented the Control Server model, suitable for the implementation of control tasks in flexible real-time systems. Features of the model include small latency and jitter, and isolation between unrelated tasks.

The work can be extended in several directions. The CBS servers used

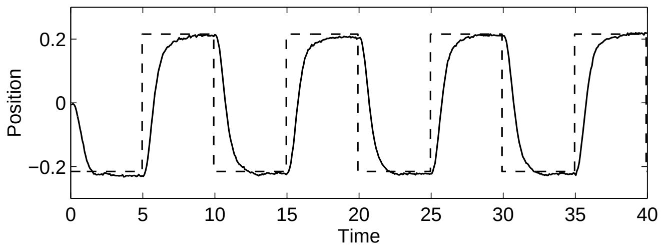


Figure 5.15 Control performance under CS scheduling. The performance is identical before and after $t = 20$.

could be modified to use a slack stealing algorithm such as CASH [Caccamo *et al.*, 2000a] or GRUB [Lipari and Baruah, 2000]. This could improve the performance further when the system is under-utilized.

We do not account for the interrupt time (including the I/O operation) in the scheduling analysis. Possibilities for more detailed analysis are found in [Liu and Layland, 1973] ("mixed scheduling") and in [Jeffay and Stone, 1993].

6

Analysis Using Jitterbug

6.1 Introduction

This chapter describes the MATLAB-based toolbox Jitterbug, which facilitates the computation of a quadratic performance criterion for a linear control system under various timing conditions. The tool helps to quickly assert how sensitive a control system is to delays, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate for instance jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. The toolbox is built upon known theory (sampled-data control theory, e.g., [Åström and Wittenmark, 1997], and jump linear systems [Krasovskii and Lidskii, 1961; Ji *et al.*, 1991]). Its main contribution is to make it easy to apply this type of stochastic analysis to a wide range of problems.

Related Work

Much of the inspiration for Jitterbug comes from [Nilsson, 1998a], which deals with LQG-control over communication networks with random delays. In conjunction with that work, two simple MATLAB toolboxes for control design and analysis were produced [Nilsson, 1998b]. The toolboxes assume a system description with periodic sampling and two random delays in the the control loop (see Figure 2.4). A discrete-time cost function may be evaluated, provided that the user formulates the closed-loop system. By constrast, our toolbox allows more general system and timing descriptions, with arbitrary numbers of subsystems and random delays. A *continuous-time* cost function is evaluated, taking the intersample behavior into account. Also, the toolbox allows for frequency domain analysis (in the form of discrete-time spectral density calculations).

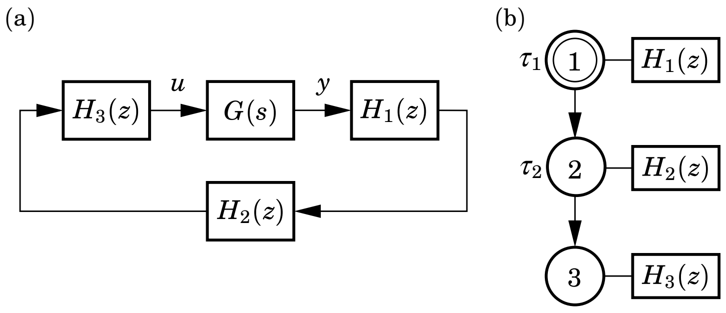


Figure 6.1 A simple Jitterbug model of a computer-controlled system: (a) signal model and (b) timing model.

6.2 System Description

In Jitterbug, a control system is described by two parallel models: a signal model and a timing model. The signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period.

An example of a Jitterbug model is shown in Figure 6.1, where a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system G, and the controller is described by the three discrete-time systems H_1, H_2, and H_3. The system H_1 could represent a periodic sampler, H_2 could represent the computation of the control signal, and H_3 could represent the actuator. The associated timing model says that, at the beginning of each period, H_1 should first be executed (updated). Then there is a random delay τ_1 until H_2 is executed, and another random delay τ_2 until H_3 is executed. The delays could model computational delays, scheduling delays, or network transmission delays.

Signal Model

The signal model consists of a number of inter-connected continuous-time and discrete-time linear systems driven by white noise. The cost is specified as a stationary, continuous-time quadratic cost function.

Continuous-Time Systems. A continuous-time system may be specified in state-space form or in transfer-function form.

In state-space form, the system is described by

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + v_c(t), & & \\ y^0(t) &= Cx(t), & &\text{(continuous output)} \\ y(t_k) &= y^0(t_k) + e_d(t_k), & &\text{(measured discrete output)} \end{aligned} \tag{6.1}$$

where v_c is a continuous-time white-noise process with zero mean and covariance[1] R_{1c}, and e_d is a discrete-time white-noise process with zero mean and covariance R_{2d}. The cost of the system is specified as

$$J_c = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ u(t) \end{pmatrix}^T Q_c \begin{pmatrix} x(t) \\ u(t) \end{pmatrix} dt, \tag{6.2}$$

where Q_c is a positive semi-definite matrix.

In transfer-function form, the system is described by

$$\begin{aligned} y^0(t) &= G(p)\big(u(t) + v_c(t)\big), & &\text{(continuous output)} \\ y(t_k) &= y^0(t_k) + e_d(t_k), & &\text{(measured discrete output)} \end{aligned} \tag{6.3}$$

where $G(p)$ is a strictly proper transfer function, v_c is a continuous-time white-noise process with zero mean and covariance R_{1c}, and e_d is a discrete-time white-noise process with zero mean and covariance R_{2d}. The cost of the system is specified as

$$J_c = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix}^T Q_c \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix} dt, \tag{6.4}$$

where Q_c is a positive semi-definite matrix.

Note that direct terms are not allowed (i.e., all continuous-time systems must be strictly proper). This restriction is imposed to avoid problems with infinite variances and algebraic loops. Also note that there is no *continuous-time* output noise. The ability to specify discrete-time measurement noise in connection with the plant is only offered as a convenience. The discrete-time output noise will be translated into input noise at any connected discrete-time system (see Figure 6.2(c) below).

Discrete-Time Systems. A discrete-time system may be given in state-space form or in transfer-function form.

[1]Strictly speaking, a continuous-time white noise process has infinite variance. What we really mean is that v_c has the spectral density $\phi(\omega) = \frac{1}{2\pi} R_{1c}$. See [Åström and Wittenmark, 1997] for further discussion.

In state-space form, the system is described by

$$\begin{aligned} x(t_{k+1}) &= \Phi x(t_k) + \Gamma u(t_k) + v_d(t_k), \\ y^0(t_k) &= Cx(t_k) + Du(t_k), \quad \text{(discrete output)} \\ y(t_k) &= y^0(t_k) + e_d(t_k), \quad \text{(measured discrete output)} \end{aligned} \tag{6.5}$$

where v_d and e_d are discrete-time white-noise processs with zero mean and covariance

$$R_d = \mathbf{E} \begin{pmatrix} v_d(t_k) \\ e_d(t_k) \end{pmatrix} \begin{pmatrix} v_d(t_k) \\ e_d(t_k) \end{pmatrix}^T .$$

The cost of the system is specified as

$$J_d = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ y^0(t) \\ u(t) \end{pmatrix}^T Q_d \begin{pmatrix} x(t) \\ y^0(t) \\ u(t) \end{pmatrix} dt, \tag{6.6}$$

where Q_d is a positive semi-definite matrix. Note that $x(t)$ and $y^0(t)$ are piecewise constant signals, while $u(t)$ may be a continuous signal.

In transfer-function form, the system is described by

$$\begin{aligned} y^0(t_k) &= H(q)\big(u(t_k) + v_d(t_k)\big), \quad \text{(discrete output)} \\ y(t_k) &= y^0(t_k) + e_d(t_k), \quad \text{(measured discrete output)} \end{aligned} \tag{6.7}$$

where $H(q)$ is a proper transfer function, and v_d and e_d are discrete-time white-noise processs with zero mean and covariance

$$R_d = \mathbf{E} \begin{pmatrix} v_d(t_k) \\ e_d(t_k) \end{pmatrix} \begin{pmatrix} v_d(t_k) \\ e_d(t_k) \end{pmatrix}^T .$$

The cost of the system is specified as

$$J_d = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix}^T Q_d \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix} dt, \tag{6.8}$$

where Q_d is a positive semi-definite matrix. Again, note that $y^0(t)$ is a piecewise constant signal, while $u(t)$ may be a continuous signal.

Connecting Systems. The total system is formed by appropriately connecting the inputs and outputs of a number of continuous-time and discrete-time systems. Throughout, multiple input-multiple output (MIMO)

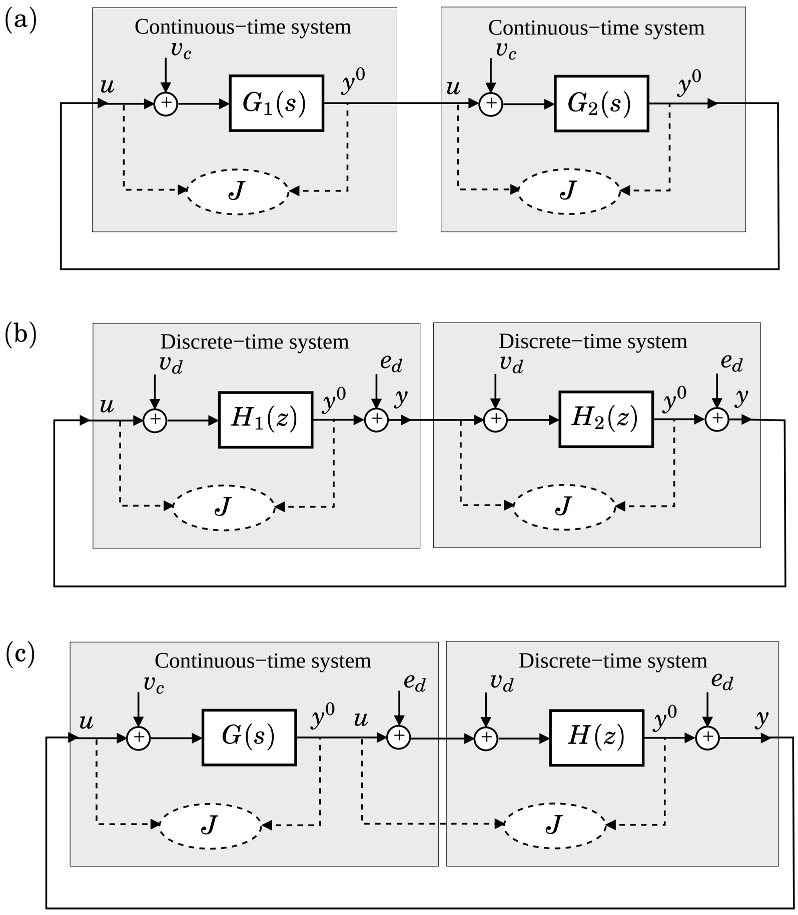


Figure 6.2 Possible interconnections of continuous-time and discrete-time systems.

formulations are allowed, and a system may collect its inputs from a number of other systems. The total cost to be evaluated is summed over all continuous-time and discrete-time systems:

$$J = \sum J_c + \sum J_d. \tag{6.9}$$

It is important to understand how cost and noise are handled when systems are interconnected. Three principal cases can be distinguished (see Figure 6.2):

(a) The interconnection of two continuous-time systems. Note that any discrete-time output noise e_d will be ignored.

(b) The interconnection of two discrete-time systems. No surprises here.

(c) The interconnection of a continuous-time and a discrete-time system. Note that the discrete-time output noise e_d will not be included in the input cost of the discrete-time system.

Timing Model

The timing model consists of a number of timing nodes. Each node can be associated with zero or more discrete-time systems in the signal model, which should be updated when the node becomes active. At time zero, the first node is activated. The first node can be declared to be *periodic* (indicated by an extra circle in the illustrations), which means that the execution will restart at this node every h seconds. This is useful for modeling periodic controllers. It also greatly simplifies the cost calculations, allowing a direct solution method to be used.

Each node is associated with a time delay τ, which must elapse before the next node can become active. (If unspecified, the delay is assumed to be zero.) The delay can be used to model computational delay, transmission delay in a network, etc. A delay is described by a finite discrete-time probability density function, specified by a vector

$$P_\tau = \begin{pmatrix} P_\tau(0) & P_\tau(1) & P_\tau(2) & \dots \end{pmatrix}, \tag{6.10}$$

where $P_\tau(k)$ represents the probability of a delay of $k\delta$ seconds. The time grain δ is a constant that is specified for the whole model.

In periodic systems, the execution is preempted if the total delay $\sum \tau$ in the system exceeds the period h. Any remaining timing nodes will be skipped. This models a real-time system where hard deadlines (equal to the period) are enforced and the control task is aborted at the deadline.

An aperiodic system can be used to model a real-time system where the task periods are allowed to drift if there are overruns. It could also be used to model a controller that samples "as fast as possible" instead of waiting for the next period. A disadvantage of aperiodic models is that much longer computation times are needed, since an iterative solver must be used.

Time-Dependent Delays. A delay distribution may be dependent on the time since the most recent activation of the first node. The delay is then described by a matrix

$$P_\tau = \begin{pmatrix} P_\tau(0,0) & P_\tau(0,1) & \dots \\ P_\tau(1,0) & P_\tau(1,1) & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}, \tag{6.11}$$

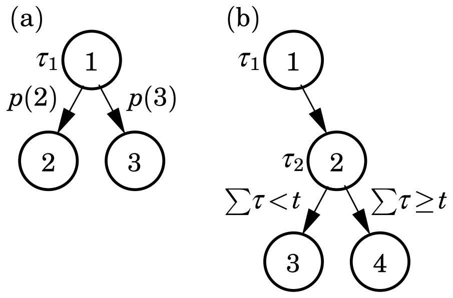


Figure 6.3 Alternative execution paths in a Jitterbug execution model: (a) random choice of path and (b) choice of path depending on the time since the most recent activation of the first node.

where $P_\tau(j, k)$ represents the probability of a delay of $k\delta$ seconds given a previous total delay of $j\delta$ seconds.

Node- and Time-Dependent Update Equations. The same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations (i.e., different Φ, Γ, C and D matrices) in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. An example of this type is given later.

It is also possible to make the update equations depend on the time since the first node last became active. This can be used to model jitter-compensating controllers for example.

Random and Time-Dependent Execution Paths. For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In Jitterbug, two such cases can be modeled (see Figure 6.3):

(a) A vector n of next nodes can be specified with a probability vector p. After the delay, execution node $n(i)$ will be activated with probability $p(i)$. This can be used to model a sample being lost with some probability.

(b) A vector n of next nodes can be specified with a time vector t. If the time since the most recent activation of the first node exceeds $t(i)$, node $n(i)$ will be activated next. This can be used to model time-outs and various compensation schemes.

6.3 Internal Workings

Inside Jitterbug, the states and the cost are considered in continuous time. The inherently discrete-time states, e.g., in discrete-time controllers or filters, are treated as continuous-time states with zero dynamics. This means that the total system can be written as

$$\dot{x}(t) = Ax(t) + v_c(t), \tag{6.12}$$

where x collects all the states in the system, and v_c is continuous-time white noise process with covariance R_c. To model the discrete-time changes of some states as a timing node n is activated, the state is instantaneously transformed by

$$x(t^+) = E_n x(t) + e_n(t), \tag{6.13}$$

where e_n is a discrete-time white noise process with covariance W_n.

The total cost (6.9) for the system can be written as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T x^T(t) Q_c x(t)\, dt, \tag{6.14}$$

where Q_c is a positive semidefinite matrix.

Sampling the System

Jitterbug relies on discretized time to calculate the variance of the states and the cost. No approximations are involved, however. Sampling the system (6.12) with a period of δ (the time-grain in the delay distributions) gives

$$x(k\delta + \delta) = \Phi x(k\delta) + v(k\delta), \tag{6.15}$$

where the covariance of v is R, and the cost (6.14) becomes

$$J = \lim_{N\to\infty} \frac{1}{N\delta} \sum_{k=0}^{N-1} \Big(x^T(k\delta) Q x(k\delta) + q \Big). \tag{6.16}$$

The matrices Φ, R, Q, and q are calculated as

$$\Phi = e^{A\delta}, \tag{6.17}$$

$$R = \int_0^\delta e^{A(\delta-\tau)} R_c e^{A^T(\delta-\tau)}\, d\tau, \tag{6.18}$$

$$Q = \int_0^\delta e^{A^T t} Q_c e^{At}\, dt, \tag{6.19}$$

$$q = \operatorname{tr}\Big(Q_c \int_0^\delta \int_0^\delta e^{A(t-\tau)} R_c e^{A^T(t-\tau)}\, d\tau\, dt \Big), \tag{6.20}$$

or, equivalently, from

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} = \exp\left(\begin{pmatrix} -A^T & Q_c \\ 0 & A \end{pmatrix} \delta \right), \tag{6.21}$$

and

$$\begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{pmatrix} = \exp\left(\begin{pmatrix} -A & I & 0 \\ 0 & -A & R_c^T \\ 0 & 0 & A^T \end{pmatrix} \delta \right), \tag{6.22}$$

so that

$$\Phi = P_{22}, \tag{6.23}$$

$$Q = P_{22}^T P_{12}, \tag{6.24}$$

$$R = M_{33}^T M_{23}, \tag{6.25}$$

$$q = \operatorname{tr}\big(Q M_{33}^T M_{13}\big). \tag{6.26}$$

Timing Representation

Internally, the timing model is translated into a Markov chain. A simple example where two timing nodes are translated into a number of Markov nodes is shown in Figure 6.4. The Markov state n keeps track of the current timing node and the number of time steps since the most recent activation of the first node. A delay between two timing nodes is represented by a number of intermediate nodes. When we move to the right in the Markov chain, i.e., when time progesses, the states of the system change according to the sampled continuous dynamics in (6.15). The state

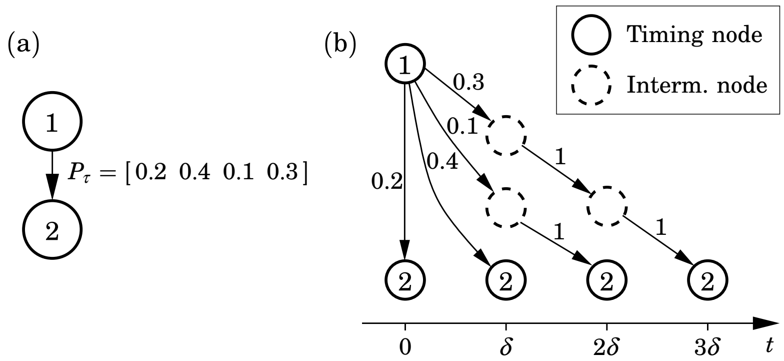


Figure 6.4 Example of a simple timing model (a) being transformed into a Markov chain (b). The random delay between the two timing nodes is modeled by a number of intermediate nodes.

covariance $P(k\delta) = \mathbf{E}\left\{x(k\delta)x^T(k\delta)\right\}$ then evolves as

$$P(k\delta + \delta) = \Phi P(k\delta)\Phi^T + R. \tag{6.27}$$

When a new timing node is visited, the states of the system change according to the discrete dynamics in (6.13). The state covariance then evolves as

$$P(k\delta^+) = E_n P(k\delta) E_n^T + W_n, \tag{6.28}$$

where W_n is the covariance of the discrete-time noise $e_n(k\delta)$ in node n. Combining the above, we define Φ_n as

$$\Phi_n = \begin{cases} \Phi & \text{if } n \text{ is an intermediate node,} \\ E_n\Phi & \text{if } n \text{ is a timing node,} \end{cases} \tag{6.29}$$

and similarly R_n as

$$R_n = \begin{cases} R & \text{if } n \text{ is an intermediate node,} \\ E_n R E_n^T + W_n & \text{if } n \text{ is a timing node.} \end{cases} \tag{6.30}$$

(Note that the expressions above are not valid for the case of zero delay between two timing nodes—that case must be treated separately.)

Calculating Variance and Cost

Now consider all possible Markov states simultaneously. Let $\pi_n(k\delta)$ be the probability of being in Markov state n at time $k\delta$, and let $P_n(k\delta)$ be the covariance of the state if the system is in Markov state n at time $k\delta$. Furthermore, let the transition matrix of the Markov chain be σ, such that

$$\pi(k\delta + \delta) = \sigma\pi(k\delta). \tag{6.31}$$

The state covariance then evolves as

$$P_n(k\delta + \delta) = \sum_i \sigma_{ni}\pi_i(k\delta)\Big(\Phi_n P_i(k\delta)\Phi_n^T + R_n\Big), \tag{6.32}$$

and the cost in time step k is given by

$$\frac{1}{\delta}\sum_n \pi_n(k\delta)\Big(\operatorname{tr}\big(P_n(k\delta)Q\big) + q\Big). \tag{6.33}$$

For systems without a periodic node, Eq. (6.32) must be iterated until the cost and variance converge. For periodic systems, the Markov state returns to the periodic timing node every h/δ time steps. Since Eq. (6.32) is affine in P, we can find the stationary covariance $P_1(\infty)$ in the periodic node by solving a linear system of n^2 equations, where n is the number of states in the total system (see [Nilsson, 1998a]). The total cost is then calculated over the timesteps in one period. The toolbox returns the cost $J = \infty$ if the system is not stable (in the mean-square sense).

Calculating Spectral Densities

For periodic systems, the toolbox can compute the discrete-time spectral densities of all outputs as observed in the periodic timing node. This can be used for frequency-domain analysis of the closed-loop system. The spectral density of an output y is defined as

$$\phi_y(\omega) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} r_y(k) e^{-ik\omega}, \tag{6.34}$$

where $r_y(k)$ is the covariance function of y. This function is computed as

$$\begin{aligned} r_y(k) &= \mathbf{E}\left\{y(t)y^T(t+kh)\right\} = \mathbf{E}\left\{Cx(t)x^T(t+kh)C^T\right\} \\ &= \mathbf{E}\left\{C\bar{\Phi}^{|k|}x(t)x^T(t)C^T\right\} = C\bar{\Phi}^{|k|}P_1(\infty)C^T, \end{aligned} \tag{6.35}$$

where $\bar{\Phi}$ is the average transition matrix over a period, and $P_1(\infty)$ is the stationary covariance in the periodic node. The spectral density is returned as a linear system $F(z)$ such that $\phi_y(\omega) = F(e^{i\omega})$.

6.4 Examples

In this section, a number of examples that illustrate the use of Jitterbug are given. Note that there is no graphical user interface to Jitterbug; rather, the timing model and the signal model are defined in a script using a number of MATLAB commands. Typically, the script is iterated over a number of interesting timing parameters to produce a cost function plot. A summary of the available commands is given in Table 6.1.

Example 1: Distributed Control System

In this example, we will study the distributed control system shown in Figure 2.4. In the control loop, the sensor, the actuator, and the controller are distributed among different nodes in a network. The sensor node is assumed to be time-driven, whereas the controller and actuator nodes are assumed to be event-driven. At a fixed period h, the sensor samples the

Table 6.1 Summary of the Jitterbug commands.

Command	Description
`initjitterbug`	Initialize a new Jitterbug system.
`addtimingnode`	Add a timing node.
`addcontsys`	Add a continuous-time system.
`adddiscsys`	Add a discrete-time system to a timing node.
`adddiscexec`	Add an execution of a previously defined discrete-time system.
`adddisctimedep`	Add time-dependence to a previously defined discrete-time system.
`calcdynamics`	Calculate the internal dynamics of a Jitterbug system.
`calccost`	Calculate the total cost of a Jitterbug system and, for periodic systems, calculate the spectral densities of the outputs.
`lqgdesign`	Design a discrete-time LQG controller for a continuous-time plant with a constant time delay and a continuous-time cost function.

process and sends the measurement sample over the network to the controller node. There, the controller computes a control signal and sends it over the network to the actuator node, where it is subsequently actuated.

The Jitterbug model of the system was shown in Figure 6.1. The process to be controlled is a DC servo, described by the continuous-time system

$$G(s) = \frac{1000}{s(s+1)}. \tag{6.36}$$

The process is driven by white continuous-time input noise. The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1. \tag{6.37}$$

The process is regulated by a discrete-time PD controller given by

$$H_2(z) = K\left(1 + \frac{T_d}{h}\frac{z-1}{z}\right), \tag{6.38}$$

where $K = 1.5$ and $T_d = 0.035$.

In the timing model, the communication delays are modeled the two random variables τ_1 $(= \tau_{sc})$ and τ_2 $(= \tau_{ca})$. The input-output latency of the controller is thus given by $L_{io} = \tau_1 + \tau_2$. It is assumed that L_{io} never exceeds h (otherwise Jitterbug would skip the remaining updates).

Listing 6.1 MATLAB commands for a Jitterbug cost calculation.

```
G = 1000/(s*(s+1));               % Define the process
H1 = 1;                           % Define the sampler
H2 = -K*(1+Td/h*(z-1)/z);         % Define the controller
H3 = 1;                           % Define the actuator
Ptau1 = [ ... ];                  % Define prob. distr. 1
Ptau2 = [ ... ];                  % Define prob. distr. 2
N = initjitterbug(delta,h);       % Set time-grain and period
N = addtimingnode(N,1,Ptau1,2);   % Define timing node 1
N = addtimingnode(N,2,Ptau2,3);   % Define timing node 2
N = addtimingnode(N,3);           % Define timing node 3
N = addcontsys(N,1,G,4,Q,R1,R2);  % Add plant + cost and noise
N = adddiscsys(N,2,H1,1,1);       % Add sampler to node 1
N = adddiscsys(N,3,H2,2,2);       % Add controller to node 2
N = adddiscsys(N,4,H3,3,3);       % Add actuator to node 3
N = calcdynamics(N);              % Calculate internal dynamics
J = calccost(N);                  % Calculate the total cost
```

The performance of the controller is evaluted using the cost function

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \big(y^2(t) + u^2(t)\big)\, dt. \tag{6.39}$$

An outline of the MATLAB commands needed to specify the model and compute the value of the cost function are given in Listing 6.1.

Sampling Period and Constant Input-Ouput Latency. A control system can typically give satisfactory performance over a range of sampling periods. In textbooks on digital control (e.g., [Åström and Wittenmark, 1997]), rules of thumb for sampling period selection are often given. One such rule suggests that the sampling interval h should be chosen such that $0.2 < \omega_b h < 0.6$, where ω_b is the bandwidth of the closed-loop system. In our case, a continuous-time PD controller with the given parameters would give a bandwidth of about $\omega_b = 80$ rad/s. This would imply a sampling period of between 2.5 and 7.5 ms. The effect of input-output latency is typically not considered in such rules of thumb, however. Using Jitterbug, the combined effect of sampling period and latency can be easily investigated. In Figure 6.5, the cost function (6.39) for the distributed control system has been evaluated for different sampling periods in the interval 1 to 10 milliseconds, and for constant input-output latency ranging from 0 to 100% of the sampling interval. As can be seen, a one-sample

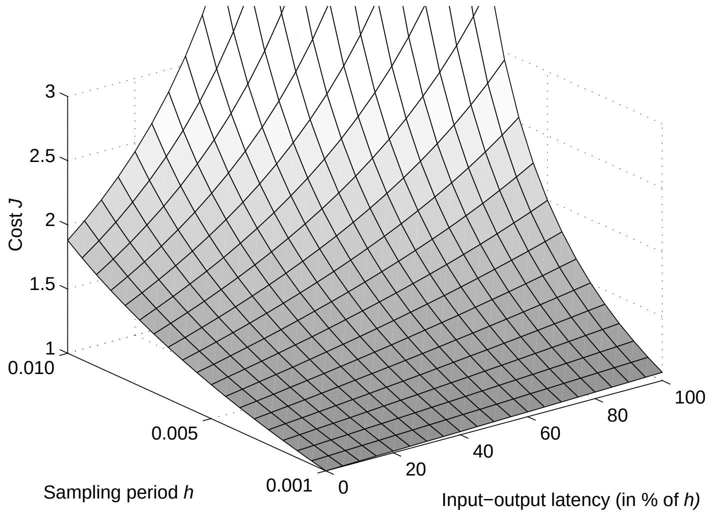


Figure 6.5 Example of a cost function computed using Jitterbug. The plot shows the cost as a function of sampling period and a constant input-output latency in the distributed control system example.

delay gives negligible performance degradation when $h = 1$ ms. When $h = 10$ ms, a one-sample delay makes the system unstable (i.e., the cost J goes to infinity).

Random Delays and Jitter Compensation. If system resources are very limited (as they often are in embedded control applications), the control engineer may have to live with long sampling intervals. Delay in the control loop then becomes a serious issue. Ideally, the delay should be accounted for in the control design. In many practical cases, however, even the mean value of the delay will be unknown at design time. The actual delay at run-time will vary from sample to sample due to real-time scheduling, the load of the system, etc. A simple approach is to use gain scheduling—the actual delay is measured in each sample and the controller parameters are adjusted according to precalculated values that have been stored in a table. Since Jitterbug allows time-dependent controller parameters, such delay compensation schemes can also be analyzed using the tool.

In the Jitterbug model of the distributed control system, we now assume that the delays τ_1 and τ_2 are uniformly distributed random variables between 0 and $L_{io}^{max}/2$, where L_{io}^{max} is the maximum input-output

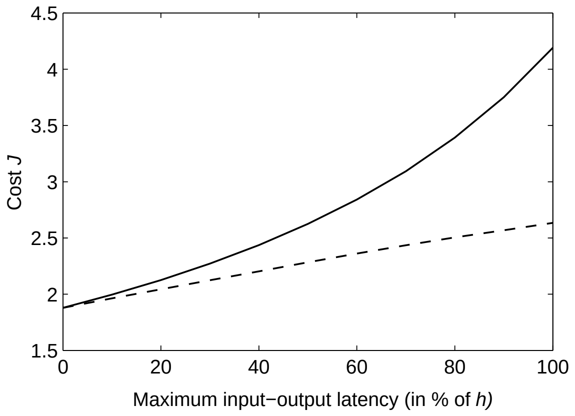


Figure 6.6 Cost as a function of maximum latency in the distributed control system example with random delays: no delay compensation (full), and dynamic delay compensation (dashed).

latency. A range of PD controller parameters (ranging from $K = 1.5$ and $T_d = 0.035$ for zero delay to $K = 0.78$ and $T_d = 0.052$ for 7.5 ms delay) are derived and stored in a table. When a sample arrives at the controller node, only the delay τ_1 from sensor to controller is known, however, so the remaining delay is predicted by its expected value of $L_{io}^{max}/4$.

The sampling interval is set to $h = 10$ ms to make the effects of latency and jitter clearly visible. In Figure 6.6, the cost function (6.39) has been evaluated with and without delay compensation for values of the maximum latency ranging from 0 to 100% of the sampling interval. The cost increases much more rapidly for the uncompensated system. The distributed control example will be studied in more detail later using TrueTime (see Chapter 7).

Example 2: Lost Samples in Notch Filters

As a second example, we will look at a signal processing application. Cleaning signals from disturbances using notch filters is important in many control systems. In some cases, the filters are very sensitive to lost samples due to their narrow-band frequency characteristics, and in real-time systems lost samples are sometimes inevitable. In this example, Jitterbug is used to evaluate the effects of lost samples in different filters and possible compensation techniques.

The setup is as follows. A good signal x (modeled as low-pass filtered noise) is to be cleaned from an additive disturbance e (modeled as band-pass filtered noise), see the signal spectra in Figure 6.7. An estimate $\hat{x}$ of

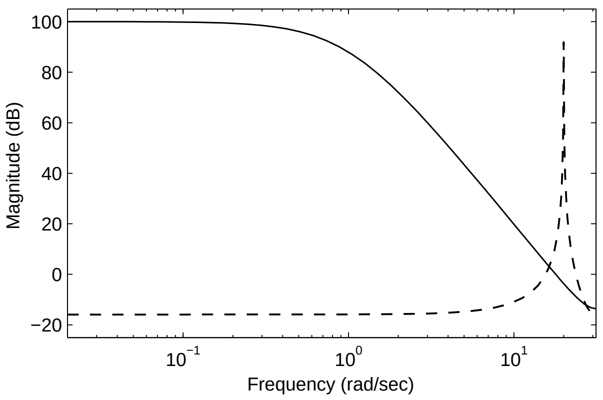


Figure 6.7 The spectral densities of the good signal x (full) and the disturbance e (dashed).

the good signal should be found by applying a digital notch filter with the sampling interval $h = 0.1$ to the measured signal $x + e$. Unfortunately, a fraction p of the measurements are lost.

A Jitterbug model of the system is shown in Figure 6.8. The signals x and e are generated by filtered continuous-time white noise through the two continuous-time systems G_1 and G_2. The digital filter is represented as two discrete-time systems: *Samp* and *Filter*. The good signal x is buffered in the system *Delay* and is then compared to the filtered estimate $\hat{x}$ in the system *Diff*, producing the estimation error $\tilde{x} = x - \hat{x}$.

In the execution model, there is a probability p that the *Samp* system will not be updated. In that case, an alternate version, *Filter*(2), of the filter dynamics can be executed and used to compensate for the lost sample.

Two different filters are compared. The first filter is an ordinary second-order notch filter with two zeros on the unit circle. It is updated with the same equations even if no sample is available. The second filter is a Kalman filter, which is based on a simplified model of the signal dynamics. In the case of a lost sample, only prediction is performed in the Kalman filter.

The performance of the filters is evaluated using the cost function

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \tilde{x}^2(t)\, dt,$$

which measures the variance of the estimation error. In Figure 6.9, the cost has been plotted for different probabilities of lost samples. The figure

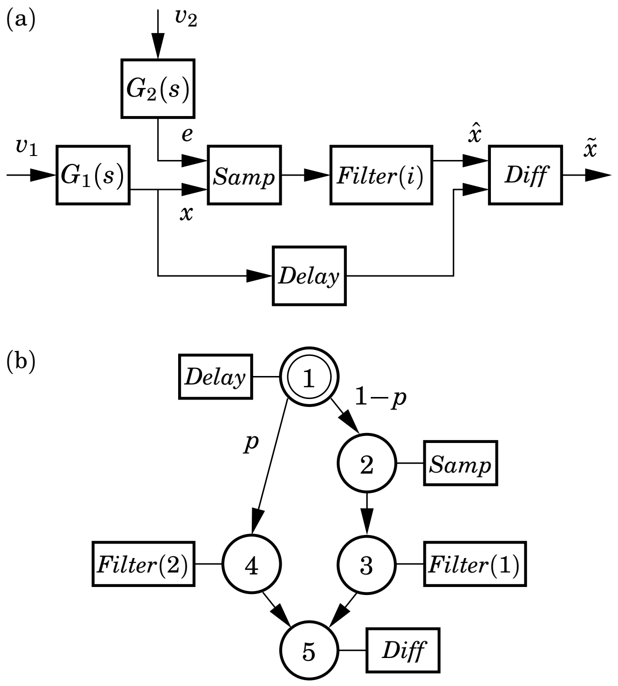


Figure 6.8 Jitterbug model of the signal processing application: (a) signal model and (b) timing model.

shows that the ordinary notch filter performs better in the case of no lost samples, but the Kalman filter performs better as the probability of lost samples increases. This is because the Kalman filter can perform prediction when no sample is available.

Assuming $p = 0.1$, the spectral density of the estimation error $\tilde{x}$ for the different filters is shown in Figure 6.10. It is seen that the ordinary notch filter performs well around the disturbance frequency while the lost samples introduce a large error at lower frequencies. The Kalman filter is less sensitive towards lost samples and has a more even error spectrum.

Example 3: Multirate Control Server Task

In this example, we calculate the performance of a multirate ball and beam controller executing under the Control Server model, see Section 5.4. The block diagram of the cascaded control structure was shown in Figure 5.5. The outer process dynamics (the ball position) is given by

$$G_1(s) = -\frac{7.0}{s^2}, \tag{6.40}$$

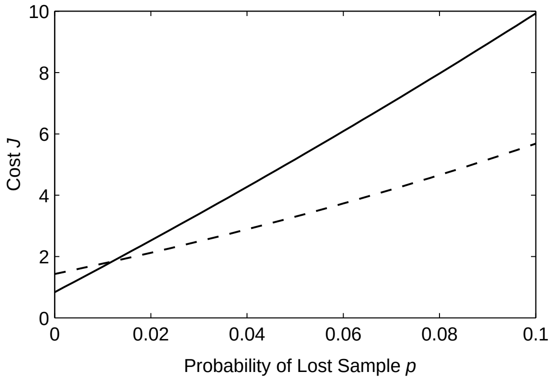


Figure 6.9 The variance of the estimation error in the different filters as a function of the probability of lost samples: notch filter (full) and time-varying Kalman filter (dashed).

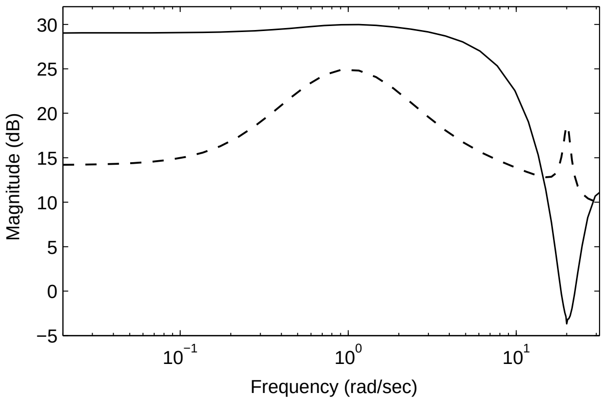


Figure 6.10 The spectral density of the error output $\tilde{x}$ when 10% of the samples are lost, using a notch filter (full) or a time-varying Kalman filter (dashed).

while the inner dynamics (the beam angle) is given by

$$G_2(s) = \frac{4.4}{s}. \tag{6.41}$$

The process is controlled by two PID controllers, implemented according to Eq. (4.5). The parameters of the outer controller are $K = -0.2$, $T_i = 10$, $T_d = 1$, and $N = 10$. The inner controller is a pure P-controller with $K = 2$.

The Jitterbug model of the multirate controller is shown in Figure 6.11. The sampling operations of the controllers are represented by the discrete-

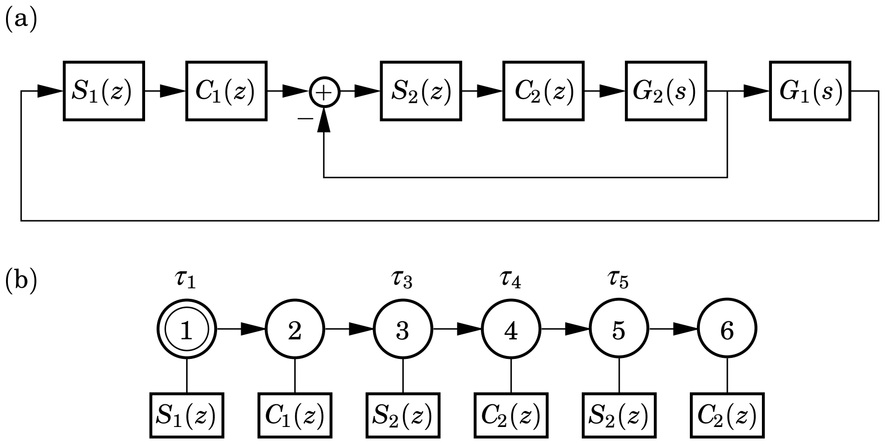


Figure 6.11 Jitterbug model of the multirate ball and beam controller executing under the Control Server model: (a) signal model, and (b) timing model. The inner controller (S_2 and C_2) is updated twice in each period.

time systems $S_1(z)$ and $S_2(z)$, while the control computations and actuations are represented by $C_1(z)$ and $C_2(z)$.

The timing model is obtained by inspection of the segment layout in Figure 5.7. The period of the timing model is equal to the outer sampling period h, and the ticksize is set to $\delta = h/8$. The delays in the model are constant and are given by $\tau_1 = 2\delta$, $\tau_3 = \delta$, $\tau_4 = 3\delta$,$\tau_5 = \delta$.

Assume that the execution time of the PID algorithm is $C = 10$ ms (even if the D and I parts are not used). During each outer period h, the control algorithm is executed three times. Given a utilization factor, the sampling period is thus given by $h = 0.03/U$. The performance of the controller is measured by the cost function

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \left(y_1^2(t) + y_2^2(t) + u^2(t)\right) dt, \tag{6.42}$$

where y_1 is the ball position, y_2 is the beam angle, and u is the inner control signal. The cost J as a function of the CPU share U is shown in Figure 6.12. The complete script for the cost computation is shown in Listing 6.2.

Example 4: Overrun Handling Methods

In this example, we study the problem of period overruns in control tasks. It is assumed that the response time of a control task varies randomly (due to, e.g., variations in the control algorithm execution time or preemp-

Listing 6.2 Jitterbug multirate cost calculation.

```
s = tf('s'); z = tf('z');
G1 = -7/s^2; G2 = 4.4/s;             % Process
S1 = 1; S2 = [1 -1];                 % Samplers
C2 = 2;                              % Inner controller
Q1 = diag([1 0]); Q2 = diag([1 1]);  % Cost
R1 = 1; R2 = 1;                      % Noise
K = -0.2; Ti = 10; Td = 1; Nd = 10;  % Outer PID parameters
Ptau1 = [0 0 1];                     % tau1 = 2*delta
Ptau3 = [0 1];                       % tau3 = 1*delta
Ptau4 = [0 0 0 1];                   % tau4 = 3*delta
Ptau5 = [0 1];                       % tau5 = 1*delta
Uvec = logspace(-2,0,50); Jvec = [];

for U = Uvec
  h = 0.03/U;                        % Outer sampling interval
  delta = h/8;                       % Time grain
  sp = (z-1)/(z*h);                  % Backwards approx.
  C1 = -K*(1+1/(Ti*sp)+sp*Td/(1+sp*Td/Nd)); % Outer controller
  N = initjitterbug(delta,h);        % Initialize Jitterbug
  N = addtimingnode(N,1,Ptau1,2);    % Add timing node 1
  N = addtimingnode(N,2,[1],3);      % Add timing node 2
  N = addtimingnode(N,3,Ptau3,4);    % Add timing node 3
  N = addtimingnode(N,4,Ptau4,5);    % Add timing node 4
  N = addtimingnode(N,5,Ptau5,6);    % Add timing node 5
  N = addtimingnode(N,6);            % Add timing node 6
  N = addcontsys(N,1,G2,6,Q2,R2);    % Add inner process
  N = addcontsys(N,2,G1,1,Q1,R1);    % Add outer process
  N = adddiscsys(N,3,S1,2,1);        % Add sampler 1 to node 1
  N = adddiscsys(N,4,C1,3,2);        % Add contr. 1 to node 2
  N = adddiscsys(N,5,S2,[4 1],3);    % Add sampler 2 to node 3
  N = adddiscsys(N,6,C2,5,4);        % Add contr. 2 to node 4
  N = adddiscexec(N,5,[],[4 1],5);   % Add sampler 2 to node 5
  N = adddiscexec(N,6,[],5,6);       % Add contr. 2 to node 6
  N = calcdynamics(N);
  J = calccost(N);                   % Calculate total cost
  Jvec = [Jvec; J];
end

plot(Uvec,Jvec)
```

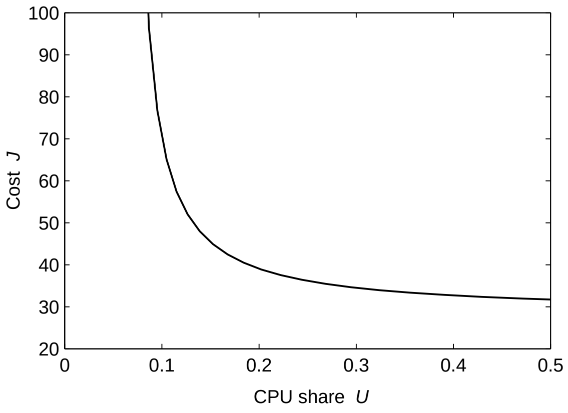


Figure 6.12 Control performance as a function of the CPU share for the multirate ball and beam controller.

tion from other tasks). The question is what to do if the control computation has not completed by the end of the sampling period. Two different approaches will be compared:

- The task is aborted at the end of the period. (This is the default behavior of a periodic timing model in Jitterbug.)
- The current period is extended until the task finishes. This will cause a shift in the following periods. (This case can be investigated using an aperiodic timing model.)

The plant to be controlled is described by an oscillatory third-order continuous-time system,

$$G(s) = \frac{1}{s(s^2 + 2\zeta\omega s + \omega^2)}, \tag{6.43}$$

where $\omega = 1$ and $\zeta = 0.2$. The plant is disturbed by white continuous-time input noise with unit variance and white discrete-time output noise with variance 0.001. A discrete-time LQG controller with the sampling interval $h = 0.25$ is designed to minimize the cost function

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \big(y^2(t) + u^2(t)\big)\, dt. \tag{6.44}$$

The Jitterbug model of the control system is shown in Figure 6.13. In the signal model, the controller is modeled by two discrete-time systems. The system H_1 represents the sampler, while H_2 represents the

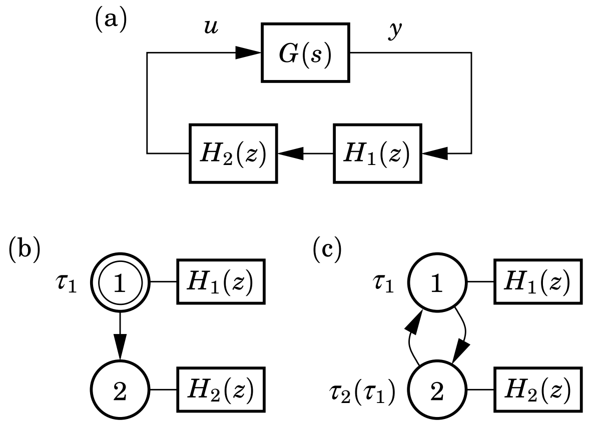


Figure 6.13 Jitterbug model in the overrun example: (a) signal model, (b) timing model for aborted computations, and (c) aperiodic timing model for extended periods.

control algorithm and the actuator. The controller is designed assuming a constant input-output latency equal to the average execution-time of the task.

In the timing model, the execution time of the control algorithm is represented by the delay τ_1. The delay varies randomly between 0 and C^{max} according to a uniform probability distribution. In the timing model for aborted computations (Figure 6.13(b)), the first node is periodic with the interval h. If $\tau_1 > h$, the execution is aborted and system H_2 is never executed. By contrast, the timing model for extended periods (Figure 6.13(c)) is aperiodic. The system stays in the first node for the whole duration of τ_1. The delay in the second node depends on the first delay and is given by $\tau_2 = \max(0,\, h - \tau_1)$. Hence, the actual sampling period is never shorter than the nominal sampling interval h.

Assuming values of C^{max} from 0 to $2h$, the cost function (6.44) has been evaluated for the different overrun handling methods. Due to the use of the iterative solver, the calculation time for the aperiodic model was in the order of 50 times longer than for the periodic timing model. The results are shown in Figure 6.14. Up to $C^{max} = h$, no overruns are possible and the methods perform identically well. Above this point, however, the method of extended periods clearly outperforms the method of aborted computations. The results indicate that enforcing hard deadlines (by aborting the task at the deadline) may not be a good idea in real-time control systems.

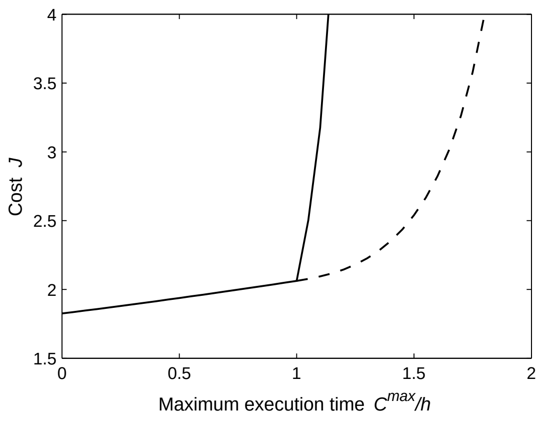


Figure 6.14 The costs in the overrun example: aborted computations (full) and extended periods (dashed).

6.5 Conclusion

This chapter has presented the MATLAB toolbox Jitterbug for analysis of real-time control performance. A stochastic timing model with random delays is used to describe the control loop timing. The timing description is somewhat limited in that it cannot handle dependencies between periods. Also, the stationary cost function only gives an average-case measure of the control performance. As a consequence, it is for instance not possible to use Jitterbug to evaluate the performance of a feedback scheduling system (see Chapter 4), where the CPU load changes and where the sampling periods of the controllers are changing over time. Another limitation of the toolbox is that only linear systems can be analyzed. These limitations can be avoided by instead using simulation (see the next chapter).

7

Simulation Using TrueTime

7.1 Introduction

The use of Jitterbug assumes knowledge of sampling period and latency distributions. This information can be difficult to obtain without access to measurements from the true target system under implementation. Also, the analysis cannot capture all the details and nonlinearities (especially in the real-time scheduling) of the computer system. A natural approach is to use simulation instead. However, today's simulation tools make it difficult to simulate the true temporal behavior of control loops. What is normally done is to introduce time delays in the control loop representing average-case or worst-case delays. Taking a different approach, the MATLAB/Simulink-based tool TrueTime facilitates simulation of the temporal behavior of a multitasking real-time kernel executing controller tasks. The tasks are controlling processes that are modeled as ordinary Simulink blocks. TrueTime also makes it possible to simulate simple models of communication networks and their influence on networked control loops. Different scheduling policies may be used (e.g., fixed-priority or earliest-deadline-first scheduling).

Furthermore, TrueTime can be used as an experimental platform for research on dynamic real-time control systems. For instance, it is possible to study compensation schemes that adjust the control algorithm based on measurements of actual timing variations. It is also easy to experiment with more flexible approaches to real-time scheduling of controllers, such as feedback scheduling (see Chapter 4).

In TrueTime, computer and network blocks are introduced. The computer blocks are event-driven and execute user-defined tasks and interrupt handlers representing, e.g., I/O tasks, control algorithms, and network interfaces. The scheduling policy of the individual computer blocks

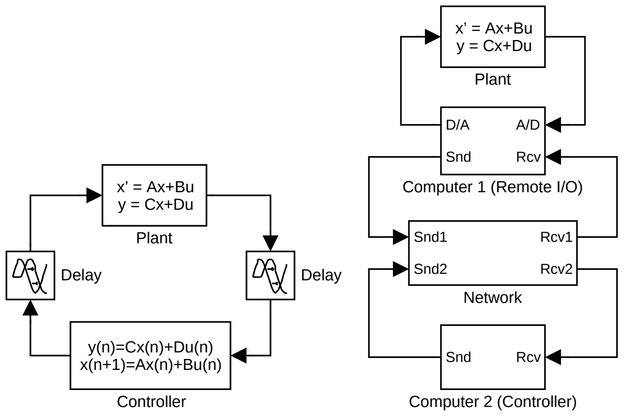


Figure 7.1 Left: Traditional simulation model of a distributed control system. Computers and network are modeled as simple delays. Right: TrueTime model where the execution of tasks and the transmission of messages are simulated in parallel with the plant dynamics.

is arbitrary and decided by the user. Likewise, in the network, messages are sent and received according to a chosen network model. A comparison between a TrueTime simulation model and a traditional simulation model of a distributed control system is shown in Figure 7.1.

The level of simulation detail is also chosen by the user—it is often neither necessary nor desirable to simulate code execution on instruction level or network transmissions on bit level. TrueTime allows the execution time of tasks and the transmission times of messages to be modeled as constant, random, or data-dependent. TrueTime can also handle simulation of context switches and task synchronization using events and monitors.

Related Work

While numerous other tools exist that support either simulation of control systems (e.g., MATLAB/Simulink [MathWorks, 2001], Dymola [Brück *et al.*, 2002]) or scheduling algorithms (e.g., STRESS [Audsley *et al.*, 1994], DRTSS [Storch and Liu, 1996], RTSIM [Casile *et al.*, 1998]), very few tools have been developed that support cosimulation of control systems and real-time scheduling. The RTSIM simulator [Casile *et al.*, 1998] has recently been extended with a numerical module (based on the Octave library) that supports simulation of continuous dynamics [Palopoli *et al.*, 2000]. Compared to TrueTime, there is no graphical control systems editor in this tool. At the other end of the usability spectrum, the simulation tool presented in [El-khoury and Törngren, 2001] is entirely based on graphical modeling in Simulink, but is limited to predefined network and CPU scheduling policies (currently, CAN networks and fixed-priority scheduling). Ptolemy [Bhattacharyya *et al.*, 2002] is a tool for modeling and simulation of heterogeneous (multi-domain) systems, such as mixed continuous-time/discrete-event systems. The recently developed *timed multitasking* (TM) domain [Liu and Lee, 2003] adds the possibility to model fixed-priority scheduling of tasks with fixed execution times.

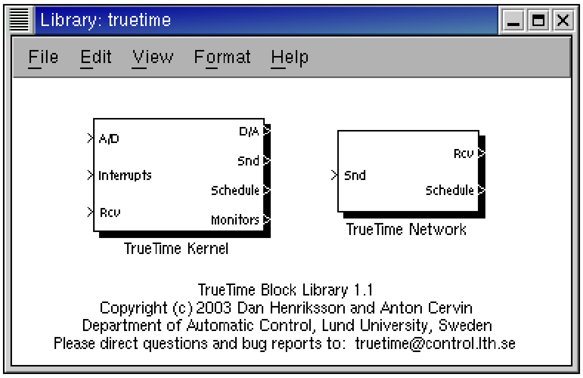
An early, tick-based prototype of the TrueTime simulator was presented in [Eker and Cervin, 1999]. The current version is event-based and written in C++, reducing simulation times by an order of magnitude or more. Furthermore, the new version supports simulation of external interrupts, context switches, and several network medium access protocols. It also adds the possibility to model control tasks using ordinary Simulink block diagrams.

7.2 Overview

The TrueTime block library is shown in Figure 7.2. Two blocks are available: the TrueTime Kernel block and the TrueTime Network block. Both blocks are event-driven, with the execution determined both by internal and external events. Internal events are time-related and correspond to events such as "a timer has expired," "a task has finished its execution," or "a message has completed its transmission." External events correspond to external interrupts, such as "a message arrived on the network" or "the engine crank angle passed zero degrees."

The block inputs are assumed to be discrete-time signals, except the signals connected to the A/D converters of the computer block, which may be continuous-time signals. All outputs are discrete-time signals. The Schedule and Monitors outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

The blocks are implemented as variable-step, discrete-time Simulink S-functions and are written in C++. The Simulink engine is only used for timing and for interfacing with the rest of the model (i.e., the continuous dynamics). It should thus be easy to port the blocks to other simulation environments, provided that these environments support event detection (zero-crossing detection).



TrueTime Block Library 1.1
Copyright (c) 2003 Dan Henriksson and Anton Cervin
Department of Automatic Control, Lund University, Sweden
Please direct questions and bug reports to: truetime@control.lth.se

Figure 7.2 The TrueTime block library. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

7.3 The Kernel Block

The TrueTime Kernel block simulates a computer with a simple but flexible real-time kernel, A/D and D/A converters, a network interface, and external interrupt channels.

Internally, the kernel maintains several data structures that are commonly found in a real-time kernel: a ready queue, a time queue, and records for tasks, interrupt handlers, monitors and timers that have been created for the simulation.

The execution of tasks and interrupt handlers is defined by user-written code functions. The code functions can be written either in C++ (for speed) or as MATLAB m-files (for ease of use). Control algorithms may also be defined graphically using ordinary discrete Simulink block diagrams.

Tasks

The task is the main construct in the TrueTime simulation environment. Tasks are used to simulate both periodic activities, such as controller and I/O tasks, and aperiodic activities, such as communication tasks and event-driven controllers.

TrueTime tasks may be periodic or aperiodic. Aperiodic tasks are executed by the creation of task instances (jobs). All pending jobs are inserted in a job queue of the task sorted by their release times. For periodic tasks, an internal timer is set up to periodically create the task jobs.

Apart from its code function, each task is characterized by a number of attributes. The static attributes of a task include

- a relative deadline,
- a priority,
- a worst-case execution time, and
- a period (if the task is periodic).

These attributes are kept constant throughout the simulation, unless explicitly changed by the user. Note that the worst-case execution time does not determine the *actual* execution time of the task (see the Code section below).

In addition to these attributes, each task instance has a number of dynamic attributes. The attributes are updated by the kernel as the simulation progresses. They include

- an absolute deadline,
- a release time, and
- an execution-time budget (by default equal to the worst-case execution time at the release of the job).

These attributes may also be changed by the user during simulation. Depending on the scheduling policy, the change of an attribute may lead to a context switch. For instance, under EDF scheduling, changing the absolute deadline of a task will result in a re-sorting of the ready queue.

Interrupts and Interrupt Handlers

Interrupts may be generated in two ways: externally or internally. An external interrupt is associated with one of the external interrupt channels of the computer block. The interrupt is triggered when the signal of the corresponding channel changes value. This type of interrupt may be used to simulate combustion engine controllers that are sampled against the crank rotation or distributed controllers that execute when measurements arrive from the network.

Internal interrupts are associated with timers. Both periodic timers and one-shot timers can be created. The corresponding interrupt is triggered when the timer expires. Timers are also used internally by the kernel to implement overrun handlers (see below).

When an external or internal interrupt occurs, a user-defined interrupt handler is scheduled to serve the interrupt. An interrupt handler works much the same way as a task, but is scheduled on a higher priority level. Interrupt handlers will normally perform small, less time-consuming tasks, such as generating an event or triggering the execution of a task. An interrupt handler is defined by a name, a priority, and a code function. External interrupts also have a latency during which they are insensitive to new invocations.

Priorities and Scheduling

Simulated execution occurs at three distinct priority levels: the interrupt level (highest priority), the kernel level, and the task level (lowest priority). The execution may be preemptive or nonpreemptive; this can be specified individually for each task and interrupt handler.

At the interrupt level, interrupt handlers are scheduled according to fixed priorities. At the kernel level, context switches are simulated. At the task level, dynamic-priority scheduling may be used. At each scheduling point, the priority of a task is given by a user-defined priority function, which is a function of the task attributes. This makes it easy to simulate different scheduling policies. For instance, a priority function that returns a priority number implies fixed-priority scheduling, whereas a priority function that returns a deadline implies deadline-driven scheduling. Predefined priority functions exist for rate-monotonic, deadline-monotonic, fixed-priority, and earliest-deadline-first scheduling.

Code

The code associated with tasks and interrupt handlers is scheduled and executed by the kernel as the simulation progresses. The code can be divided into several segments, as shown in Figure 7.3. The code can interact with other tasks and with the environment at the beginning of each code segment. This execution model makes it possible to model input-output latencies, blocking when accessing shared resources, etc. The simulated execution time of each segment is returned by the code function, and can thus be modeled as constant, random, or even data-dependent. The kernel keeps track of the current segment and calls the code functions with the proper argument during the simulation. Execution resumes in the next segment when the task has been running for the time associated with the previous segment. This means that preemption from higher-priority tasks and interrupts may cause the actual delay between the segments to be longer than the execution time.

Listing 7.1 shows an example of a code function corresponding to the time line in Figure 7.3. The function implements a simple controller. In the first segment, the plant is sampled and the control signal is computed. The execution time of the segment is set to 2 ms. In the second segment, the control signal is actuated and the controller states are updated. The execution time of this segment is set to 3 ms. The third segment indicates the end of the code function.

The functions `calculateOutput` and `updateState` in the code represent the implementation of an arbitrary control algorithm. The data structure `data` represents the local memory of the task and is used to store the controller state between calls to the different segments. A/D and

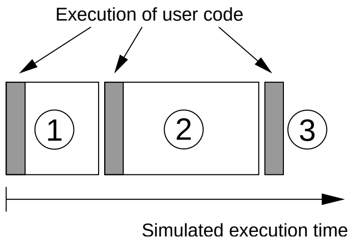


Figure 7.3 The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

D/A conversion is performed using the kernel primitives `ttAnalogIn` and `ttAnalogOut`.

Note that the input-output latency of this controller is *at least* 2 ms (i.e., the execution time of the first segment). If there is preemption from higher-priority tasks or interrupt handlers, the actual input-output latency will be longer.

Graphical Controller Representation

As an alternative to textual implementation of the controller algorithms, TrueTime also allows for graphical representation of the controllers. Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions, using the primitive `ttCallBlockSystem`.

Listing 7.1 Example of a simple controller code function written in MATLAB code. The internal state of controller is represented by the data structure `data`.

```
function [exectime,data] = myController(segment,data)
switch segment,
  case 1,
    data.y = ttAnalogIn(1);
    data = calculateOutput(data);
    exectime = 0.002;
  case 2,
    ttAnalogOut(1,data.u);
    data = updateState(data);
    exectime = 0.003;
  case 3,
    exectime = -1; % finished
end
```

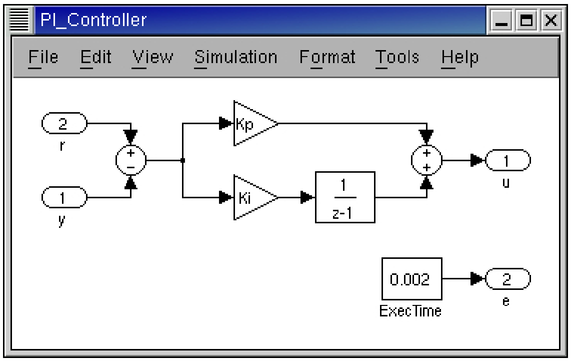


Figure 7.4 Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions. The example above shows a PI controller.

A block diagram of a PI controller is shown in Figure 7.4. The block system has two inputs, the reference signal and the process output, and two outputs, the control signal and the execution time.

Synchronization

Synchronization between tasks is supported by monitors and events. Monitors are used to guarantee mutual exclusion when accessing common data. Events can be associated with monitors to represent condition variables. Events may also be free (i.e., not associated with a monitor). This feature can be used to obtain synchronization between tasks where no conditions on shared data are involved. The example in Listing 7.2 shows the use of a free event `input_event` to simulate an event-driven controller task. The corresponding `ttNotifyAll`-call of the event is typically performed in an interrupt handler associated with an external interrupt port. (An alternative implementation of an event-based task, using `ttCreateJob`, will be given in Listing 7.4.)

Output Graphs

Depending on the simulation, several different output graphs are generated by the TrueTime blocks. Each computer block will produce two graphs, a computer schedule and a monitor graph, and the network block will produce a network schedule. The computer schedule will display the execution trace of each task and interrupt handler during the course of the simulation. If context switching is simulated, the graph will also display the execution of the kernel. An example of such an execution trace is shown in Figure 7.5. If the signal is high it means that the task is running. A medium signal indicates that the task is ready but not running (pre-

Listing 7.2 Example of a code function implementing an event-based controller. The task is aperiodic—hence the call to `ttSetNextSegment` in the final segment.

```
function [exectime,data] = eventController(segment,data)
switch segment,
  case 1,
    ttWait('input_event');
    exectime = 0.0;
  case 2,
    data.y = ttAnalogIn(1);
    data = calculateOutput(data);
    exectime = 0.002;
  case 3,
    ttAnalogOut(1,data.u);
    data = updateState(data);
    exectime = 0.003;
  case 4,
    ttSetNextSegment(1); % loop
end
```

empted), whereas a low signal means that the task is idle. In an analogous way, the network schedule shows the transmission of messages over the network, with the states representing sending (high), waiting (medium), and idle (low). The monitor graph shows what tasks are holding and waiting on the different monitors during the simulation. Generation of these execution traces is optional and can be specified individually for each task, interrupt handler, and monitor.

Advanced Features

In some modern real-time programming languages, e.g., Real-Time Java (RTSJ) [Bollella *et al.*, 2000], it is possible to monitor the execution time and the deadline of a task. In TrueTime, two interrupt handlers can be associated with each task: a deadline overrun handler (triggered if the task misses its deadline) and an execution-time overrun handler (triggered if the task executes longer than its worst-case execution time). The handlers can be used to control the behavior of tasks in overload situations. For instance, an overrun handler can contain code for terminating a job gracefully, setting the state to a suitable value before calling `ttKillJob`.

To facilitate arbitrary dynamic scheduling policies, it is possible to attach small pieces of code (called *scheduling hooks*) to each task. The hooks are executed at different stages during the simulation of the task, as shown in Figure 7.6. By default, the hooks contain code for handling

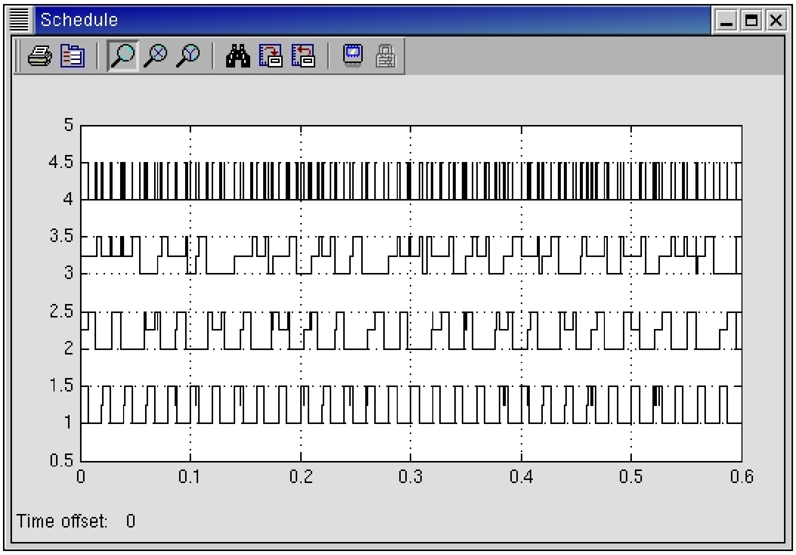


Figure 7.5 Example of an execution trace generated by a computer block during a simulation. The example involves three periodic tasks. The upper graph shows the execution of the kernel simulating context switches.

the execution-time and deadline overrun triggers, as summarized below.

- *Release hook:* If the task is associated with a deadline overrun handler, a timer is created. The expiry time of the timer is set to the absolute deadline of the task.
- *Start hook:* If the task is associated with an execution-time overrun handler, a timer is created. The expiry time of the timer is set to the current time plus the remaining execution-time budget. The start time of the task is recorded.

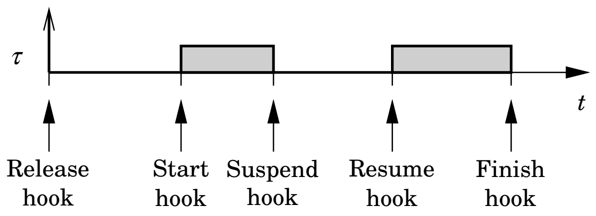


Figure 7.6 The various scheduling hooks that can be used to attach arbitrary functionality to the scheduling algorithm.

- *Suspend hook:* The execution-time budget is decreased based on the time since the last start of the task. The execution-time overrun timer is removed.
- *Resume hook:* The execution-time overrun timer is created again. The new start time is recorded.
- *Finish hook:* The execution-time budget is updated. Both overrun timers are removed.

Internal Workings

The TrueTime real-time kernel is implemented in a function `runKernel` that is called by the Simulink S-function callback functions at appropriate times during the simulation. The workings of the kernel are described by the pseudo code in Listing 7.3. Note that interrupt handlers are not treated in the pseudo code; they are handled in essentially the same way as the tasks.

Command Summary

To give an overview of the functionality of TrueTime, a summary of the available functions and commands is given in Table 7.1. The table is divided into three sections. The first section contains commands that are typically used in the initialization script of a simulation. The second section contains commands for setting and getting task (or job) attributes. Finally, the third section contains real-time primitives that may be used in the task code.

7.4 The Network Block

The network model is similar to the real-time kernel model, albeit simpler. The network block is event-driven and executes when messages enter or leave the network. A message contains information about the sending and the receiving computer node, arbitrary user data (typically measurement signals or control signals), the length of the message, and optional real-time attributes such as a priority or a deadline.

In the network block, it is possible to specify the transmission rate, the medium access control protocol (CSMA/CD, CSMA/CA, round robin, FDMA, or TDMA), and a number of other parameters. A long message can be split into frames that are transmitted in sequence, each with an additional overhead. When the simulated transmission of a message has completed, it is put in a buffer at the receiving computer node, which is notified by a hardware interrupt. More details on the network block can be found in [Henriksson and Cervin, 2003].

Listing 7.3 Pseudo-code for the TrueTime kernel function.

```
double runKernel() {

    // Compute time elapsed since last invocation
    timeElapsed = currentTime - prevHit;
    prevHit = currentTime;
    nextHit = 0.0;

    while (nextHit == 0.0) {

        // Count down execution time for current task instance
        // and check if it has finished its execution
        if (there exists a running task) {
            Decrease remaining exec. time with timeElapsed;

            if (remaining execution time <= 0.0) {
                Execute next segment of the code function;
                Update remaining execution time;
                Update execution time budget;

                if (remaining execution time < 0.0) {
                    // Negative execution time = Job finished
                    Remove the task from the ready queue;
                    Execute finish-hook;
                    Simulate saving context;
                    if (there are pending jobs) {
                        Move the next job to the time queue;
                    }
                }
            }
        }

        // Go through the time queue (ordered after release)
        for (each task) {
            if (release time - currentTime <= 0.0) {
                Remove the task from the time queue;
                Move the task to the ready queue;
                Execute release-hook;
            }
        }
```

Listing 7.3 (Continued)

```
        // Go through the timer queue (ordered after expiry time)
        for (each timer) {
            if (expiry time - currentTime <= 0.0) {
                Activate handler associated with timer;
                Remove timer from timer queue;
                if (timer is periodic) {
                    Increase the expiry time with the period;
                    Insert the timer in the timer queue;
                }
            }
        }

        // Dispatching
        Make the first task in the ready queue the running task;

        if (the task is being started) {
            Execute the start-hook for the task;
            Simulate restoring context;
        } else if (the task is being resumed) {
            Execute the resume-hook for the task;
            Simulate restoring context;
        }
        if (another task is suspended) {
            Execute suspend-hook of the previous task;
            Simulate saving context;
        }

        // Determine next invocation of the kernel function
        time1 = remaining execution time of the current task;
        time2 = next release of a task from the time queue;
        time3 = next expiry time of a timer;
        nextHit = min(time1, time2, time3);

    } // loop while nextHit = 0.0
    return nextHit;
}
```

Table 7.1 Summary of the TrueTime commands.

Command	Description
`ttInitKernel`	Initialize the kernel.
`ttInitNetwork`	Initialize the network interface.
`ttCreatePeriodicTask`	Create a task with periodic jobs.
`ttCreateTask`	Create a task (but no jobs).
`ttCreateInterruptHandler`	Create an interrupt handler.
`ttCreateExternalTrigger`	Associate an interrupt handler with an external interrupt channel.
`ttCreateMonitor`	Create a monitor.
`ttCreateEvent`	Create an event variable, possibly associated with a monitor.
`ttCreateMailbox`	Create a mailbox for inter-task communication.
`ttNoSchedule`	Switch off the schedule output graph for a specific task or interrupt handler.
`ttNonPreemptable`	Make a task non-preemptable.
`ttAttachDLHandler`	Attach a deadline overrun handler to a task.
`ttAttachWCETHandler`	Attach a worst-case execution time overrun handler to a task.
`ttAttachPrioFcn`[1]	Attach an arbitrary task priority function to be used by the scheduler.
`ttAttachHook`[1]	Attach a scheduling hook to a task.
`ttSetDeadline`	Set the relative deadline of a task.
`ttSetAbsDeadline`	Set the absolute deadline of a job.
`ttSetPriority`	Set the priority of a task.
`ttSetPeriod`	Set the period of a periodic task.
`ttSetBudget`	Set the execution time budget of a job.
`ttSetWCET`	Set the worst-case execution time of a task.
`ttGetRelease`	Get the release time of a job.
`ttGetDeadline`	Get the relative deadline of a task.
`ttGetAbsDeadline`	Get the absolute deadline of a job.
`ttGetPriority`	Get the priority of a task.
`ttGetPeriod`	Get the period of a periodic task.
`ttGetBudget`	Get the execution time budget of a job.
`ttGetWCET`	Get the worst-case execution time of a task.

[1]Available in the C++ version only.

Table 7.1 (Continued)

Command	Description
`ttCreateJob`	Create a job with a given release time.
`ttKillJob`	Kill the running job of a task.
`ttEnterMonitor`	Attempt to enter a monitor.
`ttExitMonitor`	Exit a monitor.
`ttWait`	Wait for an event.
`ttNotifyAll`	Notify all tasks waiting for an event.
`ttTryFetch`	Fetch a message from a mailbox.
`ttTryPost`	Post a message to a mailbox.
`ttCreateTimer`	Create a one-shot timer and associate an interrupt handler with the timer.
`ttCreatePeriodicTimer`	Create a periodic timer and associate an interrupt handler with the timer.
`ttRemoveTimer`	Remove a specific timer.
`ttCurrentTime`	Get the current time in the simulation.
`ttSleepUntil`	Put a task to sleep until a certain point in time.
`ttSleep`	Put a task to sleep for a certain duration.
`ttAnalogIn`	Read the value of an analog input.
`ttAnalogOut`	Write a value to an analog output.
`ttSetNextSegment`	Set the next segment to be executed in the code function.
`ttInvokingTask`	Get the name of the task that invoked an interrupt handler.
`ttCallBlockSystem`	Call a Simulink block diagram from within a code function.
`ttSendMsg`	Send a message over the network.
`ttGetMsg`	Get a message that has been received over the network.

7.5 Examples

In this section, a number of examples that illustrate the use of TrueTime are given.

Example 1: Distributed Control System

As a first example of simulation in TrueTime, we again turn our attention to the networked control system (see Example 1 in Section 6.4). Using TrueTime, detailed simulation of the distributed control system is

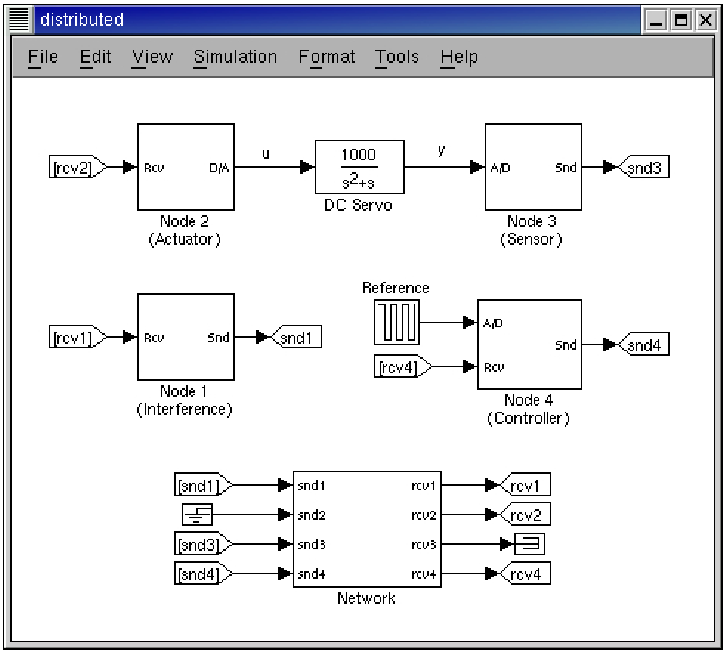


Figure 7.7 TrueTime model of the distributed control system. An interference node that generates high-priority network traffic has been introduced. The four computer nodes are communicating over a network.

possible, where the effects of scheduling in the CPUs and simultaneous transmission of messages over the network can be studied.

The TrueTime model of the system is shown in Figure 7.7. An interference node that generates high-priority network traffic is introduced. Each of the four nodes in the model contains a TrueTime Kernel block. The time-driven sensor node contains a periodic task, which at each invocation samples the process and sends the sample to the controller node over the network. The controller node contains an event-driven task that is triggered each time a sample arrives over the network from the sensor node. Upon receiving a sample, the controller computes a control signal, which is then sent to the event-driven actuator node, where it is actuated. Finally, the interference node contains a periodic task that generates random interfering traffic over the network.

Initialization of the Actuator Node. Listing 7.4 shows the complete code needed to initialize the actuator node in this particular example. The computer block contains one task and one interrupt han-

Listing 7.4 Complete code for the actuator node in the distributed control system example. Instances of the aperiodic actuator task are created using `ttCreateJob`.

```
%% Code function for the actuator task
function [exectime,data] = actcode(segment,data)
switch segment,
  case 1,
    data.u = ttGetMsg;
    exectime = 0.0005;
  case 2,
    ttAnalogOut(1, data.u);
    exectime = -1;
end

%% Code function for the network interrupt handler
function [exectime,data] = msgRcvHandler(segment,data)
ttCreateJob('act_task',ttCurrentTime);
exectime = -1;

%% Initialization function
function actuator_init
nbrOfInputs = 0;
nbrOfOutputs = 1;
ttInitKernel(nbrOfInputs,nbrOfOutputs,'prioFP',0);
priority = 5;
deadline = 0.010;
ttCreateTask('act_task',deadline,priority,'actcode');
ttCreateInterruptHandler('msgRcv',1,'msgRcvHandler');
ttInitNetwork(2,'msgRcv'); % I am node 2
```

dler, and their execution is defined by the code functions `actcode` and `msgRcvHandler`, respectively. The task and the interrupt handler are created in the `actuator_init` initialization function. The node is "connected" to the network using the function `ttInitNetwork` by supplying a node identification number and the name of the interrupt handler (`'msgRcv'`) to be executed when a message arrives to the node. In the `ttInitKernel` function, the kernel is initialized by specifying the number of A/D and D/A channels, the scheduling policy, and the time for a full context switch (zero in this case). The built-in priority function `prioFP` specifies fixed-priority scheduling.

Experiments. In the following simulations, we will assume a CAN-type network where transmission of simultaneous messages is decided based on priorities of the packages. The PD controller executing in the controller node is designed with a 10 ms sampling interval. The same sampling interval is used in the sensor node.

The execution time of the controller is assumed to be 0.5 ms and the ideal transmission time from one node to another is 1.5 ms. The minimum input-output latency is thus 3.5 ms. The packages generated by the interference node have high priority and occupy 50% of the network bandwidth. We further assume that an interfering, high-priority task with a 7 ms period and a 3 ms execution time is executing in the controller node. Colliding transmissions and preemption in the controller node will thus cause the input-output latency to be longer and time-varying. The resulting degraded control performance can be seen in the simulated step response in Figure 7.8. Also shown in the figure are the network schedule and the controller node schedule.

Finally, a simple compensation is introduced to cope with the delays. The packages sent from the sensor node are now time-stamped, which makes it possible for the controller to determine the actual delay from sensor to controller. The total delay is estimated by adding the expected value of the delay from controller to actuator. The control signal is then calculated based on linear interpolation among a set of controller parameters precalculated for different delays. Using this compensation, better control performance is obtained, as seen in Figure 7.9.

Example 2: Subtask Scheduling

As noted in the subtask scheduling example in Section 3.6, having different priorities in the Calculate Output and Update State parts of the control algorithm can increase the number of context switches. If the penalty for context switches is large, this can increase the latencies and potentially degrade the control performance. This can be investigated in TrueTime, where effects of context switches can be included in the simulation.

A code function implementing subtask scheduling under fixed-priority scheduling is shown in Listing 7.5. The priority is changed between the subtasks using calls to `ttSetPriority`.

The time for a full context switch is set to 0.5 ms (specified in the `ttInitKernel` call). This should be compared with the controller execution time, which is 7 ms. A simulation of the pendulum system, including the kernel overhead, is shown in Figure 7.10. The performance is still better than under naive scheduling (see Figure 1.3), despite the very large number of context switches introduced by the subtask scheduling policy.

(a)

(b)

(c)

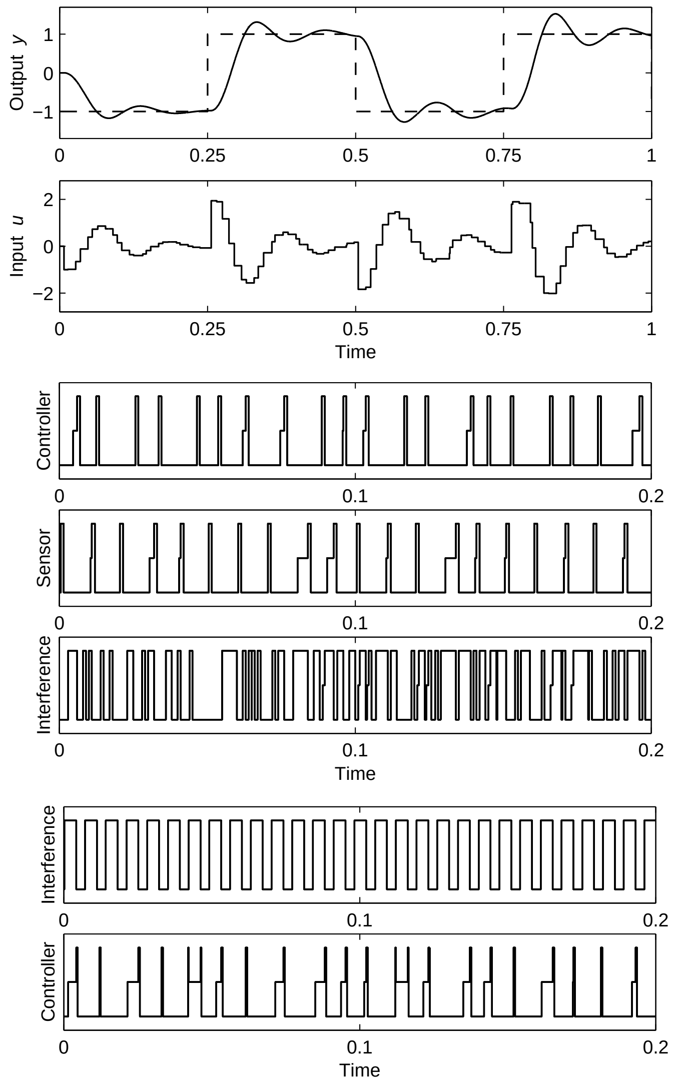


Figure 7.8 Simulation of the distributed control system: (a) control performance, (b) close-up of the network schedule, and (c) close-up of the computer node schedule. The scheduling-induced latency in the control loop degrades the control performance.

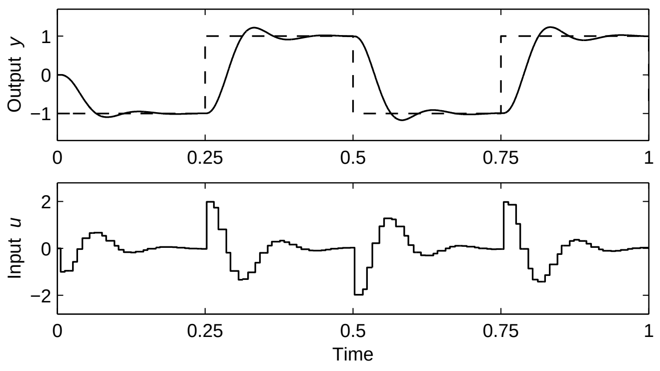


Figure 7.9 Simulation of the distributed control system with scheduling-induced delays and delay-compensation. The performance is improved compared to Figure 7.8.

Example 3: Execution-Time Estimation

The feedback scheduler introduced in Chapter 4 requires that the execution times of the various tasks can be measured. Functionality to measure execution times is provided in TrueTime through the use of budgets. When a task instance is created, the budget is set to the declared WCET of the task. As the task executes, the budget is decreased by the same amount. When the task finishes, the actual execution time of the task instance can be found by comparing the remaining budget to the WCET. Execution-time estimation using a forgetting factor (see Eq. (4.1)) can be performed in a custom *finish hook* (see Section 7.3) that can be reused for all tasks. Assuming that the task user data structure (`task->data`) has two fields called `Chat` (the estimate) and `lambda` (the forgetting factor), a finish hook implementing the execution-time estimation is shown in Listing 7.6.

Example 4: Server-Based Scheduling

Server-based scheduling (see Section 2.2) can be used to limit the utilization of aperiodic tasks, or tasks that have unknown or unbounded execution time. In Chapter 5, modified constant bandwidth servers (CBSs) were used to provide isolation between unrelated tasks in the Control Server model. In TrueTime, general server mechanisms can be implemented through the use of scheduling hooks and timers. This requires knowledge about the internal workings of the kernel and also that some custom C++ code is written.

More simple server mechanisms can be implemented using task budgets and execution-time overrun handlers. In this example, we will limit

Listing 7.5 Implementation of subtask scheduling in TrueTime. The priority is changed between the subtasks using calls to `ttSetPriority`.

```
function [exectime,data] = regulCode(segment,data)
switch segment,
  case 1,
    data.y = ttAnalogIn(data.inChan);
    data.u = data.C * data.x + data.D * data.y;
    exectime = 0.003;
  case 2,
    ttAnalogOut(data.outChan, data.u);
    ttSetPriority(data.prioUS);
    exectime = 0;
  case 3,
    data.x = data.Phi * data.x + data.Gamma * data.y;
    exectime = 0.004;
  case 4,
    ttSetPriority(data.prioCO);
    exectime = -1;
end
```

the utilization of a task with unbounded execution time using a simplified version of the CBS (that only handles a single task instance). Again, assume that a DC servo should be controlled by a PD controller with the sampling interval $h = 10$ ms (see Example 1 in this section). The execution time of the control algorithm is 2 ms. Further, assume that a disturbance task with unbounded execution time starts to execute at

Listing 7.6 Execution-time estimation using a custom finish hook written in C++.

```
void myFinishHook(Task *task) {

  // Get the execution time of the current job
  double exectime = task->WCET - task->currentJob->budget;
  // Update the execution-time estimate
  task->data->Chat = task->data->lambda * task->data->Chat +
    (1.0 - task->data->lambda) * exectime;

  default_finish(task); // Execute the default finish hook
}
```

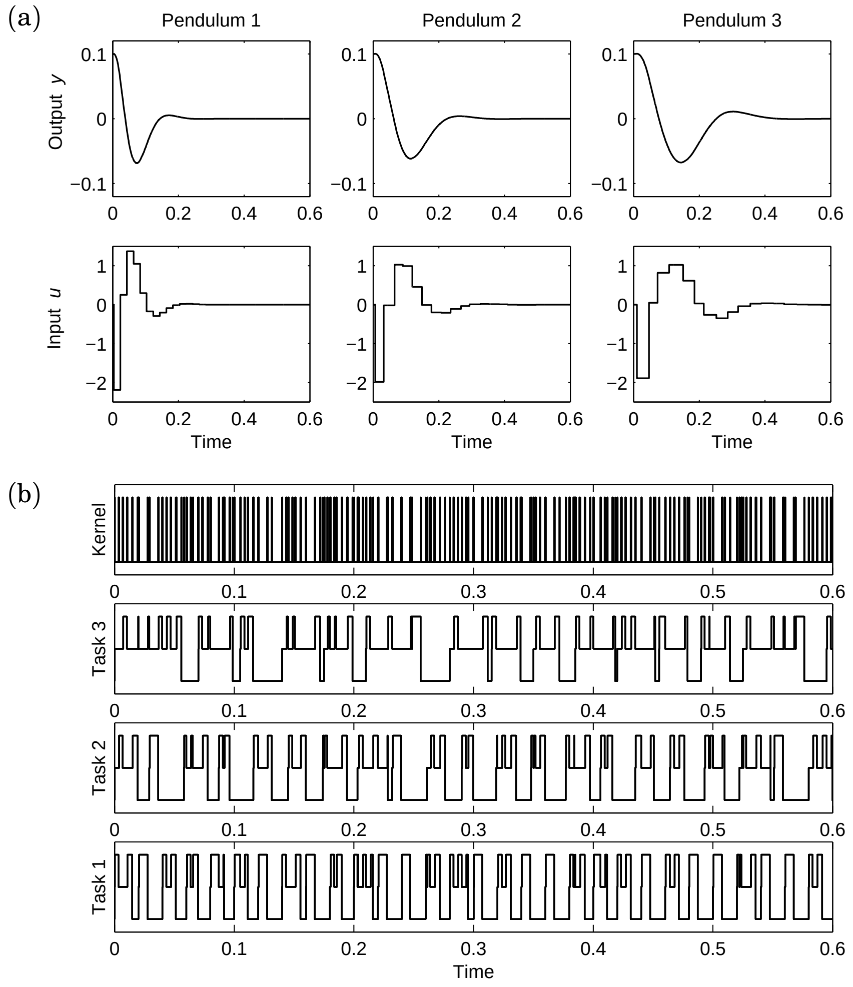
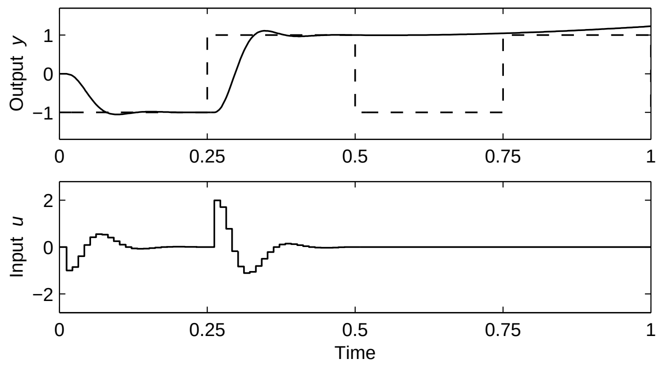


Figure 7.10 TrueTime simulation of the pendulum system under subtask scheduling, including context switches: (a) control performance, and (b) task schedule. The performance is good, despite the very large number of context switches.

time $t = 0.5$. A simulation of the system under ordinary EDF scheduling is shown in Figure 7.11. The disturbance task blocks the controller completely.

Next, a server is introduced to limit the utilization of the disturbance task. The server utilization is set to $U_s = 0.5$ and the server period is (somewhat arbitrarily) chosen as $T_s = 0.04$. There should thus be enough time left for the controller (which requires a utilization of 0.2) to execute.

(a)



(b)

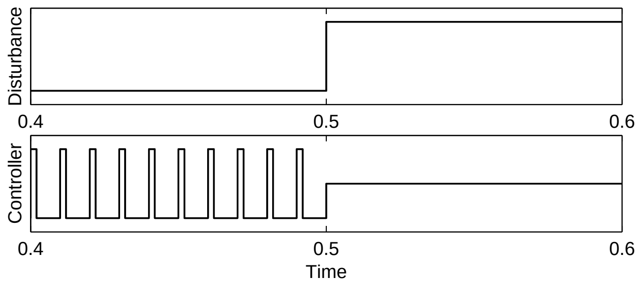


Figure 7.11 Simulation of the DC servo system under ordinary EDF scheduling: (a) control performance, and (b) close-up of schedule. The control task is completely blocked when the disturbance task starts to execute.

The complete code for the disturbance task, the server, and the initialization script is shown in Listing 7.7. In the initialization script, a WCET overrun handler (`ORhandler`) is attached to the disturbance task. Whenever the budget is exhausted, the overrun handler is invoked. There, the budget of the task is recharged to its maximum value ($T_s U_S$) and the task deadline is postponed one server period (T_s). (Compare with the CBS rules in Section 2.2.) A new simulation of the system, including the server, is shown in Figure 7.12. The overrun handler is invoked regularly, postponing the deadline of the disturbance task so that the controller task may execute at its desired rate.

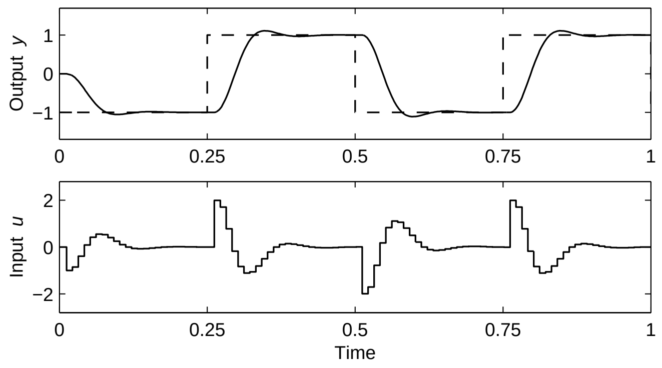
7.6 Conclusion

This chapter has presented TrueTime, a MATLAB/Simulink toolbox that facilitates event-based simulation of distributed real-time control systems. The simulations capture the true, timely behavior of controllers executing

Listing 7.7 Code for server-based scheduling in TrueTime.

```
%% Disturbance task
function [exectime,data] = distcode(segment,data)
switch seg,
  case 1,
    exectime = 1000; % "unbounded" execution time
  case 2,
    exectime = -1;   % finished
end

%% Budget overrun handler
function [exectime,data] = ORhandler(segment,data)
Ts = 0.04;                               % Server period
Us = 0.5;                                % Server bandwidth
task = ttInvokingTask;
ttSetBudget(Ts*Us,task);                 % Recharge budget
deadline = ttGetAbsDeadline(task)+Ts;    % Move deadline
ttSetAbsDeadline(deadline,task);
exectime = -1;

%% Initialization
function servosystem_init
nbrOfInputs = 2;
nbrOfOutputs = 1;
ttInitKernel(nbrOfInputs,nbrOfOutputs,'prioEDF');

% Controller task
period = 0.010;
ttCreatePeriodicTask('controller',0.0,period,1,'pidcode');

% Disturbance task
release = 0.5;
deadline = 0.0;
ttCreateTask('disturbance',deadline,1,'distcode');
ttCreateJob(release,'disturbance');

% Server
ttCreateInterruptHandler('ORhandler',1,'ORhandler');
ttAttachWCETHandler('disturbance','ORhandler');
ttSetBudget(0.0,'disturbance');
```

(a)



(b)

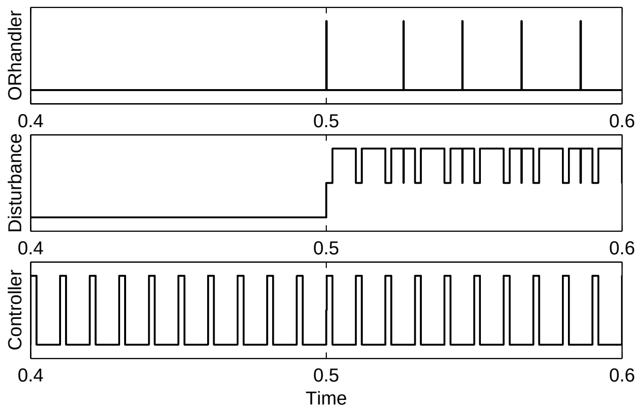


Figure 7.12 Server-based scheduling using an overrun handler: (a) control performance, and (b) close-up of schedule. The budget overrun handler repeatedly postpones the deadline of the disturbance task, such that the control task can execute periodically.

in a simulated real-time kernel. The simulator allows various scheduling strategies, control algorithms, and network protocols to be evaluated from a control performance perspective. The simulated real-time kernel executes user-defined tasks that can be implemented as MATLAB functions, C++ functions, or ordinary Simulink diagrams.

Another interesting possible use of TrueTime is run the simulation in real-time! This of course requires that a fast enough computer is used. Interfacing the computer to a real process using A-D and D-A converters, the setup can be used to emulate a slow, multitasking computer controlling a real plant.

8

Conclusion

8.1 Summary

The design of a real-time control system is essentially a codesign problem. Decisions made in the real-time design affect the control design, and vice versa. For instance, the choice of scheduling policy influences the latency distributions in the control loops, and, ideally, this should be taken into account in the control design. At the same time, the performance requirements of the individual control loops place demands on the real-time system with regard to, e.g., sampling periods and latencies. This thesis has dealt with various aspects of the control and scheduling codesign problem. The work has fallen into two main categories: the development of tools for analysis of control timing, and the development of scheduling techniques tailored to control tasks.

Tools for Analysis of Control Timing

A multitasking real-time control system is a very complex system. There are interactions between the continuous-time plant dynamics, the control tasks, and the real-time scheduling algorithm. In order to understand and analyze these systems, new software tools are needed. This thesis has presented two such tools: Jitterbug (Chapter 6) and TrueTime (Chapter 7).

Analysis Using Jitterbug. Controllers are often designed with little regard for the real-time implementation. Using the MATLAB-based toolbox Jitterbug, the control performance degradation due to slow sampling and scheduling-induced input-output latency and jitter can be computed analytically. The control system is built from a number of linear systems, and the performance is evaluated by a quadratic cost function. A stochastic timing model with random delays is used to describe the execution of the controller in each period. The tool can be used at early stages in the design process to determine how sensitive a controller is to timing variations. Being quite general, Jitterbug is applicable to a wide range of problems within control and signal processing.

Simulation Using TrueTime. The MATLAB/Simulink-based simulator TrueTime can be used to investigate real-time control systems in more detail. The control tasks are described by arbitrary code functions (written as MATLAB functions, C++ functions, or Simulink block diagrams), while the plant dynamics can be modeled using the full power of Simulink. In TrueTime, the internal workings of real-time kernels and communications networks are simulated in parallel with the plant. Unlike Jitterbug, the tool allows for time-domain responses and transient phenomena (in both scheduling and control) to be studied. Also, nonlinear systems may be studied. The various scheduling techniques presented in the thesis (see below) have all been developed and evaluated with the aid of TrueTime.

Scheduling Techniques Tailored to Control Tasks

Vast amounts of scheduling theory has been developed over the last decades, but very little of it is tailored to control tasks. It has been pointed out that the performance of a controller depends on the sampling interval, the input-output latency, and the sampling and input-output jitter. Using subtask scheduling (Chapter 3), the input-output latency can be reduced. Feedback scheduling was introduced in Chapter 4 to dynamically adjust the sampling periods, optimizing the overall performance. The Control Server (Chapter 5) combines ideas from the previous chapters to create control tasks with both predictable and adjustable performance.

Subtask Scheduling. Scheduling the two main parts of a control algorithm (Calculate Output and Update State) as separate tasks, it is possible to reduce the input-output latency (and jitter) in a set of control tasks. Deadline assignment—something which is seldom treated in the scheduling literature—has been considered under both fixed-priority (FP) and earliest-deadline-first (EDF) scheduling. Subtask scheduling under EDF is one of the ideas behind the Control Server.

Feedback Scheduling. A simple feedback scheduling strategy has been developed, where the sampling periods of a set of control tasks are adjusted based on execution-time measurements. The goal has been to keep a high CPU utilization, providing high control performance despite variations in the execution time of the control algorithms. Simulation case studies on hybrid and linear controllers have been presented. The simulations have mainly assumed FP scheduling, which is unfair to the lowest-priority tasks. Better performance is obtained using EDF scheduling. An even better alternative is to use Control Servers, which facilitate exact resource distribution among the tasks.

The Control Server. A new computational model for control tasks, called the Control Server model, has been proposed. The Control Server creates the abstraction of a control task with a specified sampling period and a constant input-output latency (shorter than the period), making it possible to account for the latency in the control design. Tasks may be combined into more complex components without loss of their individual fixed-latency properties. Based on the constant bandwidth server, the Control Server also provides isolation between unrelated tasks. Implementation of the Control Server requires that the real-time operating system supports earliest-deadline-first scheduling.

8.2 Suggestions for Future Work

The current work can be extended in many directions. Some suggestions for future work are given below.

Overrun Handling. A topic that has not received much attention is the problem of task overruns. What should be done if a control task does not finish before the deadline? Many alternatives exist: the task could be aborted, the computations could continue in the next period, an alternative action could be taken, etc. The problem becomes especially intricate when the control algorithm has been divided into segments. For instance, should the Update State part be executed even if the Calculate Output part was aborted (or did not finish in time)? Are some controller realizations more sensitive to aborted computations than others? The Control Server model could be extended to allow various overrun handling methods to be specified for different tasks.

Feedback Scheduling Structures. The feedback scheduling structure proposed in this thesis is only one of many possible. Other variables than the CPU utilization could be controlled, for instance the latency and jitter in each control loop. The resource distribution should ideally be based on the *current* performance of the controllers, and not just a stationary cost function. Move involved schemes could be developed, where the scheduler and the controllers negotiate for available resources at regular intervals.

Scheduling of Anytime Controllers. An interesting class of controllers that require new scheduling techniques are *anytime controllers*.

In such controllers, the quality of the control signal is gradually refined as the execution progresses. A good example is model-predictive control (MPC), where an optimization problem is solved by iterative techniques in each sample. To reach the optimum, very long execution times may be needed. Terminating the optimization early, acceptable results may still be obtained. There is an interesting trade-off between the input-output latency (due to the long execution time) and the quality of the control signal. A preliminary study on scheduling of MPCs is presented in [Henriksson *et al.*, 2002].

Control-Theoretic Issues. Jitter and changes of sampling intervals due to dynamic scheduling strategies lead to many interesting control-theoretical issues. Are there analytical results regarding the sensitivity of a controller towards jitter? Can a controller be designed to be robust against both sampling jitter and input-output jitter?

In feedback scheduling applications, the performance of a controller should ideally be measured using a non-stationary cost function that reflects the current status of the plant. How should these cost functions be formulated?

Scheduling of Networks. In distributed control systems, the network bandwidth may be the bottleneck resource, and not the CPU time in the nodes. Similar to the CPU scheduling case, the network resources could be dynamically distributed among several competing control loops. Network transmissions are typically nonpreemptive, and this makes the scheduling problem more difficult. Also, the feedback mechanism may need to be distributed among the nodes in the network. TrueTime is currently being extended to simulate higher-level network protocols such as TCP.

Implementation Issues. The feedback scheduler and the Control Server presented in the thesis assume that task execution times can be measured in the real-time operating system. Furthermore, the Control Server is based on EDF scheduling. Currently, these features are not available in many RTOSs. The Control Server was implemented in a public domain real-time kernel [Andersson and Blomdell, 1991]. An alternative would be to base the implementation on the open source RTLinux kernel, which can easily be modified to use EDF, e.g., [Vidal *et al.*, 2002]. Another promising implementation platform is Real-Time Java [Bollella *et al.*, 2000], which, according to the specification, supports both execution-time measurements and various scheduling policies. Currently, there is no known Real-Time Java implementation that supports EDF scheduling, however.

Appendix

Cost Calculation in Example 3.1

The delayed integrator process can be written

$$\frac{dx(t)}{dt} = u(t-L) + v_c(t), \qquad 0 \le L \le h, \tag{A.1}$$

where L is the input-output latency, and h is the sampling interval. The cost function to be minimized can be written

$$J = \frac{1}{h}\mathbf{E}\left\{\int_0^h x^2(t)\,dt\right\}. \tag{A.2}$$

Inserting the process description (A.1) into (A.2) gives

$$\begin{aligned} J = \frac{1}{h}\mathbf{E}\Bigg\{ & \int_0^L \left(x(kh) + tu(kh-h) + \int_0^t v_c(s)\,ds \right)^2 dt \\ & + \int_L^h \left(x(kh) + Lu(kh-h) + (t-L)u(kh) + \int_0^t v_c(s)\,ds \right)^2 dt \Bigg\} \\ = \frac{1}{h}\mathbf{E} & \left\{ \begin{pmatrix} x(kh) \\ u(kh-h) \\ u(kh) \end{pmatrix}^T \begin{pmatrix} Q_1 & Q_{12} \\ Q_{12}^T & Q_2 \end{pmatrix} \begin{pmatrix} x(kh) \\ u(kh-h) \\ u(kh) \end{pmatrix} \right\} + J_{samp}, \end{aligned} \tag{A.3}$$

where

$$Q_1 = \begin{pmatrix} h & \dfrac{L^2}{2} + (h-L)L \\ \dfrac{L^2}{2} + (h-L)L & \dfrac{L^3}{3} + (h-L)L^2 \end{pmatrix}, \quad Q_{12} = \begin{pmatrix} \dfrac{(h-L)^2}{2} \\ \dfrac{(h-L)^2}{2}L \end{pmatrix},$$

$$Q_2 = \frac{(h-L)^3}{3}, \quad J_{samp} = \frac{1}{h}\int_0^h \int_0^t 1\, ds\, dt = h/2.$$

Sampling the process (A.1) with the interval h gives

$$\begin{pmatrix} x(kh+h) \\ u(kh) \end{pmatrix} = \Phi \begin{pmatrix} x(kh) \\ u(kh-h) \end{pmatrix} + \Gamma u(kh) + v(kh), \tag{A.4}$$

where

$$\Phi = \begin{pmatrix} 1 & L \\ 0 & 0 \end{pmatrix}, \qquad \Gamma = \begin{pmatrix} h-L \\ 1 \end{pmatrix},$$

and v is a discrete-time white noise process with covariance

$$R = \begin{pmatrix} h & 0 \\ 0 & 0 \end{pmatrix}.$$

Introduce the positive definite matrix S. The controller that minimizes the cost satisfies the algebraic Riccati equation (e.g. [Åström and Wittenmark, 1997])

$$S = \Phi^T S \Phi + Q_1 - \Big(\Phi^T S \Gamma + Q_{12}\Big)\Big(\Gamma^T S \Gamma + Q_2\Big)^{-1}\Big(\Gamma^T S \Phi + Q_{12}^T\Big), \tag{A.5}$$

with the solution

$$S = \begin{pmatrix} L + \dfrac{\sqrt{3}\,h}{6} & \dfrac{L}{6}\Big(3L - \sqrt{3}\,h\Big) \\ \dfrac{L}{6}\Big(3L - \sqrt{3}\,h\Big) & \dfrac{L^3}{3} - \dfrac{\sqrt{3}\,L^2 h}{6} \end{pmatrix}.$$

The optimal control law is

$$u(kh) = -K \begin{pmatrix} x(kh) \\ u(kh-h) \end{pmatrix}, \tag{A.6}$$

where

$$K = \Big(Q_2 + \Gamma^T S \Gamma\Big)^{-1}\Big(\Gamma^T S \Phi + Q_{12}^T\Big) = \begin{pmatrix} \dfrac{1}{h}\dfrac{\sqrt{3}+3}{2+\sqrt{3}} & \dfrac{L}{h}\dfrac{\sqrt{3}+3}{2+\sqrt{3}} \end{pmatrix},$$

and the optimal cost is given by

$$J = \frac{1}{h}\operatorname{tr} SR + J_{samp} = \frac{3+\sqrt{3}}{6}h + L. \tag{A.7}$$

Proof of Theorem 4.2

The release time of job number k of task i is $kT_i + \phi_i$. The deadline of job number k of task i is $kT_i + \phi_i + D_i$. Let $k_i(t)$ be the number of finished jobs of task i at time t. After a while, CPU is never idle, therefore,

$$I_{idle}(t) + \sum_i (k_i(t)C_i + e_i(t)) = t, \tag{A.8}$$

where $I_{idle}(t)$ is the accumulated idle time, e_i is how long the current invocation of task i has executed. Both $I_{idle}(t)$ and e_i are bounded and $0 \leq e_i \leq C_i$. Furthermore, due to the overload situation, tasks are finished in the order of their deadlines. Therefore,

$$k_l(t)T_l + D_l + \phi_l \leq (k_i(t) + 1)T_i + D_i + \phi_i. \tag{A.9}$$

(Otherwise task l would not have finished job number $k_l(t)$ before task i finished job number $k_i(t) + 1$). Symmetrically,

$$k_i(t)T_i + D_i + \phi_i \leq (k_l(t) + 1)T_l + D_l + \phi_l. \tag{A.10}$$

The two equations above give

$$k_l(t)T_l + D_l + \phi_l - T_i \leq k_i(t)T_i + D_i + \phi_i \leq (k_l(t) + 1)T_l + D_l + \phi_l. \tag{A.11}$$

Hence,

$$\frac{k_l(t)T_l + D_l - T_i + \phi_l}{k_l(t)T_l} \leq \frac{k_i(t)T_i + D_i + \phi_i}{k_l(t)T_l} \leq \frac{(k_l(t) + 1)T_l + D_l + \phi_l}{k_l(t)T_l}. \tag{A.12}$$

Here, the limit of the left-hand side and the right-hand side are both equal to one, so,

$$\lim_{t \to \infty} \frac{k_j(t)T_j}{k_l(t)T_l} = 1. \tag{A.13}$$

Rearranging the terms in (A.8) and letting $t \to \infty$, we have

$$\begin{aligned}
1 &= \lim_{t\to\infty} \frac{1}{t} \sum_i k_i(t) C_i \\
&= \lim_{t\to\infty} \frac{1}{t} \sum_i k_i(t) T_i \frac{C_i}{T_i} \\
&= \lim_{t\to\infty} \frac{\sum_j k_j(t) T_j}{t} \sum_i \frac{k_i(t) T_i}{\sum_j k_j(t) T_j} \frac{C_i}{T_i} \\
&= \lim_{t\to\infty} \frac{k_i(t) T_i \sum_j \frac{k_j(t) T_j}{k_i(t) T_i}}{t} \sum_i \frac{1}{\sum_j \frac{k_j(t) T_j}{k_i(t) T_i}} \frac{C_i}{T_i} \\
&= \lim_{t\to\infty} \frac{k_i(t) T_i n}{t} \sum_i \frac{1}{n} \frac{C_i}{T_i} \\
&= \lim_{t\to\infty} \frac{k_i(t) T_i}{t} \sum_i \frac{C_i}{T_i}.
\end{aligned} \tag{A.14}$$

Hence,

$$\bar{T}_i = \lim_{t\to\infty} \frac{t}{k_i(t)} = T_i \sum_j \frac{C_j}{T_j} = T_i U. \tag{A.15}$$

References

Abdelzaher, T. F., E. M. Atkins, and K. Shin (2000): “QoS negotiation in real-time systems and its application to automated flight control.” *IEEE Transactions on Computers*, **49:11**, pp. 1170–1183.

Abeni, L. (1998): “Server mechanisms for multimedia applications.” Technical Report RETIS TR98-01. Scuola Superiore S. Anna, Pisa, Italy.

Abeni, L. and G. Buttazzo (1998): “Integrating multimedia applications in hard real-time systems.” In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. Madrid, Spain.

Abeni, L. and G. Buttazzo (1999): “Adaptive bandwidth reservation for multimedia computing.” In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*. Hong Kong, P.R. China.

Abeni, L., L. Palopoli, G. Lipari, and J. Walpole (2002): “Analysis of a reservation-based feedback scheduler.” In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*.

Agrawal, M., D. Cofer, and T. Samad (2003): “Real-time adaptive resource management for advanced avionics.” *IEEE Control Systems Magazine*, **23:1**, pp. 76–86.

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

Albertos, P., A. Crespo, I. Ripoll, M. Vallés, and P. Balbastre (2000): “RT control scheduling to reduce control performance degrading.” In *Proceedings of the 39th IEEE Conference on Decision and Control*. Sydney, Australia.

Andersson, L. and A. Blomdell (1991): “A real-time programming environment and a real-time kernel.” In Asplund, Ed., *National Swedish*

Symposium on Real-Time Systems, Technical Report No 30 1991-06-21. Dept. of Computer Systems, Uppsala University, Uppsala, Sweden.

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems.* Prentice Hall.

Audsley, N., A. Burns, M. Richardson, and A. Wellings (1994): "STRESS—A simulator for hard real-time systems." *Software—Practice and Experience*, **24:6**, pp. 543–564.

Audsley, N., K. Tindell, and A. Burns (1993): "The end of the line for static cyclic scheduling." In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems.*

Balbastre, P., I. Ripoll, and A. Crespo (2000): "Control task delay reduction under static and dynamic scheduling policies." In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications.*

Baruah, S., G. Buttazzo, S. Gorinsky, and G. Lipari (1999a): "Scheduling periodic task systems to minimize output jitter." In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications.*

Baruah, S., D. Chen, and A. Mok (1999b): "Static-priority scheduling of multi-frame tasks." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems.*

Beccari, G., S. Caselli, M. Reggiani, and F. Zanichelli (1999): "Rate modulation of soft real-time tasks in autonomous robot control systems." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 21–28. York, England.

Bhattacharyya, S. S., E. Cheong, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, and H. Zheng (2002): "Heterogeneous concurrent modeling and design in Java." Technical Report UCB/ERL M02/23. Dept. EECS, University of California at Berkeley.

Blevins, P. and C. Ramamoorthy (1976): "Aspects of a dynamically adaptive operating system." *IEEE Transactions on Computers*, **25:7**, pp. 713–725.

Bollella, G., B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull (2000): *The Real-Time Specification for Java.* Addison-Wesley.

Brück, D., H. Elmqvist, S. E. Mattsson, and H. Olsson (2002): "Dymola for multi-engineering modeling and simulation." In *Proceedings of the 2nd International Modelica Conference*.

Burns, A., D. Prasad, A. Bondavalli, F. D. Giandomenico, K. Ramamritham, J. Stankovc, and L. Stringini (2000): "The meaning and role of value in scheduling flexible real-time systems." *Journal of Systems Architecture*, **46**, pp. 305–325.

Burns, A., K. Tindell, and A. J. Wellings (1994): "Fixed priority scheduling with deadlines prior to completion." In *Proceedings of the 6th Euromicro Workshop on Real-Time Systems*, pp. 138–142.

Burns, A. and A. Wellings (2001): *Real-Time Systems and Programming Languages*, 3rd edition. Addison-Wesley.

Buttazzo, G., G. Lipari, and L. Abeni (1998): "Elastic task model for adaptive rate control." In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 286–295.

Buttazzo, G. C. (1997): *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers.

Caccamo, M., G. Buttazzo, and L. Sha (2000a): "Capacity sharing for overrun control." In *Proceedings of the IEEE Real-Time Systems Symposium.* Orlando, Florida.

Caccamo, M., G. Buttazzo, and L. Sha (2000b): "Elastic feedback control." In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pp. 121–128. Stockholm, Sweden.

Casile, A., G. Buttazzo, G. Lamastra, and G. Lipari (1998): "Simulation and tracing of hybrid task sets on distributed systems." In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications.*

Crespo, A., I. Ripoll, and P. Albertos (1999): "Reducing delays in RT control: The control action interval." In *Proceedings of the 14th IFAC World Congress*, pp. 257–262.

Crnkovic, I. and M. Larsson, Eds. (2002): *Building Relable Component-Based Software Systems.* Artech House Publishers.

Eker, J. and A. Cervin (1999): "A Matlab toolbox for real-time and control systems co-design." In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pp. 320–327. Hong Kong, P.R. China.

Eker, J., A. Cervin, and A. Hörjel (2001): "Distributed wireless control using Bluetooth." In *Proceedings of the IFAC Conference on New Technologies for Computer Control*. Hong Kong, P.R. China.

Eker, J., P. Hagander, and K.-E. Årzén (2000): "A feedback scheduler for real-time control tasks." *Control Engineering Practice*, **8:12**, pp. 1369–1378.

Eker, J. and J. Malmborg (1999): "Design and implementation of a hybrid control strategy." *IEEE Control Systems Magazine*, **19:4**.

El-khoury, J. and M. Törngren (2001): "Towards a toolset for architecural design of distributed real-time control systems." In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*.

Fletcher, R. (1987): *Practical Methods of Optimization*. Wiley.

Gerber, R. and S. Hong (1993): "Semantics-based compiler transformations for enhanced schedulability." In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pp. 232–242.

Gerber, R. and S. Hong (1997): "Slicing real-time programs for enhanced schedulabilty." *ACM Transactions on Programming Languages and Systems*, **19:3**, pp. 525–555.

Gerber, R., S. Hong, and M. Saksena (1995): "Guaranteeing real-time requirements with resource-based calibration of periodic processes." *IEEE Trans on Software Engineering*, **21:7**.

Gill, C. D., D. L. Levine, and D. C. Schmidt (1998): "Dynamic scheduling strategies for avionics mission computing." In *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference*.

Gutierrez, J. and M. Harbour (1998): "Schedulability analysis for tasks with static and dynamic offsets." In *Proceedings of the 19th IEEE Real-Time Systems Symposium*.

Hägglund, T. (1992): "A predictive PI controller for processes with long dead times." *IEEE Control Systems Magazine*, **12:1**, pp. 57–60.

Halang, W. (1993): "Achieving jitter-free and predictable real-time control by accurately timed computer peripherals." *Control Engineering Practice*, **1:6**, pp. 979–987.

Harbour, M. G., M. G., M. H. Klein, and J. P. Lehoczky (1994): "Timing analysis for fixed-priority scheduling of hard real-time systems." *IEEE Transactions on Software Engineering*, **20:1**, pp. 13–28.

Henriksson, D. and A. Cervin (2003): "TrueTime 1.1—Reference manual." Technical Report ISRN LUTFD2/TFRT--7605--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Henriksson, D., A. Cervin, J. Åkesson, and K.-E. Årzén (2002): "Feedback scheduling of model predictive controllers." In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*. San Jose, CA.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch (2001): "Giotto: A time-triggered language for embedded programming." In *Proceedings of the First International Workshop on Embedded Software*.

Jeffay, K. and D. L. Stone (1993): "Accounting for interrupt handling costs in dynamic priority systems." In *Proceedings of the 14th IEEE Real-Time Systems Symposium*.

Ji, Y., H. Chizeck, X. Feng, and K. Loparo (1991): "Stability and control of discrete-time jump linear systems." *Control-Theory and Advanced Applications*, **7:2**, pp. 247–270.

Joseph, M. and P. Pandya (1986): "Finding response times in a real-time system." *The Computer Journal*, **29:5**, pp. 390–395.

Klein, M. H., T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Härbour (1993): *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publisher.

Kleinrock, L. (1970): "A continuum of time-sharing scheduling algorithms." In *AFIPS Conference Proceedings, Spring Joint Computer Conference*, pp. 453–458.

Krasovskii, N. and E. Lidskii (1961): "Analytic design of controllers in systems with random attributes, I, II, III." *Automation and Remote Control*, **22:9–11**, pp. 1021–1025, 1141–1146, 1289–1294.

Krishna, C. M. and K. G. Shin (1997): *Real-Time Systems*. McGraw-Hill.

Lehoczky, J., L. Sha, and J. Strosnider (1987): "Enhanced apriodic responsiveness in hard real-time environment." In *Proceedings of the 8th IEEE Real-Time Systems Symposium*.

Leung, J. Y. T. and J. Whitehead (1982): "On the complexity of fixed-priority scheduling of periodic, real-time tasks." *Performance Evaluation*, **2:4**, pp. 237–250.

Li, B. and K. Nahrstedt (1998): "A control theoretical model for quality of service adaptations." In *Proceedings of the 6th International Workshop on Quality of Service*, pp. 145–153.

Lincoln, B. (2002a): “Jitter compensation in digital control systems.” In *Proceedings of the 2002 American Control Conference.*

Lincoln, B. (2002b): “A simple stability criterion for control systems with varying delays.” In *Proceedings of the 15th IFAC World Congress.*

Lincoln, B. and B. Bernhardsson (2000): “Efficient pruning of search trees in LQR control of switched linear systems.” In *Proceedings of the Conference on Decision and Control.*

Lipari, G. and S. Baruah (2000): “Greedy reclamation of unused bandwidth in constant-bandwidth servers.” In *Proceedings of the Euromicro Conference on Real-Time Systems.* Stockholm, Sweden.

Liu, C. L. and J. W. Layland (1973): “Scheduling algorithms for multiprogramming in a hard-real-time environment.” *Journal of the ACM*, **20:1**, pp. 40–61.

Liu, J. and E. Lee (2003): “Timed multitasking for real-time embedded software.” *IEEE Control Systems Magazine*, **23:1**.

Liu, J. W. S. (2000): *Real-Time Systems.* Prentice Hall.

Locke, C. D. (1992): “Software architecture for hard real-time applications: Cyclic vs. fixed priority executives.” *Real-Time Systems*, **4**, pp. 37–53.

Lu, C., J. Stankovic, T. Abdelzaher, G. Tao, S. Son, and M. Marley (2000): “Performance specifications and metrics for adaptive real-time systems.” In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pp. 13–23.

Lu, C., J. Stankovic, G. Tao, and S. H. Son (1999): “Design and evaluation of a feedback control EDF scheduling algorithm.” In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pp. 56–67.

Lu, C., J. A. Stankovic, S. H. Son, and G. Tao (2002): “Feedback control real-time scheduling: framework, modeling and algorithms.” *Real-Time Systems*, **23:1/2**, pp. 85–126.

Marti, P., G. Fohler, K. Ramamritham, and J. M. Fuertes (2001): “Jitter compensation for real-time control systems.” In *Proceedings of the 22nd IEEE Real-Time Systems Symposium.*

MathWorks (2001): *Simulink: A Program for Simulating Dynamic Systems—User’s Guide.* The MathWorks Inc.

Nilsson, J. (1998a): *Real-Time Control Systems with Delays.* PhD thesis ISRN LUTFD2/TFRT--1049--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Nilsson, J. (1998b): "Two toolboxes for systems with random delays." Technical Report ISRN LUTFD2/TFRT--7572--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Palopoli, L., L. Abeni, and G. Buttazzo (2000): "Real-time control system analysis: An integrated approach." In *Proceedings of the 21st IEEE Real-Time Systems Symposium.*

Palopoli, L., C. Pinello, A. Sangiovanni-Vincentelli, L. El-Ghaoui, and A. Bicchi (2002): "Synthesis of robust control systems under resource constraints." In *Proceedings of the Workshop on Hybrid Systems: Computation and Control.*

Parekh, A. and R. Gallager (1993): "A generalized processor sharing approach to flow control in integrated services networks: the single node case." *IEEE/ACM Transactions on Networking*, **1:3**, pp. 344–357.

Persson, P., A. Cervin, and J. Eker (2000): "Execution-time properties of a hybrid controller." Technical Report ISRN LUTFD2/TFRT--7591--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Potier, D., E. Gelenbe, and J. Lenfant (1976): "Adaptive allocation of central processing unit quanta." *Journal of the ACM*, **23:1**, pp. 97–102.

Rajkumar, R., C. Lee, J. Lehoczky, and D. Siewiorek (1997): "A resources allocation model for QoS management." In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 298–307.

Redell, O. and M. Sanfridson (2002): "Exact best-case response time analysis of fixed priority scheduled tasks." In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*. Vienna, Austria.

Redell, O. and M. Törngren (2002): "Calculating exact worst-case reponse times for static priority scheduled tasks with offsets and jitter." In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. San Jose, California.

Rehbinder, H. and M. Sanfridson (2000): "Integration of off-line scheduling and optimal control." In *Proceedings of the 12th Euromicro Conference on Real-Time Systems.*

Ryu, M. and S. Hong (1998): "Toward automatic synthesis of schedulable real-time controllers." *Integrated Computer-Aided Engineering*, **5:3**, pp. 261–277.

Ryu, M., S. Hong, and M. Saksena (1997): "Streamlining real-time controller design: From performance specifications to end-to-end timing constraints." In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pp. 91–99.

Seto, D., J. P. Lehoczky, L. Sha, and K. G. Shin (1996): "On task schedulability in real-time control systems." In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 13–21. Washington, DC.

Shin, K. and C. Meissner (1999): "Adaptation of control system performance by task reallocation and period modification." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 29–36.

Sprunt, B., L. Sha, and J. Lehoczky (1989): "Aperiodic task scheduling for hard real-time systems." *Real-Time Systems*.

Spuri, M. and G. Buttazzo (1996): "Scheduling aperiodic tasks in dynamic priority systems." *Real-Time Systems*, **10:2**, pp. 179–210.

Stankovic, J. A., C. Lu, S. H. Son, and G. Tao (1999): "The case for feedback control real-time scheduling." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 11–20.

Stankovic, J. A., M. Spuri, K. Ramamritham, and G. C. Buttazzo (1998): *Deadline Scheduling for Real-Time Systems—EDF and Related Algorithms*. Kluwer Academic Publishers.

Storch, M. F. and J. W.-S. Liu (1996): "DRTSS: A simulation framework for complex real-time systems." In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*.

Vidal, J., A. Crespo, and P. Balbastre (2002): "Control task implementation in rtlinux." In *Proceedings of the 15th IFAC World Congress*.

Wittenmark, B. and K. J. Åström (1980): "Simple self-tuning controllers." In Unbehauen, Ed., *Methods and Applications in Adaptive Control*, number 24 in Lecture Notes in Control and Information Sciences, pp. 21–29. Springer-Verlag, Berlin, FRG.

Zhao, Q. C. and D. Z. Zheng (1999): "Stable and real-time scheduling of a class of hybrid dynamic systems." *Journal of Discrete Event Dynamical Systems*, **9:1**, pp. 45–64.