Flexible Scheduling Methods and Tools for Real-Time Control Systems

Henriksson, Dan

2003

Document Version:
Publisher's PDF, also known as Version of record

Flexible Scheduling Methods and Tools for Real-Time Control Systems

Dan Henriksson

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

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

ISSN 0280–5316
ISRN LUTFD2/TFRT--3233--SE

Printed in Sweden,
Lund University, Lund 2003

Contents

Acknowledgments

First of all, I would like to thank my supervisor Karl-Erik Årzén for his support, inspiration, and insightful ideas throughout my work. I am also very grateful for the work he is doing to acquire funding for myself and other graduate students.

A special thanks goes to Anton Cervin who originally got me into the field of real-time systems, and who has been the ultimate colleague in every respect. Anton is also gratefully acknowledged for all the money he has bluffed away during our poker sessions. Many thanks also to Johan Åkesson for the co-operation and fruitful discussions we had during the work on the MPC delay compensation.

This summer I had the pleasure to spend three months at the University of Virginia working with Tarek Abdelzaher. This was a very rewarding visit, and I hope for continued collaboration in the future.

I would like to thank all my colleagues at the department for the great atmosphere you all provide. It is a privilege working with you! My fellow innebandy friends at the department are worth a special mention for never giving up. Who knows, maybe some time in a not so distant future we will be able to capture the much coveted t-shirt.

The work has been sponsored by the Swedish Foundation for Strategic Research via the program FLEXCON, the Swedish Research Council, and by LUCAS — the VINNOVA-funded Center for Applied Software Research.

The development of TrueTime has also been funded by ARTES (A network for Real-Time research and graduate Education in Sweden). ARTES is also gratefully acknowledged for the travel grant that enabled my research visit to University of Virginia.

Finally I will thank my friends and family for their encouragement and support, and Elinore for her patience and love.

Dan

1

Introduction

1.1 Motivation

Embedded micro-computers are increasingly being deployed in modern engineering applications, and real-time control systems constitute an important subclass of these embedded systems. Modern automotive systems, e.g., contain several embedded ECUs (electronic control units) used for various feedback control tasks, such as engine performance control, anti-lock braking, active stability control, exhaust emission reduction, and cruise control.

Real-time control systems have traditionally been relatively static systems operating in closed environments under well-defined load conditions. However, this situation is changing rapidly. The complexity of the control systems is increasing, and the design process of these system often involves many conflicting objectives, including cost, performance, reliability, and safety.

Market requirements, such as reduced time-to-market and lower development costs, often is the decisive factor in the process. Hence, as a result of economic considerations, many embedded control systems are subject to resource constraints, manifesting itself by limited CPU speed, memory, and network bandwidth of the target platform. In addition, a strong trend within industry today is to use commercially available IT technology and commercial-off-the-shelf (COTS) components deeper and deeper in the real-time control systems.

Limited resources combined with non-optimized hardware and software components introduce nondeterminism in the real-time system. For control systems this is of particular concern. Timing variations in sampling periods and latencies degrade the control performance and may in extreme cases lead to instability. Further adding to the nondeterminism is the fact that most embedded control systems are implemented using distributed architectures, where the sensor, actuator and control functionality is located on different nodes connected by a communication network.

In highly safety-critical applications, such as nuclear power plants and fly-by-wire systems, the main objective in the software design is to maximize the determinism in order to guarantee predictable behavior. This requires static design methodologies, including scheduling by static, cyclic executives [Locke, 1992], and time-triggered architectures, such as TTA [Kopetz, 1997].

For the majority of control systems, however, the drawbacks of using a static design vastly outweigh the benefits. While the static techniques increase the predictability and allow for off-line guarantees, they also reduce the flexibility and limit the possibilities for dynamic modifications. Instead less rigid approaches are called for, including dynamic task scheduling, communication protocols, and memory management. The projected advantages of using this approach include: more efficient use of the available resources thereby allowing the use of cheaper hardware, the possibility to dynamically adapt to changing load conditions, and higher obtainable control performance under the given resource constraints.

The key in obtaining flexibility is co-design of the control system and the real-time system. Integrating control theory and real-time scheduling theory [Cervin, 2003], it is possible to take the constraints of the target platform into consideration in the controller design, and to develop scheduling schemes specially tailored towards control tasks.

One promising approach is dynamic run-time flexibility by the introduction of feedback in the real-time system. Treating the control performance as a quality-of-service parameter that should be maximized, resources may be dynamically allocated to the controller tasks based on measurements of actual resource consumption. A method for manipulating execution times in order to maximize performance for model predictive controllers is presented in this thesis.

To aid in the development process, computer-based tools for simulation, analysis, and synthesis of real-time control systems are needed. A simulator that allows complete simulation of the interaction between real-time tasks, network transmissions, and continuous-time plant dynamics, is another topic of this thesis.

1.2 Outline and Related Publications

This section contains the outline of the rest of the thesis, together with references to related publications.

Chapter 2: Background

This chapter gives a short overview of computer-based control and real-time scheduling and their interaction. This includes control loop timing issues, control and scheduling co-design, and a summary of existing simulation tools for real-time control systems.

Chapter 3: Feedback Scheduling

The chapter deals with dynamic run-time scheduling techniques for real-time control systems. A general feedback scheduling structure is detailed and possible sensors and actuators are identified. Scheduling techniques specially tailored for certain control algorithms are described.

Publications

Årzén, K.-E., A. Cervin, and D. Henriksson (2003): "Resource-constrained embedded control systems: Possibilities and research issues." In *Proceedings of CERTS'03 – Co-design of Embedded Real-Time Systems Workshop*. Porto, Portugal.

Chapter 4: Flexible Implementation of Model Predictive Control

This chapter describes a flexible implementation and scheduling approach for model predictive controllers (MPCs). The control signal of an MPC is computed by on-line optimization of a cost function in every sample. The iterative nature of the control algorithm allows for a trade-off between computational delay and the quality of the obtained

control signal. The trade-off is quantified by a delay-dependent termination criterion rendering a sub-optimal, yet stabilizing, MPC formulation. Unlike traditional MPC, the effects of computational delay is taken into consideration in the optimization. A dynamic scheduling policy based on the MPC cost functions is also described.

Publications

Henriksson, D., J. Åkesson, and K.-E. Årzén (2004): "Flexible real-time implementation of model predictive control using sub-optimal solutions." Submitted to the 2004 American Control Conference, Boston, MA.

Preliminary simulation studies were 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.

Henriksson, D., A. Cervin, J. Åkesson, and K.-E. Årzén (2002): "On dynamic real-time scheduling of model predictive controllers." In *Proceedings of the 41st IEEE Conference on Decision and Control*. Las Vegas, NV.

The work in this chapter represents joint work with Johan Åkesson. Åkesson provided the tools used for implementation and analysis of the MPC controller. Henriksson conducted the real-time simulations, using the TrueTime simulator. The delay compensation and dynamic scheduling schemes were developed in close collaboration between the authors.

Chapter 5: The TrueTime Simulator

In this chapter the simulation tool TrueTime is presented. The simulator is based on MATLAB/Simulink and allows for co-simulation of controller task execution in real-time kernels, network communication, and continuous-time plant dynamics. A general description of the simulator is given and the event-based kernel implementation is detailed.

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.

Cervin, A., D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén (2003): "How does control timing affect performance?" *IEEE Control Systems Magazine*, **23:3**, pp. 16–30.

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, and K.-E. Årzén (2003): "TrueTime: Real-time control system simulation with MATLAB/Simulink." In *Proceedings of the Nordic MATLAB Conference*. Copenhagen, Denmark.

The simulator work represents joint work with Anton Cervin, who also implemented the first prototype of TrueTime together with Johan Eker. Cervin has implemented the major parts of the network block, whereas Henriksson has implemented the TrueTime kernel block. The publications have been written in close collaboration between the authors.

Chapter 6: Simulation Case Studies

This chapter contains two simulation case studies performed using the TrueTime simulator. The first case study simulates networked control of a robot system. It is shown how transport layer network protocols such as TCP may be implemented on top of the MAC layer protocols provided by the TrueTime network block. The second case study uses TrueTime to simulate a web server application. A feedback scheduling scheme based on schedulability results for aperiodic tasks is used to control the delays of individual connections to the server. The performance of the scheme is evaluated using the simulator.

Publications

Henriksson, D., Y. Lu, and T. Abdelzaher (2004): "Improved prediction for web server delay control." Submitted to the 16th Euromicro Conference on Real-Time Systems, Catania, Sicily, Italy.

Chapter 7: Conclusions

The thesis concludes with a summary and suggestions for future work in the field.

2

Background

2.1 Introduction

The advances in micro-electronics, and the increasing speed of microprocessors during the last 30 years, has led to a situation where today almost all control algorithms are realized by computers. The design of real-time control systems, therefore, requires inter-disciplinary knowledge of both control engineering and computer science and especially their inter-relations.

In the early days of computer control, implementation issues related to the computing hardware were well-known problems among control engineers [Hanselmann, 1987]. However, as the computing power has increased and is continuing to increase, implementation issues such as, e.g., real-time scheduling are often dismissed as non-problems. While this might be true for desktop computers, the situation is different when it comes to embedded systems and embedded control systems in particular.

Embedded control systems are most often subject to limited computer resources as a result of economic considerations. This combined with the trend of having more and more functionality being realized in software, make resource scheduling and its effect on control performance a relevant issue. However, traditionally, there has been a separation between the control and computer science communities in their view of real-time control systems.

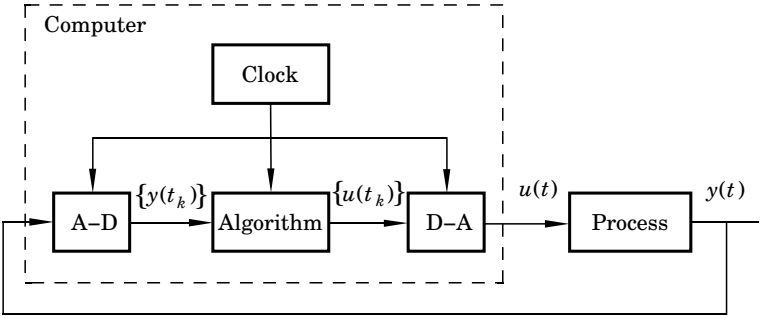

Figure 2.1 Schematic diagram of a computer-controlled system.

In the control community timing effects caused by the hardware platform are generally not taken into account in the controller design. Instead, computer-based control theory is based on assumptions of equidistant sampling instants and a zero or constant delay between sampling of the measurements and actuation of the control signal.

Real-time scheduling theory, on the other hand, is concerned with providing hard timing guarantees, and often use control tasks as their prime example of a hard real-time system. The objective is to make sure that no deadlines are missed, and the actual impact of the scheduling on the timing performance of the application is seldom considered.

Given this, it should come as no surprise that large improvements in control performance can be achieved by considering the control design and the real-time scheduling design at the same time. This way it is possible to make maximum use of limited computing resources and to optimize the control performance.

This chapter recaptures basic concepts of computer-based control and implementation of real-time control systems. This includes timing variations and their effect on the control system performance and stability, traditional real-time scheduling design, and a section on integrated approaches to real-time control and scheduling. The chapter concludes with a summary and comparison of available simulation tools for real-time control system co-design.

2.2 Computer-Based Control and Implementation

The basic structure of a computer-based control system is shown in Figure 2.1. The continuous process output is sampled at regular time intervals and converted to digital form by an A/D-converter. The control algorithm reads the sampled process output and computes a control signal that is converted back to analog form by a D/A-converter. The D/A-conversion is usually performed by keeping the output constant between conversions, so called *zero-order-hold*.

The standard implementation of a periodic control loop is given by the pseudo code in Listing 2.1. The control algorithm is often designed using sampled-data control methods, see, e.g., [Åström and Wittenmark, 1997]. Normally, the reading of inputs and writing of output signals correspond to direct calls to external A/D and D/A conversion interfaces. However, it is also possible to have the sampling and actuation being performed by dedicated tasks, in which case buffers often are used to communicate the values between the tasks. In the case of a networked control system the reading and writing of signals also involve communication with other nodes in the network.

To minimize the input-output delay, the control algorithm is often divided into two parts, where the first part computes the control signal based on current measurements and previous states. The second part then updates the internal states of the controller for the next sample.

Listing 2.1 A standard implementation of a periodic control loop.

```
t = currentTime();
LOOP
  Read Inputs;
  Control Calculation;
  Write Outputs;
  Update Internal States;
  t = t + h;
  waitUntil(t)
END
```

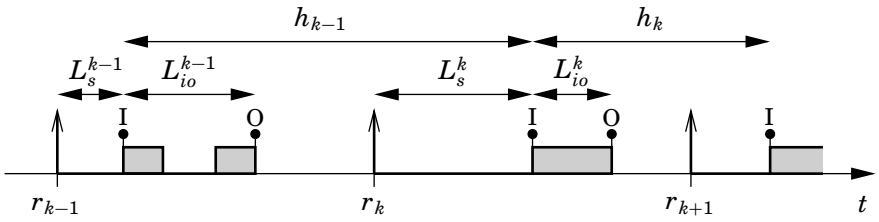

Figure 2.2 Controller timing.

2.3 Controller Timing

Computer-based control theory is based on idealized assumptions about perfect sampling periodicity and constant or negligible control delays. This can, however, seldom be achieved by the practical implementation in a resource-constrained system.

Within individual computer nodes, e.g., tasks interfere with each other through preemption and blocking when waiting for common resources. The execution times of the control tasks may be data-dependent or vary due to hardware features such as caches. On the distributed level, the communication gives rise to delays that can be more or less deterministic depending on the communication protocol.

The resulting timing properties between the reading of the inputs and the generation of the outputs is a crucial factor for the performance of the controlled system. The timing variations introduced by the computer system may lead to substantial performance degradation, and even instability. The basic timing variations experienced by control tasks are depicted in Figure 2.2.

Input-Output Latency

The delay between the sampling of the measurement signal and the output of the control signal is called the *input-output latency*, denoted L_{io}. This delay has the same effect on the closed-loop system as a process input delay, and may compromise the overall system stability if not handled properly.

Input-output latency is primarily caused by preemption from higher-priority tasks, and by the execution time of the control algorithm itself.

The traditional way to minimize input-output latency is by separation of the control algorithm into *calculate* and *update* parts, as shown in Listing 2.1. Input-output latency can also be reduced by using non-preemptive scheduling. With the increasing speed of modern computers it can be argued that the relative execution times of different tasks will become smaller and smaller, thus making this approach realistic also from a schedulability point-of-view.

Jitter

The periodic task that implements the control algorithm is released at equidistant time intervals given by $r_k = hk$, where h is the sampling interval of the controller. However, the scheduling may cause the actual start of the task to be delayed some time. This time is known as the sampling latency of the task, denoted L_s. Variation in the sampling latency is called *sampling jitter*. Sampling jitter will also cause jitter in the actual sampling period.

Another source of jitter is variations in the input-output latency, called *input-output jitter*. This is often caused by variations in the execution time of the control algorithm. For simple controllers such as PID-controllers these variations are negligible, whereas more advanced algorithms may have very large execution time variations. One example is model predictive controllers (further treated in Chapter 4), where a constrained quadratic programming problem is solved on-line in every sample. The effects of varying delays are often very difficult to analyze. [Lincoln, 2003] treats methods for analyzing stability properties of systems with varying delays. Dynamic compensation schemes are also presented.

One way to remove the sampling jitter altogether is by performing the sampling operation in a dedicated high-priority task. This way the sampling is always performed at the right time instants. However, while this technique removes the sampling jitter it instead increases the input-output latency.

Using this approach, the input-output jitter can also be eliminated by always delaying the output to the end of the period, thus introducing a constant one-sample delay in the system. This delay may then be compensated for in the controller design. However, as shown in [Cervin, 2003], the compensation may only recover a part of the loss introduced by the added input-output latency. Therefore, for controllers

with highly varying execution times, designing and compensating for the worst-case execution time is not a viable option.

2.4 Real-Time Scheduling

Real-time scheduling theory is concerned with the problem of, given a set of tasks, finding an execution order that guarantees that all tasks meet their timing constraints. Real-time scheduling algorithms fall in two basic categories; *static* and *dynamic* scheduling.

Static scheduling is an off-line approach, where an optimized execution order is determined once and for all before the system is commissioned. This execution order is then repeated cyclically at run-time. The main benefit of this approach is that it is easy to analyze and thereby guarantee all timing requirements. The main drawback is that the cyclic schedules may be very long and difficult to obtain. They also need to be re-calculated every time changes are made to the real-time system.

In dynamic scheduling schemes, the decision of which task to run is taken at run-time. The standard and still most commonly used dynamic scheduling schemes were presented in the seminal paper [Liu and Layland, 1973]. The schedulability theory is based on a task model where all tasks are periodic and where each task, i, is characterized by the following parameters

- a fixed period, T_i,
- a hard deadline, D_i,
- and a fixed and known worst-case execution time (WCET), C_i.

Fixed-Priority Scheduling

Fixed-priority scheduling is the most common scheduling mechanism and is supported by all major commercial real-time operating systems. Using this approach, each task is assigned a fixed priority value. During run-time, the ready task with the highest priority gets access to the CPU. If a task with a lower priority is currently running, this task is preempted by the higher priority task.

For control tasks it is natural to assume that the relative deadlines, D_i, of the tasks are equal to their periods, T_i. In this case the most common priority assignment is the *rate-monotonic* assignment, where the priorities are set according to the periods of the tasks. The shorter the period, the higher the priority.

It is shown in [Liu and Layland, 1973] that this is an optimal scheduling policy, i.e., if the task set is not schedulable using rate-monotonic assignment it is not schedulable using any other fixed-priority assignment either.

Assuming a set of n tasks, a sufficient condition for schedulability using the rate-monotonic priority assignment is that the utilization factor

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

In the more general case where $D_i \leq T_i$, deadline-monotonic priority assignment is optimal [Liu and Layland, 1973]. Here the priorities are assigned according to the relative deadlines of the tasks.

For any fixed priority scheduling assignment, an exact schedulability analysis may be performed by computing the worst-case response times, R_i, for each task, see [Joseph and Pandya, 1986].

Earliest-Deadline-First Scheduling

Using fixed-priority assignment the priorities of the tasks are static and not changed during run-time. An alternative approach is *earliest-deadline-first* (EDF) scheduling which exploits dynamic priority assignment based on the absolute deadlines of the tasks. At any point in time, the task with the shortest remaining time to its deadline will get access to the CPU.

EDF is more resource-effective than rate-monotonic scheduling and a necessary and sufficient condition for schedulability (given $D_i = T_i$) is that the utilization factor is below one:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.2}$$

A benefit of deadline-based scheduling over priority-based scheduling is that it is usually more intuitive to assign deadlines to tasks than

to assign priorities. To assign priorities, global information about the relative importance of all tasks in the system is needed, which is not required to assign deadlines.

The main drawback with EDF is that it offers no guarantees at all during overload. In that case all tasks will miss their deadlines, which is known as the domino effect [Stankovic *et al.*, 1998]. For hard real-time systems this may be fatal. However, the result during overload under EDF, is that the effective periods of the tasks will be scaled in such a way that the utilization of the system is still 100 per cent [Cervin *et al.*, 2002]. Under reasonable overload, this fair distribution of resources will for most control systems still give reasonable performance for all loops.

Scheduling of Aperiodic Tasks

In many applications, the assumption of purely periodic tasks does not hold. An important example is web server systems, which handle large volumes of aperiodically arriving requests. With individual requests are often associated specific quality-of-service (QoS) requirements related to their deadlines. Motivated by this, schedulability bounds for aperiodic tasks has been an active research area during recent years. Schedulability bounds for aperiodic tasks were presented in [Abdelzaher and Lu, 2001].

The bounds are based on a measure called *synthetic utilization*, $U^{\zeta}(t)$, defined as

$$U^{\zeta}(t) = \sum_{i \in V^{\zeta}(t)} \frac{C_i}{D_i} \tag{2.3}$$

where $V^{\zeta}(t)$ is the set of *current* tasks at time t, i.e., tasks that have arrived but whose deadlines have yet to expire.

It can be proven that deadline-monotonic scheduling is an optimal policy for aperiodic tasks. Using this assignment all tasks will meet their deadlines if, $\forall t$

$$\begin{aligned} U^{\zeta}(t) &< \frac{1}{2} + \frac{1}{2n}, \quad \text{for } n < 3 \\ U^{\zeta}(t) &< \frac{1}{1 + \sqrt{\frac{1}{2}(1 - \frac{1}{n-1})}}, \quad \text{for } n \geq 3 \end{aligned} \tag{2.4}$$

2.5 Integrated Control and Real-Time Scheduling

Many of the assumptions made in the control and real-time scheduling communities are either too restrictive or too idealized to describe the actual behavior of real-time control loops.

E.g., the standard hard real-time task model used in the real-time scheduling community does not capture the special requirements of control tasks. While it is true that most hard real-time systems are control systems, most control systems are not hard real-time systems. For almost all controllers, single missed deadlines are not critical for the system performance or system stability.

On the other hand, the assumptions made in computer-based control theory do not consider the effects of the actual implementation of the controller as a task in a real-time system. The timing variations introduced by the computer system are crucial for the performance of the control system and must be taken into account at design time.

Consequently, we realize that design of real-time control systems is essentially a co-design problem. For optimal use of limited computing resources and for optimal control performance, the controller design and the software design need to go hand in hand. Two promising approaches to control and scheduling co-design are reservation-based scheduling and feedback scheduling. For more on integrated control and real-time scheduling, see [Cervin, 2003].

Reservation-Based Scheduling

The concept of server-based scheduling has recently gained much interest in the real-time scheduling community. In the constant bandwidth server (CBS) [Abeni and Buttazzo, 1998], e.g., the CPU is conceptually divided into a number of virtual sub-CPUs with given capacities, U_s. The CBS then guarantees that tasks running in the virtual CPUs never consume more than the alloted capacity.

The control server, an extension of CBS tailored for control tasks, is presented in [Cervin, 2003; Cervin and Eker, 2004]. A control server creates the abstraction of a control task with a specified period and a fixed input-output latency shorter than the period. The control server model is well suited for co-design in that the single parameter linking the scheduling design and the controller design is the task utilization factor.

Feedback Scheduling

Feedback scheduling is an approach to achieve flexibility in the run-time scheduling of control tasks. The objective is to optimize the control performance for control loops under resource constraints. In feedback scheduling, the available resources are scheduled dynamically based on measurements of actual timing variations and control performance. An elaborate discussion of feedback scheduling will be given in Chapter 3.

2.6 Simulation Tools

To aid in the development process, new, computer-based tools for real-time and control system co-design are needed. However, the separation between the control community and the real-time scheduling community is also apparent when it comes to existing simulation tools for this type of systems.

The main simulation tool used for control system design and simulation is MATLAB/Simulink [The Mathworks, 2001b]. Also, during recent years, Modelica [Tiller, 2001] has emerged as a strong alternative to MATLAB/Simulink when it comes to physical modeling and simulation. However, neither of these simulation environments have sufficient support for simulation of real-time implementation issues. Real-Time Workshop [The Mathworks, 2001a] allows prototyping and implementation of real-time control systems, but has very limited support for simulation of shared CPU resources and no support for simulation of networks.

On the other hand, several tools exist for simulation of real-time scheduling. Examples include STRESS [Audsley *et al.*, 1994] and PERTS/DRTSS [Storch and Liu, 1996]. These tools are typically used to prove feasibility of task sets and to perform co-simulation of task execution and hardware architecture and kernels. The simulations do not capture the effects of the scheduling on the performance of the application implemented by the various tasks.

So, while numerous tools exist that support either simulation of control systems or simulation of real-time scheduling, very few tools support co-simulation of control systems and real-time scheduling. However, during the last years a few co-simulation tools have emerged.

The TrueTime simulator, which will be described thoroughly in Chapter 5, is a complete co-simulation tool based on MATLAB/Simulink. In its current version it supports task scheduling by arbitrary scheduling policies, network simulation by standard MAC layer protocols, and a variety of real-time primitives used for experimentation with flexible scheduling and compensation schemes.

An early, tick-based version of TrueTime was presented in [Eker and Cervin, 1999]. The event-based C++ implementation of the current version has decreased simulation times by orders of magnitude. This early version had no support for interrupt handling and being tick-based it could not handle fine-grained simulation details. Also, there was no support for simulation of networks.

The RTSIM real-time scheduling simulator (a stand-alone C++ program) has recently been extended with a numerical module (based on the Octave library) that supports simulation of continuous dynamics, see [Palopoli *et al.*, 2000]. However, it lacks a graphical plant modeling environment, and so far its network capabilities are limited.

The Ptolemy system developed at Berkeley, has recently added support for timed multi-tasking [Liu and Lee, 2003]. This makes it possible to model fixed-priority scheduling of tasks with constant execution times.

Another tool similar to TrueTime is the XILO toolset presented in [El-khoury and Törngren, 2001]. This tool is entirely graphical and currently limited to a number of pre-defined scheduling policies and network protocols provided in the tool libraries.

3

Feedback Scheduling

3.1 Introduction

The objective of feedback scheduling is to increase flexibility and to master uncertainty with respect to resource scheduling. A general feedback scheduling structure is shown in Figure 3.1. The idea is to feed back the actual use of critical resources to the scheduler and to continuously adjust the tasks' demands of resources according to the current situation. The reactive feedback may also be combined with pro-active feedforward actions, such as, e.g., task admission control schemes.

The feedback scheduled resources may be any computer resource, such as CPU time, network bandwidth, or memory allocation. E.g., an approach to achieve adaptive garbage collection and incorporate GC scheduling into a general feedback scheduling framework was presented in [Gestegård Robertz, 2003]. However, here we will focus on the scheduling of CPU time for real-time controller tasks.

The main motivation for the introduction of feedback-based scheduling for control tasks is the highly varying execution time characteristics associated with many control algorithms. Examples include model predictive controllers, controllers using vision-based sensor information, and hybrid controllers switching between different modes. For these control schemes, the execution time variations are inherent in the algorithms. Other sources to the variations may be external, such as changing environments or load conditions.

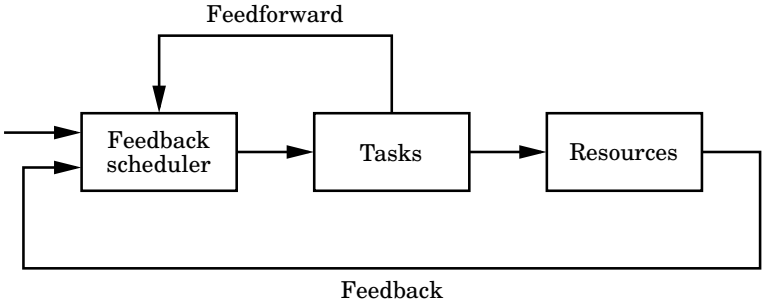

Figure 3.1 A general feedback scheduling system. The scheduler adjusts the tasks' demands based on feedback from the current use of critical resources. The tasks may also inform the scheduler that they are about to consume more resources (feedforward).

The varying execution times make traditional task scheduling as described in the previous chapter infeasible. Algorithms such as rate-monotonic and earliest-deadline-first are both *open-loop* scheduling algorithms, in the sense that the schedulability results are obtained off-line, assuming complete knowledge of the tasks and their constraints.

Since the execution time may vary significantly, the main limitation lies in the assumption of known worst-case execution time bounds for all tasks. A design based on worst-case bounds will likely become far too pessimistic, and lead to severe underutilization of the computer resources. In reality these bounds are also very difficult to obtain.

A feedback-based approach for task scheduling is presented in [Lu *et al.*, 2002], where control theory is used to provide performance guarantees for dynamically changing real-time systems. The guarantees are, however, related to deadline miss-ratios and utilization levels, and have no direct connection to the actual application performance.

For control systems, the decisive factor should be the control performance, and to distribute the resources in a way that optimizes the global control performance. For control tasks, there are two main ways to control the CPU demand: by manipulating the task periods, or by manipulating the execution times. The feedback should contain information related to the timing behavior and control performance of the controlled system.

3.2 Task Period Rescaling

The most commonly explored way to dynamically adjust CPU utilization for control tasks is by changing sampling periods. The classical rule-of-thumb, see, e.g., [Åström and Wittenmark, 1997], for selection of sampling interval in a digital control system is that

$$\omega_c h = 0.2 - 0.6, \tag{3.1}$$

where ω_c is the bandwidth of the closed-loop system. This means that a computer-based control system may operate according to specifications also using another sampling interval than originally designed for.

Dynamic resource allocation by means of task period rescaling has been explored in several papers. An adaptive rate control mechanism based on an elastic task model is presented in [Buttazzo *et al.*, 1998]. The task period adjustment is based on elasticity coefficients, e_i, related to the utilization factors of the tasks.

[Beccari *et al.*, 1999] considers modulation of sampling rates for robot systems. A range of admissible rates is identified for each task, and different rate-monotonic schemes are presented and evaluated.

[Shin and Meissner, 1999] studies resource adaptation in multiprocessor systems. Reallocation of control tasks and on-line adjustment of sampling rates is used to optimize a quadratic performance index related to the global control performance.

A feedback scheduled system manipulating sampling intervals can be viewed as a special case of a hybrid control system. An interesting example is given in [Schinkel *et al.*, 2002], which considers switching between two LQ-controllers designed with different sampling intervals. Although both closed-loop systems are stable, it is shown that a special switching sequence between the systems will lead to instability.

Optimization-Based Approaches

An optimal strategy for rescaling sampling periods for LQ-controllers was presented in [Eker *et al.*, 2000; Cervin, 2003]. Here it was shown that simple linear

$$J(h) = \alpha + \gamma h \tag{3.2}$$

or quadratic

$$J(h) = \alpha + \beta h^2 \tag{3.3}$$

cost functions are good approximations of how the actual control performance depends on the sampling interval. The sampling interval is denoted by h and α, β, and γ are constants.

Based on these functions an optimal feedback scheduling strategy was developed that minimized the global cost while meeting certain utilization set-points. The utilization was computed on-line based on execution time measurements.

The resulting optimal feedback scheduling scheme consisted of a simple rescaling of the nominal sampling periods, where all periods were changed by the same factor. This is a nice property, since it is fast and easy to implement. It also preserves the rate-monotonic ordering among the control tasks, and thus avoids priority changes of the tasks.

However, the cost functions only concern the sampling periods and not the actual input-output latencies. The feedback should ideally also contain feedback from the actual control performance and not only execution time measurements. Another problem arises when scheduling tasks that are described by both linear and quadratic cost functions. In this case the optimization becomes harder, and the linear rescaling property would be lost.

Mode Changes

In certain applications it is possible to combine the feedback scheduling with feedforward. In these schemes, the tasks themselves inform the scheduler that they are about to consume more resources. In [Cervin *et al.*, 2002], a case study with hybrid controllers is presented, where the sampling rates are adjusted to avoid CPU overloads. The controller changed between ordinary PID control and an optimal control mode with very different execution times. In this scheme, the controller tasks notified the scheduler when they were about to change mode.

From a schedulability aspect, mode changes may cause transient overloads, i.e., a system that is schedulable both before and after the mode change may still miss deadlines during the transient phase. This mode changing problem is treated in [Buttazzo *et al.*, 1998; Tindell *et al.*, 1992].

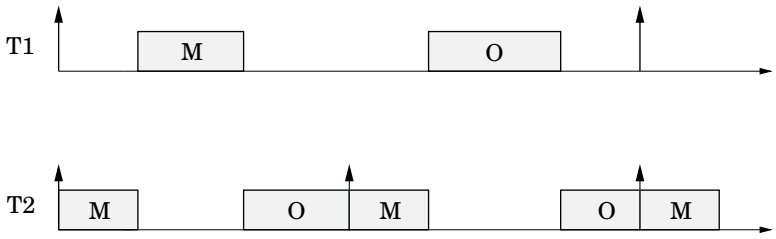

Figure 3.2 Scheduling of imprecise computations are based on a task model where each task can be divided into two parts; a mandatory part and an optional part. The optional part may be aborted to meet scheduling constraints or to optimize performance.

3.3 Scheduling of Imprecise Computations

For certain classes of control algorithms an alternative to sampling time adjustments is manipulation of the actual execution time of the control signal computation. These types of algorithms are generally referred to as *anytime algorithms* or imprecise computation algorithms.

The main characteristic of anytime algorithms is that they always generate a result, but with a quality level that increases with the execution time. This means that there is a trade-off to consider between the computational time and the result generated by the algorithm.

The basic task model for scheduling of imprecise computations [Liu *et al.*, 1991; Liu *et al.*, 1994] assumes that all tasks can be divided into two subtasks; a *mandatory* subtask and an *optional* subtask, see Figure 3.2. An imprecise result may be returned by the algorithm as long as the mandatory subtask has completed.

In [Liu *et al.*, 1991], imprecise calculation methods are categorized into three main types; *sieve function methods*, *multiple version methods*, and *milestone methods*.

Sieve functions constitute optional computation steps that may be skipped to save processing time. Obvious examples of sieve functions for control algorithms include the updating of the estimated parameters in an adaptive controller, or the observer step in an LQG-controller. Multiple version methods exploit several versions of the algorithm, with different processing times and result quality.

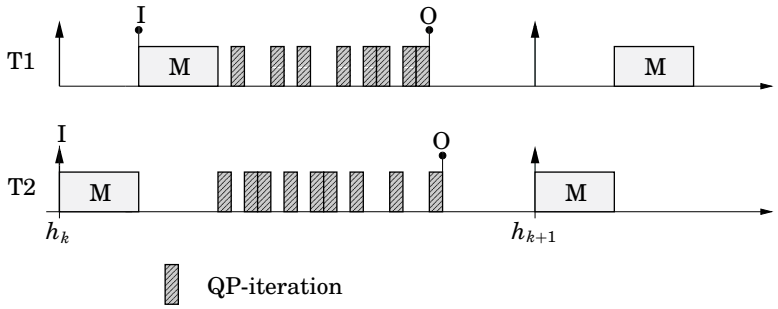

Figure 3.3 Imprecise computation model for model predictive control tasks. The mandatory part (M) consists of finding a feasible starting solution and iterating the QP-solver until the stability requirement is fulfilled. The additional QP-iterations constitute the optional part and may be skipped.

Milestone methods are based on monotone algorithms, ensuring that the quality of intermediate results increases monotonically with time. This type of algorithms can be found in many application areas, including numerical optimization, estimation, and prediction. Scheduling of monotone imprecise tasks is treated in [Chung *et al.*, 1990]. In this scheme, each mandatory subtask is scheduled to complete before the deadline of the task, and the optional parts refine the results to minimize the total error. Both average error between consecutive jobs, and cumulative errors are considered.

Application to Model Predictive Control

An example of a control methodology that fits the milestone method very well, is model predictive control (MPC), which is the topic of Chapter 4. This control strategy is based on on-line minimization in every sample of a quadratic cost function subject to constraints on control signals and controlled variables. In the MPC formulation used in Chapter 4, the optimization problem is solved by an iterative QP-solver that guarantees that the value of the cost function is reduced by each step in the algorithm.

The mandatory part of the control algorithm consists of finding a starting solution that fulfills the constraints of the QP-problem, and to iterate the solver until the solution guarantees closed-loop stability.

The optional part consists of the remaining QP-iterations that further reduce the value of the cost function. These iterations may be skipped if computing time is scarce.

Figure 3.3 illustrates a situation of two MPC tasks running concurrently. A dynamic scheduling strategy schedules the mandatory parts using distinct high priorities and the optional parts of the tasks using the cost functions as dynamic task priorities. By constantly executing the task with the highest cost we aim at achieving as low global cost as possible before the optional parts are terminated. Dynamic resource allocation for MPC tasks will be treated in detail in Chapter 4.

3.4 Direct Feedback Scheduling

A drawback with many dynamic resource allocation schemes is that the control performance is only affected indirectly by adjusting task parameters and assuring certain timing properties. The true effect on the control performance is often not easily determined from these parameters. Another approach is direct feedback scheduling, where scheduling decision are made based on instantaneous cost measures related to the control performance.

E.g., the previously described MPC approach with scheduling based on cost functions can be seen as a direct feedback scheduling strategy. A general direct feedback scheduling approach would typically be based on an instantaneous cost related to the control error. This includes derivatives and the integral of the control error, and quadratic cost terms of the error and control signal.

Just like in the imprecise task model, an implementation of a task used in a direct feedback scheduling context, would consist of two parts. The first part should contain the sampling of the process and evaluation of the instantaneous cost. The second part should then be optional and scheduled based on the value of the cost measure.

The resulting control system will run in open loop between invocations of the optional part, see Figure 3.4. A new control action is applied when the instantaneous cost causes the control task to be scheduled and not periodically as in traditional computer-based control.

The problem with the resulting control system is thus that it will be event-triggered and will therefore not follow the traditional model of

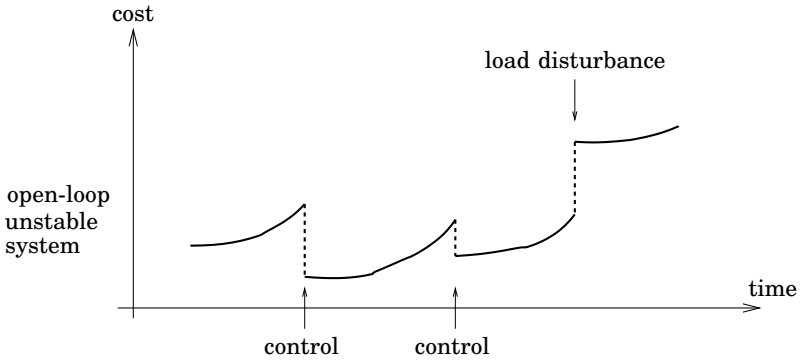

Figure 3.4 Direct feedback scheduling where scheduling decisions are made based on instantaneous cost measures for each controller tasks. The system runs in open loop between scheduling points and the resulting controlled system is event-triggered.

equidistant sampling instants for which the theory is well developed. Although many systems, including combustion engines and satellite control by thrusters, are naturally treated using an event-based approach, very little theory exist in the area. Event-based control systems have been explored in, e.g. [Åström and Bernhardsson, 1999; Årzén, 1999].

3.5 Quality-of-Service

Feedback scheduling is closely related to quality-of-service (QoS) approaches. Quality-of-service techniques for soft real-time activities, such as multimedia applications, have been an active research area during recent years.

For control systems it would be desirable to also treat the control performance as a quality-of-service parameter, or quality-of-control (QoC). This means that it would be necessary to specify reasonable ranges for the performance metrics, including, e.g., rise times, overshoots, and steady-state variances. The on-line resource negotiation could then, e.g., be specified using contracts [Eker and Blomdell, 2000] relating the control performance to the available resources.

One example is [Abdelzaher *et al.*, 2000], which considers quality-of-service negotiation in flight control systems. Here task periods and deadlines are treated as negotiable parameters between tasks in the system, allowing graceful QoS degradation under conditions where traditional schedulability analysis fails.

The use of feedback control theory has also recently emerged as a promising foundation for performance control in large, complex software applications. One prominent example is contemporary web servers, which typically operate under very unpredictable and poorly modeled load conditions. Managing this uncertainty by means of control theory has proven successful in order to provide quality-of-service guarantees for these systems [Abdelzaher *et al.*, 2003; Robertsson *et al.*, 2003].

3.6 Summary

This chapter has treated feedback-based approaches for scheduling of real-time controller tasks. The potential benefits of feedback scheduling are, e.g.,

- the possibility to relax the requirements on known worst-case execution time bounds,
- increased flexibility,
- higher resource utilization, and
- better control performance under the given resource constraints.

Feedback scheduling approaches that use task period rescaling, execution time manipulation, and scheduling based on instantaneous cost functions were treated. Quality-of-service based methods for control systems were also discussed.

4

Flexible Implementation of Model Predictive Control

4.1 Introduction

Model predictive control (MPC), see, e.g., [Garcia *et al.*, 1989; Richalet, 1993; Qin and Badgwell, 2003], is a control methodology that has been widely accepted industrially during recent years, mainly because of its ability to handle constraints explicitly and the natural way in which it can be applied to multi-variable processes.

The computational requirements of MPC, where typically a quadratic optimization problem is solved on-line in every sample, have previously prohibited its application in areas where fast sampling is required. Therefore MPC has traditionally only been applied to slow processes, mainly in the chemical industry. However, the advent of faster computers and the development of more efficient optimization algorithms, see, e.g., [Cannon *et al.*, 2001], has recently led to applications of MPC also to processes governed by faster dynamics. Some recent examples include [Dunbar *et al.*, 2002; Dunbar and Murray, 2002].

Still, from a real-time implementation perspective the execution time characteristics associated with MPC tasks poses many interesting challenges. Execution time measurements show that the computation time of an MPC controller varies significantly from sample to sample. The variations are due to, e.g., reference changes and exter-

nal disturbances. To cope with this, an increased level of flexibility is required in the real-time implementation. Because of the variations, a static compensation for the worst-case execution time would be too pessimistic and lead to unnecessary reduction of the obtainable control performance.

As described in the previous chapter, the MPC algorithm is of anytime nature, and fits nicely into the general framework of scheduling of imprecise computations. The milestone characteristics of the optimization algorithm makes it possible to abort the optimization before it has reached the optimum, and still fulfill the stability conditions. The key observation is that computational delay may significantly degrade control performance, and premature termination of the optimization algorithm may be advantageous over actually finding the optimum.

Stability of model predictive control algorithms has been the topic of much research in the field. For linear systems, the stability issue is well understood, and also for nonlinear systems there are results ensuring stability under mild conditions. For an excellent review of the topic, see [Mayne *et al.*, 2000]. In summary, there are two main ingredients in most stabilizing MPC schemes; terminal penalty and terminal constraint. These two tools has been used separately or in combination to prove stability for many existing MPC algorithms. It is also well known that feasibility, rather than optimality, is sufficient to guarantee stability, see, e.g., [Scokaert *et al.*, 1999]

In this chapter, the trade-off between computational delay and optimization is quantified by the introduction of a delay-dependent cost index. The index is based on a parameterization of the cost function in the MPC formulation. The objective of the optimization is then to minimize the cost index instead of the original cost function. This results in a delay-aware MPC formulation. The cost index is also applied in a real-time scheduling context.

The chapter concludes with two simulation case studies, where the suggested approaches are evaluated on a double-integrator process. The first simulation treats the trade-off between computational delay and optimization of a single MPC task. The second simulation considers scheduling of two MPC tasks concurrently on the same CPU.

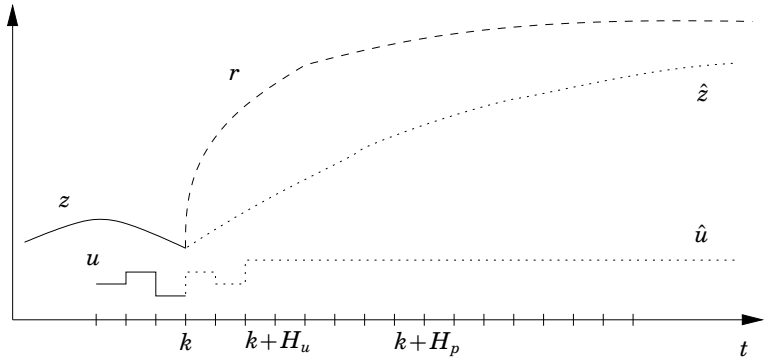

Figure 4.1 The basic principle of model predictive control.

4.2 MPC Formulation

The MPC formulation is based on [Maciejowski, 2002] and assumes a discrete linear process model on the form

$$\begin{aligned} x(k+1) &= \Phi x(k) + \Gamma u(k) \\ y(k) &= C_y x(k) \\ z(k) &= C_z x(k) + D_z u(k) \end{aligned} \tag{4.1}$$

where $y(k)$ is the measured output, $z(k)$ the controlled output, $x(k)$ the state vector, and $u(k)$ the input vector. The function to minimize at time k is

$$\begin{aligned} J(k, \Delta\mathcal{U}, x(k)) = &\sum_{i=1}^{H_p} \|\hat{z}(k+i|k) - r(k+i)\|_Q^2 \\ &+ \sum_{i=0}^{H_u-1} \|\Delta\hat{u}(k+i|k)\|_R^2 \end{aligned} \tag{4.2}$$

where $\hat{z}$ is the predicted controlled output, r is the current setpoint, $\hat{u}$ is the predicted control signal, H_p is the prediction horizon, H_u is the control horizon, $Q \geq 0$ and $R > 0$ are weighting matrices, and $\Delta u(k) = u(k) - u(k-1)$. It is assumed that $H_u < H_p$ and that $\hat{u}(k+i) = \hat{u}(k+H_u-1)$ for $i \geq H_u$. See Figure 4.1. $\Delta\mathcal{U} = \left(\Delta\hat{u}(k)^T \ldots \Delta\hat{u}(k+H_u-1)^T\right)^T$ is the solution vector.

Introducing sequences $\mathcal{U}$ and $\mathcal{Z}$ equivalently to $\Delta\mathcal{U}$, the state and control signal constraints may be expressed as

$$W\Delta\mathcal{U} \leq w \quad F\mathcal{U} \leq f \quad G\mathcal{Z} \leq g \tag{4.3}$$

This formulation leads to a convex linear-inequality constrained quadratic programming problem (LICQP) to be solved at each sample. The problem can be written on matrix form as

$$\min_{\theta} \; V(k) = \theta^T \mathcal{H}\theta - \theta^T \mathcal{G} + \mathcal{C} \quad \text{s.t.} \quad \Omega\theta \leq \omega. \tag{4.4}$$

where $\theta = \Delta\mathcal{U}$ and the matrices $\mathcal{H}$, $\mathcal{G}$, $\mathcal{C}$, Ω, and ω depend on the process model and the constraints, see [Maciejowski, 2002]. Only the first element of $\Delta\mathcal{U}$ is applied to the process and the optimization is then repeated in the next sample. This is referred to as the *receding horizon* principle.

Feasibility and Optimality

The problem of formulating stabilizing MPC schemes has received much attention in the last decade. For linear MPC, the conditions for stability are well understood, and several techniques for ensuring stability exist including terminal penalty, terminal equality constraint, and terminal sets, see [Mayne *et al.*, 2000]. For simplicity, we will use a terminal equality constraint to ensure stability, see, e.g., [Bemporad *et al.*, 1994].

The following theorem (adopted from [Bemporad *et al.*, 1994]) summarizes the important features of a stabilizing MPC scheme based on a terminal equality constraint. Without lack of generality we assume that $r(k)$ is zero.

THEOREM 1
Consider the system (4.1) controlled by the receding horizon controller based on the cost function (4.2), subject to the constraints (4.3). Let $r(k)$=0. Further assume terminal constraints $\hat{x}(k + H_p + 1)$=0 and $\hat{u}(k+H_u)$=0, $Q\geq$0 and $R>$0 and that $(C_z Q^{\frac{1}{2}}, A)$ is a detectable pair. If the optimization problem is feasible at time k, then the origin is stable, and $z(k)^T Q z(k) \to 0$ as $k \to \infty$.

Proof. Let $\Delta\mathcal{U}_k^*=(\Delta\hat{u}_k^*(k), \Delta\hat{u}_k^*(k+1), \ldots, \Delta\hat{u}_k^*(k+H_u-1))$ denote the optimal control sequence at time k. Obviously, $\Delta\mathcal{U}_{k+1}=(\Delta\hat{u}_k^*(k+1), \ldots, \Delta u_k^*(k+H_u-1),\ 0)$ is then feasible at time $k+1$. Consider the function $V(k)=J(k, \Delta\mathcal{U}_k^*, x(k))$ with $r(k)=0$. Then we have the following relations:

$$\begin{aligned} V(k+1) &= J(k+1, \Delta\mathcal{U}_{k+1}^*, x(k+1)) \\ &\leq J(k+1, \Delta\mathcal{U}_{k+1}, x(k+1)) \\ &= V(k) - z(k+1)^T Q z(k+1) \\ &\quad - \Delta u(k)^T R \Delta u(k). \end{aligned} \tag{4.5}$$

Since $V(k)$ is lower-bounded and decreasing, $z(k)^T Q z(k) \to 0$ and $\Delta u(k)^T R \Delta u(k) \to 0$ as $k \to \infty$. Further, using the fact that $(C_z Q^{\frac{1}{2}}, A)$ is a detectable pair, it follows that $\|x(k)\| \to C < \infty$ as $k \to \infty$. □

REMARK 4.1
To prove the stronger result that the origin is asymptotically stable, the additional assumption that the system (4.1) has no transmission zeros at $q = 1$ from u to z could be imposed. Notice also that the sensible assumption that $Q > 0$ implies that $z(k) \to 0$ as $k \to \infty$, which is, however, automatically achieved if the transmission zero condition is fulfilled. □

The important feature in the proof of this theorem is embedded in equation (4.5). In order for the stability proof to work, it must be ensured that $V(k)$ is decreasing, which, however, does not require optimality of the control sequence $\Delta\mathcal{U}$. See, e.g., [Scokaert *et al.*, 1999] for a thorough discussion on this topic. Rather, having fulfilled the stability condition $V(k+1) < V(k)$, the optimization may be aborted prematurely without losing stability. In the case study in Section 4.5, the terminal constraint $\hat{u}(k+H_u)=0$ has been relaxed, in order to increase the feasibility region of the controller. To remove this complication, the control signal, u, rather than the control increments, Δu, could be included in the cost function. Notice, however, that the important feature of the stability proof that will be explored is the inequality (4.5) and that other, more sophisticated, stabilizing techniques may well be used instead.

QP-Solver

There are two major families of algorithms for solving LICQPs; *active set methods* [Fletcher, 1991] and *primal-dual interior point methods*, e.g., Mehrotra's predictor-corrector algorithm, [Wright, 1997]. Both types of methods have advantages and disadvantages when applied to MPC, as noted in [Bartlett *et al.*, 2000] and [Maciejowski, 2002]. Rather, the key to efficient algorithms lies in exploration of the structure of the optimization problem generated by the MPC algorithm.

Recent research has also suggested interesting, and fundamentally different MPC algorithms, see, e.g., [Kouvaritakis *et al.*, 2002] and [Bemporad *et al.*, 2002], known as explicit MPC. Here, the optimization problem is solved *off-line* for all $x(k)$, resulting in an explicit piecewise affine control law. At run-time, the problem is then transformed into finding the appropriate (linear) control law, based on the current state estimation. However, when the complexity of the problem increases, so does the complexity of the problem of finding the appropriate control law at each sample.

An MPC algorithm based on the on-line solution of a QP-problem is used. The value of the cost function at each iteration in the optimization algorithm is of importance. Specifically, if the decay of the cost function is slow, it may be a good choice to terminate the optimization algorithm, and use the sub-optimal solution, rather than allowing the algorithm to continue and thereby introduce additional delay in the control loop. In the scheduling case, long execution times will also affect the performance of other control loops.

From this point of view, there is a fundamental difference between an active set algorithm and a typical primal-dual interior point method. The active set algorithm explicitly strives to decrease the cost function in each iteration, whereas a primal-dual interior point algorithm rather tries to find, simultaneously, a point in the primal-dual space that fulfills the Karush-Kuhn-Tucker conditions. In the latter case, the duality gap is explicitly minimized in each iteration, rather than the cost function. With these arguments, and from our experience using both types of algorithms, we conclude that an active set algorithm is preferable for our application.

4.3 Termination Criterion

To be able to determine when to abort the MPC optimization and output the control signal, it is necessary to quantify the trade-off between the performance gain resulting from subsequent solutions of the QP-problem, and the performance loss resulting from the added computational delay. This will be achieved by the introduction of a delay-dependent cost index, which is based on a parameterization of the cost function (4.2).

Assuming a constant time delay, $\tau < h$, the process model (4.1) can be augmented (see, e.g., [Åström and Wittenmark, 1997]) as

$$\begin{aligned} \tilde{x}(k+1) &= \tilde{\Phi}\tilde{x}(k) + \tilde{\Gamma}u(k) \\ y(k) &= \tilde{C}_y\tilde{x}(k) \\ z(k) &= \tilde{C}_z\tilde{x}(k) + D_z u(k) \end{aligned} \tag{4.6}$$

where

$$\begin{aligned} \tilde{x}(k) &= \begin{pmatrix} x(k) & u(k-1) \end{pmatrix}^T \\ \tilde{\Phi} &= \begin{pmatrix} \Phi & \Gamma_1(\tau) \\ 0 & 0 \end{pmatrix}, \quad \tilde{\Gamma} = \begin{pmatrix} \Gamma_0(\tau) \\ 1 \end{pmatrix} \\ \tilde{C}_y &= \begin{pmatrix} C_y & 0 \end{pmatrix}, \qquad \tilde{C}_z = \begin{pmatrix} C_z & 0 \end{pmatrix} \\ \Gamma_0(\tau) &= \int_0^{h-\tau} e^{As} ds \; B \\ \Gamma_1(\tau) &= e^{A(h-\tau)} \int_0^{\tau} e^{As} ds \; B \end{aligned}$$

where A and B are the continuous system matrices of the plant. The matrices $\mathcal{H}$, $\mathcal{G}$, $\mathcal{C}$, Ω, and ω in (4.4) all depend on the system matrices and thus on the delay. Ideally, these matrices should be updated from sample to sample based on the current computational delay.

However, using the representation (4.6) it is possible to evaluate the cost function (4.2) assuming a constant computational delay, τ, over the prediction horizon. The assumption that the delay is constant over the prediction horizon is in line with the assumptions commonly made in the standard MPC formulation, e.g., that the current reference values will be constant over the prediction horizon. Thus, for each iterate,

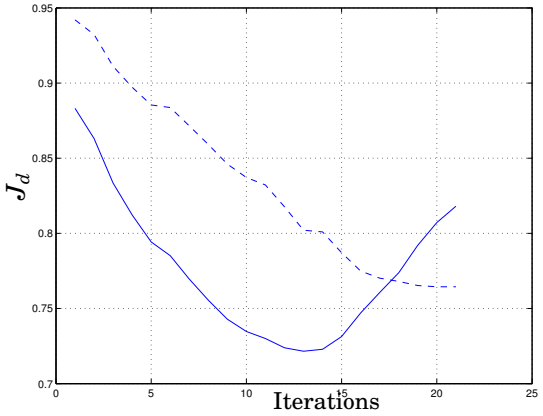

Figure 4.2 The solid curve shows the delay-dependent cost index J_d, and the dashed curve shows the original cost function used in the QP-algorithm.

$\Delta\mathcal{U}_i$, produced by the optimization algorithm, we compute

$$J_d(\Delta\mathcal{U}_i, \tau) = \Delta\mathcal{U}_i^T \mathcal{H}(\tau)\Delta\mathcal{U}_i - \Delta\mathcal{U}_i^T \mathcal{G}(\tau) + \mathcal{C}(\tau) \tag{4.7}$$

This cost index penalizes not only deviations from the desired reference trajectory, but also performance degradation due to computational delay. There are two major factors that affect the evolution of J_d. On one hand, an increasing τ, corresponding to an increased computational delay, may degrade control performance and cause J_d to increase. On the other hand, J_d will decrease for successive $\Delta\mathcal{U}_i$:s since the quality of the control signal has improved. Figure 4.2 shows the evolution of J_d during an optimization run. In the beginning of the optimization, J_d is decreasing rapidly, but then increases due to computational delay. In this particular example, the delayed control trajectory seems to achieve a lower cost than the original. This situation may occur since the cost functions are evaluated for non-optimal control sequences, except for the last iteration. Notice, however, that for the optimal solution, J_d is higher than the original cost. The proposed termination strategy is then to compare the value of $J_d(\Delta\mathcal{U}_i, \tau_i)$ with the cost index computed after the previous iteration, i.e., $J_d(\Delta\mathcal{U}_{i-1}, \tau_{i-1})$, where

τ_i denotes the current computational delay after the ith iteration. If the cost index has decreased since the last iteration, we conclude that we gained more by optimization than we lost by the additional delay. On the other hand, if the cost index has increased, the optimization is aborted. Notice that the matrices needed to evaluate J_d should be calculated off-line.

In the MPC formulation we are assuming a process model without delay. Another possible approach would be to include a fixed-sample delay in the process description. However, since the computational delay is highly varying, compensating for the maximum delay may become very pessimistic and lead to decreased obtainable performance. We will also assume that the control signal is actuated as soon as the optimization algorithm terminates, not to induce any unnecessary delay.

4.4 Dynamic Real-Time Scheduling of MPCs

The cost index described above, will now be applied in a dynamic real-time scheduling context. The basic ideas of the dynamic scheduling scheme were given in Section 3.3.

MPC tasks do not fit the traditional task model very well, mainly because of their highly varying execution times. On the other hand, MPC offers two features that distinguish it from ordinary control algorithms from a real-time scheduling perspective. First, as we have seen in the previous sections, it is possible to abort the computation and thereby reduce the execution time. Second, the cost index contains relevant information about the state of the controlled process. Thus, the cost index can be viewed as a real-world quality-of-service measure for the controller, and be used as a dynamic task priority by the scheduler. This also enables a tight and natural connection between the control and the real-time scheduling.

The MPC algorithm can be divided into two parts. The first (mandatory) part consists of finding a starting point fulfilling the constraints in the MPC formulation (constraints on the controlled and control variables and the terminal equality constraint) and to iterate the QP optimization algorithm until the stability condition of Theorem 1 is fulfilled. The second (optional) part consists of the additional QP-iterations that further reduce the value of the cost function.

Based on this insight, the MPC algorithm can be cast into the framework of scheduling of imprecise computations presented in Chapter 3. The mandatory sub-tasks will be given the highest priority, whereas the optional sub-tasks will be scheduled based on the values of the MPC cost indices. Listing 4.1 contains pseudo code of a dynamic scheduling scheme of the optional sub-tasks. The strategy also exploits the trade-off between optimization and computational delay.

It should be noted that comparing cost indices directly may not be appropriate when the controllers have different sampling intervals, prediction horizons, weighting matrices, etc. In those cases, it would be necessary to scale the cost indices to obtain a fair comparison. The scheduling could also use feedback from the derivatives of the cost functions, as well as the relative deadlines of the different controllers.

Listing 4.1 Dynamic scheduling strategy for MPC tasks.

```
determine MPC sub-task i with highest J_d;
schedule sub-task i for one iteration;

now = currentTime;
if (optimum_reached_i) {
   actuate plant_i;
} else {
   delay_i = now - start_i;

   if (J_d(u_i,delay_i) > prev J_d) {
      abort optimization;
      actuate plant_i;
   }
}
```

4.5 Case Study

The proposed termination criterion and dynamic real-time scheduling strategy have been evaluated in simulation using a second order system, a double-integrator:

$$\begin{aligned} \dot{x} &= \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 1 \end{pmatrix} u \\ y &= \begin{pmatrix} 1 & 0 \end{pmatrix} x \end{aligned} \tag{4.8}$$

The plant was discretized using the sampling interval $h = 0.1$ s. In the simulations, $z = x_1$ was set to be the controlled state and the constraints $|u| \leq 0.3$ and $|x_2| \leq 0.1$ were enforced.

The MPC controller used in the simulations was implemented as described in Section 4.2, with prediction horizons $H_p = 50$ and $H_u = 20$ and weighting matrices $Q = 1$ and $R = 0.1$.

Simulation Environment and Implementation

Real-time MPC control of the double-integrator process was simulated using the TrueTime toolbox (see Chapter 5). Using TrueTime it is possible to perform detailed co-simulation of the MPC control task executing in a real-time kernel and the continuous dynamics of the controlled process. Using the toolbox it is easy to simulate different implementation and scheduling strategies and evaluate them from a control performance perspective.

In the standard implementation, the MPC task is released periodically and new instances may not start to execute until the previous instance has completed. This implementation will allow for task overruns without aborting the ongoing computations. The control signal is actuated as soon as the task has completed.

In the dynamic scheduling scheme, the MPC task is divided into a mandatory and an optional part as described in Section 4.4. The mandatory part is scheduled with a distinct high priority, whereas the priority of the optional part is changed depending on the current value of the cost index compared to other running MPC tasks.

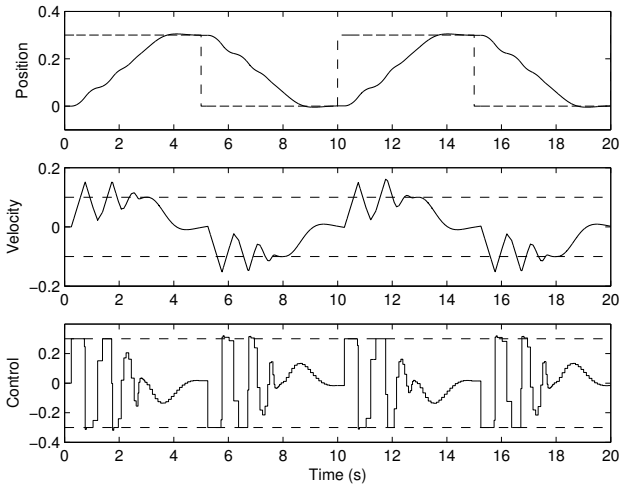

Figure 4.3 Control performance when the optimization algorithm is allowed to finish in every sample. The bad performance is a result of considerable delay and jitter induced by the large variations in execution time. During the transients the long execution times cause the control task to miss its next invocation, inducing sampling jitter. The dashed lines in the velocity and control signal plots show the constraints used in the MPC formulation.

Simulation of One MPC Controller

The first simulations consider the case of a single MPC task implemented according to the standard task model described in the previous section. Figure 4.3 shows the result of a simulation where the optimization is allowed to finish in each sample. Delay and jitter induced by the large variations in execution time compromise the optimal control performance. The constraints are shown by the dashed lines in the velocity and control signal plots. As seen in the plots the constraints are violated at some points. This is due to the computational delay, which is not accounted for in the MPC formulation.

Figure 4.4 shows a simulation utilizing the termination criterion of Section 4.3. The cost index (4.7) is evaluated after each iteration, and if it has increased since the last iteration, the optimization is aborted

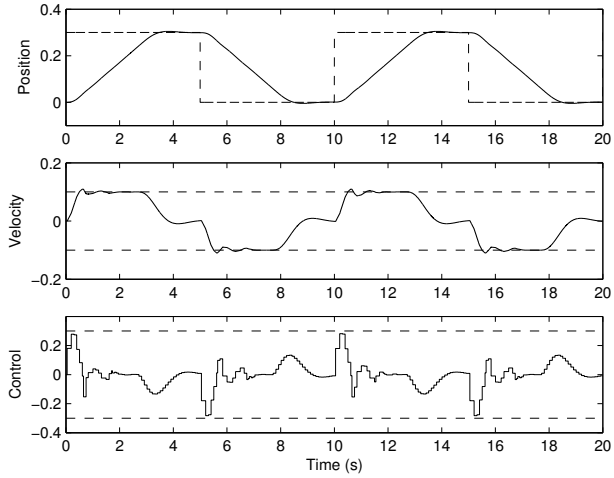

Figure 4.4 Control performance obtained using the proposed sub-optimal approach where the QP-optimization may be aborted according to the termination criterion described in Section 4.3. The performance is increased substantially compared to Figure 4.3.

and the current control signal is actuated. As can be seen from the simulations, the control performance has increased significantly.

Figure 4.5 shows a comparison of the number of iterations needed for full optimization (top) and the number of iterations after which the optimization was aborted due to an increasing value of J_d (bottom). The execution time of each iteration in the simulation was 10 ms. Average values for computation times and the number of iterations in the QP optimization algorithm in each sample is summarized in Table 4.1. The number of necessary iterations denotes the number of QP-iterations needed to fulfill the stability condition. It can be seen that the total execution time of the MPC task is reduced by 35 percent by using the proposed termination criterion. The execution time for the mandatory part of the algorithm is roughly constant for both approaches. In the full optimization case, the execution time will exceed the 100 ms sampling period during the transients, causing the control task to miss deadlines and experience sampling jitter.

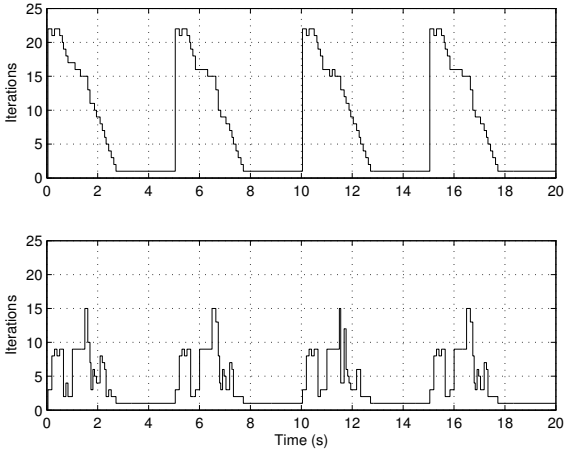

Figure 4.5 Number of iterations for the QP-solver. The top plot shows the number of iterations to find the optimum. The bottom plot shows the number of iterations after which the optimization is terminated and the sub-optimal control is actuated.

Table 4.1 Average timing values per sample for a simulation.

Optimization	Full	Sub-optimal
Total time [s]	0.1055	0.0692
Mandatory time [s]	0.0302	0.0297
Number of iterations	8.87	5.66
Number of necessary iterations	1.70	1.89

To quantify the simulation results, the *performance loss*

$$J = \int_0^{T_{sim}} \left(\|z(t) - r(t)\|_Q^2 + \|\Delta u(t)\|_R^2 \right) dt \tag{4.9}$$

was recorded in both cases. The weighting matrices, Q and R, were the same as those used in the MPC formulation. The performance loss was scaled with the loss for an ideal simulation. The ideal case was

Table 4.2 Performance loss comparison in the single MPC case.

Strategy	Loss
Ideal case	1.0
Full optimization	1.35
Sub-optimal	1.09

obtained by simulating full optimization and zero execution time in each sample. The results are given in Table 4.2.

Dynamic Scheduling of Two MPC Tasks

In the following simulations the dynamic scheduling strategy proposed in Section 4.4 will be compared to ordinary fixed-priority scheduling. Two MPC controllers are implemented and executed by two different tasks running concurrently on the same CPU controlling two different double-integrator processes. Both MPC controllers are designed with the same prediction and control horizons, sampling periods, and weighting matrices in the MPC formulation.

Both controllers were given square-wave reference trajectories, but with different amplitudes and periods. The reference trajectory for MPC1 had an amplitude of 0.3 and a period of 10 s. The corresponding values for MPC2 were 0.4 and 12 s. The different reference trajectories will cause the relative computational demands of the MPC tasks to vary over time. Therefore, it is not obvious which controller task to give the highest priority. Rather, this should be decided on-line based on the current state of the controlled process.

The simulation results are shown in Figures 4.6-4.8. The first two simulations show the fixed-priority cases. MPC1 is given the highest priority in the first simulation, and MPC2 is given the highest priority in the second simulation. It is seen that we get different control performance, depending on how we choose the priorities. By giving MPC2 the highest priority, the performance in this particular simulation scenario is considerably better than if the priorities are reversed.

The performance using dynamic scheduling based on the cost index (4.7) is shown in Figure 4.8, and the performance is improved signifi-

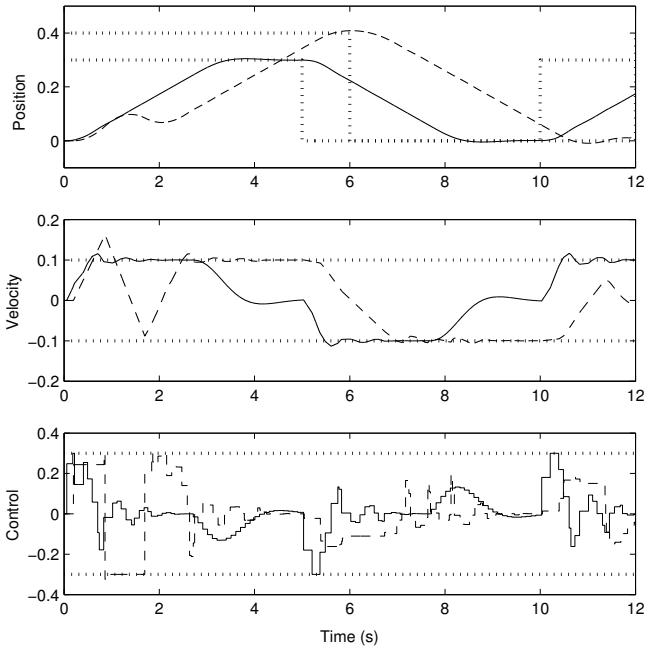

Figure 4.6 Control performance using fixed-priority scheduling where MPC1 (solid) is given the highest priority. MPC2 (dashed) is constantly preempted by the higher priority task, consequently degrading its performance.

cantly. Figure 4.9 shows a close-up of the computer schedule during one sample. After both tasks have completed the mandatory parts of their algorithms, the execution trace (the dynamic priority assignments) is determined based on the values of the cost functions of the individual tasks. These values after each iteration are shown in the figure. The termination criterion aborts both tasks at time 0.08.

The scaled performance loss (4.9) was recorded for the individual control loops and added up to obtain a total loss for each of the different scheduling strategies. The results are summarized in Table 4.6. It can be seen that the improvement using the dynamic scheduling is less significant in the case where MPC1 is given the highest priority. This

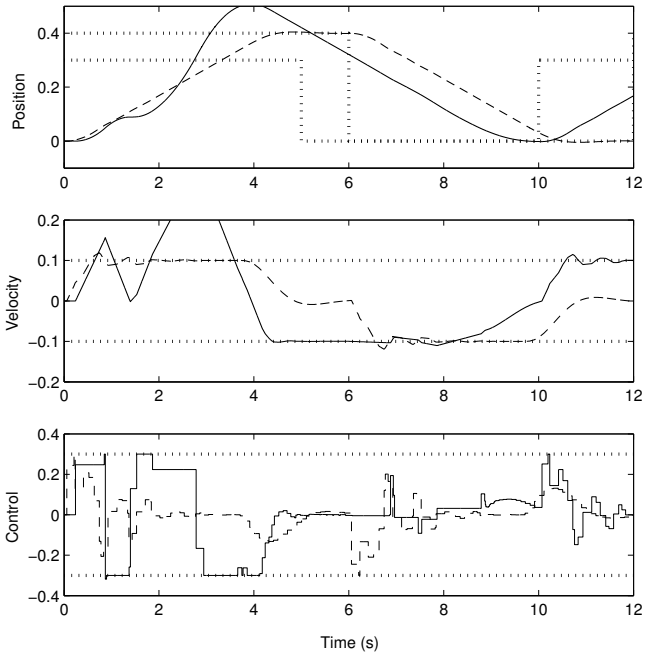

Figure 4.7 Control performance using fixed-priority scheduling where MPC2 (dashed) is given the highest priority. Comparing with Figure 4.6 it can be seen that the performance is worse using this priority assignment.

is, however, due to the particular reference trajectories applied in this simulation.

Using the proposed dynamic scheduling strategy we arbitrate the computing resources according to the current situation for the controlled processes, and the varying computational demands caused by reference changes and other external signals are taken into account at run-time. It should be noted that the control performance obtained using the dynamic cost-based scheduling would have been the same if the reference trajectories for the two controllers had been switched. As we have shown this would not have been the case using ordinary fixed-priority scheduling.

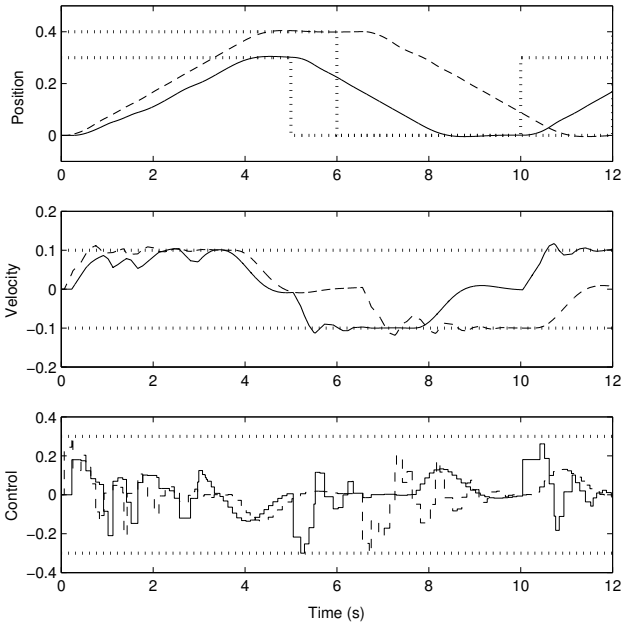

Figure 4.8 Control performance using the dynamic scheduling approach. Scheduling based on cost functions makes sure that the most urgent task gets access to the processor, thus increasing the overall performance.

4.6 Summary

This chapter has presented a flexible implementation approach for model predictive controllers (MPCs). Premature termination of the optimization algorithm was exploited to improve control performance. The resulting stabilizing MPC control sequences were sub-optimal from an optimization point-of-view, but optimal from a control performance perspective when taking the computational delay into account.

A delay-dependent cost index was presented to quantify the trade-off between improving control signal quality resulting from successive iterations in the optimization algorithm and potential control performance degradation due to computational delay. The cost index provided

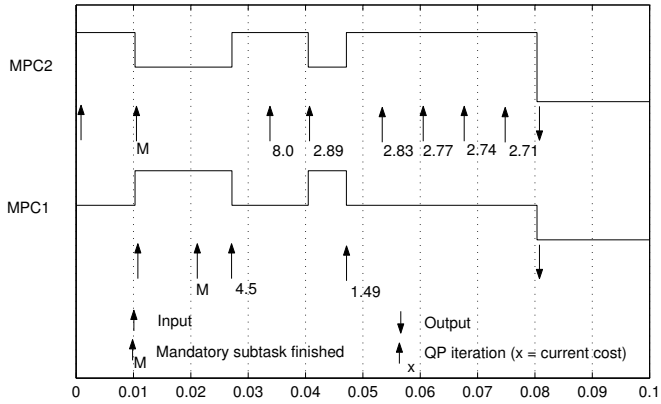

Figure 4.9 Computer schedule in a sample using the dynamic scheduling approach (high = running, medium = preempted, low = idle). The figure shows the completion of the mandatory part, as well as the value of the cost index after each iteration of the QP-solver.

Table 4.3 Performance loss for the different scheduling strategies.

Strategy	Loss
Ideal case	2.0
Fixed priority / MPC1 highest priority	2.47
Fixed priority / MPC2 highest priority	2.79
Dynamic cost-based scheduling	2.43

guidance for when to terminate the optimization algorithm, while preserving the stability properties of the MPC algorithm.

It has also been shown how the cost index can be used in the context of dynamic real-time scheduling. The cost index has been used to provide the scheduling algorithm with information to be used for deciding which of two MPC controllers should be allocated execution time. Using the index for scheduling, it has been shown how the overall control performance may be significantly improved compared to traditional fixed-priority scheduling.

5

The TrueTime Simulator

5.1 Introduction

To achieve good performance of control systems subject to limited computer resources, the constraints of the implementation platform should be taken into account at design time. The true effects of timing nondeterminism, e.g., delay and jitter, on control performance are, however, often very hard to investigate analytically. A natural approach is then to instead use simulation. However, today's simulation tools make it difficult to simulate the true temporal behavior of control loops. What is normally done in, e.g., Simulink, is to introduce time delays in the control loops representing average- or worst-case delays.

A more detailed simulation can be performed using TrueTime, which is a MATLAB/Simulink-based toolbox facilitating 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., the priority-based preemptive scheduling and earliest-deadline-first (EDF) scheduling described in Chapter 2). A comparison between a TrueTime simulation model and a traditional simulation model of a distributed control system is shown in Figure 5.1.

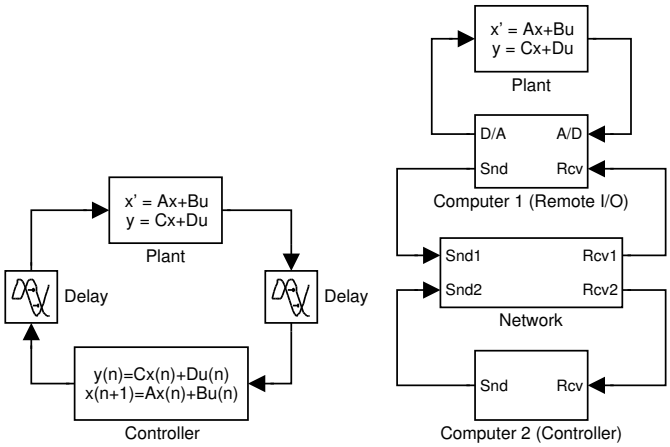

Figure 5.1 Left: Traditional simulation model of a distributed control system. Computers and network are modeled as simple delays. Right: TrueTime model where the execution of tasks and the transmission of messages are simulated in parallel with the plant dynamics.

TrueTime can be used in several ways, e.g., to study compensation schemes that adjust the control algorithm based on measurements of actual timing variations (i.e., to treat the temporal uncertainty as a disturbance and manage it with feed-forward or gain scheduling). It is also easy to experiment with flexible approaches to real-time scheduling of controllers, such as, e.g., the feedback scheduling approaches described in Chapter 3.

This chapter contains a general overview of TrueTime and a detailed description of the kernel implementation and event-based simulation using Simulink. For a detailed description of how to use the simulator, see [Henriksson and Cervin, 2003].

The TrueTime development has been ongoing since 1998, and an early version of the simulator was presented in [Eker and Cervin, 1999]. Modifications and extensions to the simulator are being made regularly and are posted at the TrueTime web page[1].

[1]TrueTime is available for download at http://www.control.lth.se/~dan/truetime

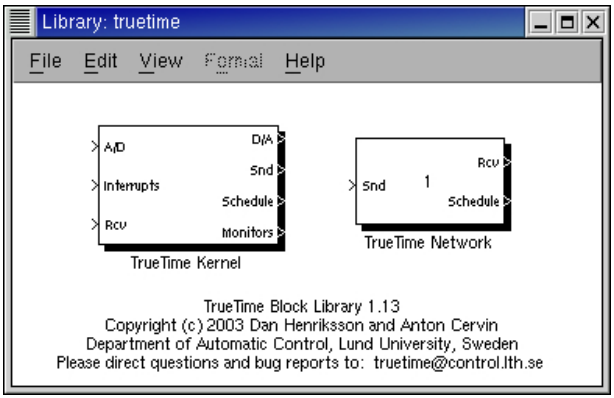

Figure 5.2 The TrueTime block library. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

5.2 Simulator Overview

TrueTime consists of a block library with a computer kernel block and a network block, as shown in Figure 5.2. The kernel block executes user-defined tasks and interrupt handlers representing, e.g., I/O tasks, control algorithms, and network interfaces. The scheduling policies of the individual kernel blocks are arbitrary and decided by the user. The network block distributes messages between computer nodes according to a chosen network model.

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

The block inputs are assumed to be discrete-time signals, except the signals connected to the A/D converters of the kernel 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.

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 crank angle passed zero degrees.”

The blocks are implemented as variable-step, MATLAB 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).

5.3 The Kernel Block

The kernel block S-function 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. These 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 abstraction in the TrueTime simulation environment. An arbitrary number of tasks can be created to run in the TrueTime kernel. Tasks may also be created dynamically as the simulation progresses. 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. Aperiodic tasks are executed by the creation of task instances (jobs). All pending jobs are inserted in a job queue which is sorted by release time. For periodic task, an internal timer is set up to periodically create jobs for the task.

Each task is characterized by a number of static and dynamic 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 from the application code. The worst-case execution time is only used if an execution-time overrun handler is attached to the task (see below). Also note that the worst-case execution time does not determine the actual execution time of the task. Rather, this is specified by the user in the code function of the task.

In addition to these attributes, each task instance has a number of dynamic attributes associated with it. These attributes are updated by the kernel as the simulation progresses, and 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 from the user code during simulation. Depending on the scheduling policy, changing an attribute may lead to a context switch. E.g., under EDF scheduling, changing the absolute deadline of a task will result in a re-sorting of the ready queue.

In accordance with the Real-Time Specification for Java (RTSJ) [Bollella *et al.*, 2000], it is furthermore possible to attach two overrun handlers to 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).

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 kernel block. The interrupt is triggered when the signal of the corresponding channel changes value. This type of interrupt may be used to simulate engine controllers that are sampled against the rotation of the motor or distributed controllers that execute when measurements arrive on 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 the overrun handlers that may be associated with each task.

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 non-preemptive; 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 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 the absolute deadline implies earliest-deadline-first scheduling. Predefined priority functions exist for rate-monotonic, deadline-monotonic, fixed-priority, and earliest-deadline-first scheduling.

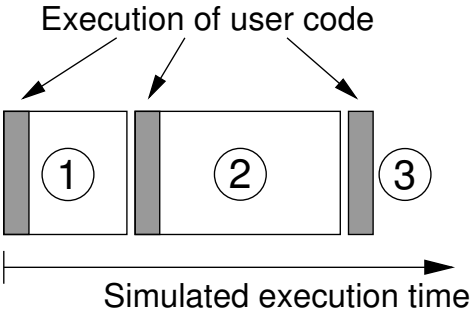

Figure 5.3 The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

Code

The code associated with tasks and interrupt handlers is scheduled and executed by the kernel as the simulation progresses. The code is normally divided into several segments, as shown in Figure 5.3. 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 number of segments can be chosen to simulate an arbitrary time granularity of the code execution. Technically it would, e.g., be possible to simulate very fine-grained details occurring at the machine instruction level, such as race conditions. However, that would require a large number of code segments.

The simulated execution time of each segment is returned by the code function, and can 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 by higher-priority activities and interrupts may cause the actual delay between execution of segments to be longer than the execution time.

Listing 5.1 Example of a standard code function written in MATLAB code. The local memory of the controller task is represented by the data structure `data`. This stores the controller gain and the control signal between invocations of different code segments.

```
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
```

Listing 5.1 shows an example of a code function corresponding to the time line in Figure 5.3. The same example implemented as a C function is shown in Listing 5.2. The function implements a standard controller with a calculate part and an update part. In the first segment, the plant is sampled and the control signal is computed. In the second segment, the control signal is actuated and the controller states are updated. The third segment indicates the end of execution in this sample by returning a negative execution time.

The data structure `data` represents the local memory of the task and is used to store the control signal and measured variable between calls to the different segments. A/D and D/A conversion is performed using the kernel primitives `ttAnalogIn` and `ttAnalogOut`.

Note that the input-output latency of this controller will be *at least* 2 ms (i.e., the execution time of the first segment). However, if there is preemption from other high-priority tasks, the actual input-output latency will be longer.

Listing 5.2 The C implementation of the code example in Listing 5.1.

```
double myController(int segment, void* data) {

  Ctrl_Data* d = (Ctrl_Data*) data;

  switch (segment) {
  case 1:
    d->y = ttAnalogIn(1);
    calculateOutput(d);
    return 0.002;
  case 2:
    ttAnalogOut(1, d->u);
    updateState(d);
    return 0.003;
  case 3:
    return FINISHED; // end of execution
  }
}
```

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 built-in function `ttCallBlockSystem`.

A block diagram of an ordinary PI-controller is shown in Figure 5.4. The block system has two inputs, the reference signal and the process output, and two outputs, the control signal and the execution time. The use in a code function is given by Listing 5.3.

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

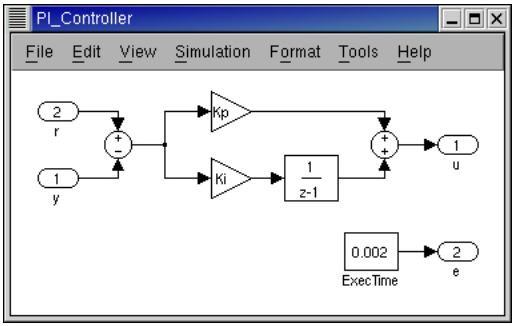

Figure 5.4 Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions. The example above shows a PI controller.

tasks where no conditions on shared data are involved. The example in Listing 5.4 shows the use of a free event `input_event` to simulate an event-driven controller task. The corresponding `ttNotifyAll`-call on the event is typically performed in an interrupt handler associated

Listing 5.3 Example of a code function calling the PI-controller block diagram in Figure 5.4 to compute the control signal.

```
function [exectime,data] = PIController(segment,data)
switch segment,
  case 1,
    inp(1) = ttAnalogIn(1);
    inp(2) = ttAnalogIn(2);
    outp = ttCallBlockSystem(inp, 'PI_Controller');
    data.u = outp(1);
    exectime = outp(2);
  case 2,
    ttAnalogOut(1,data.u);
    exectime = 0.003;
  case 3,
    exectime = -1; % finished
end
```

with an external interrupt port. (An alternative implementation of an event-based task, using the kernel primitive `ttCreateJob`, will be given in Listing 5.5.)

Output Graphs

Depending on the simulation, several different output graphs are generated by the TrueTime blocks. Each kernel 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 overhead is simulated, the graph will also display the execution of the kernel.

For an example of such an execution trace, see Figure 5.7. If the signal is high it means that the task is running. A medium signal indicates that the task is ready but not running (preempted), whereas a low signal means that the task is idle. In an analogous way, the network schedule shows the transmission of messages over the net-

Listing 5.4 Example of a code function implementing an event-based controller.

```
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 back
end
```

work, with the states representing sending (high), waiting (medium), and idle (low). The monitor graph shows which 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.

5.4 The Network Block

The network block is event-driven and executes when messages enter or leave the network. When a node tries to transmit a message, a triggering signal is sent to the network block on the corresponding input channel. When the simulated transmission of the message is finished, the network block sends a new triggering signal on the outport channel corresponding to the receiving node. The transmitted message is put in a buffer at the receiving computer node.

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.

The network block simulates medium access and packet transmission in a local area network. Six simple models of networks are currently supported: CSMA/CD (e.g. Ethernet), CSMA/AMP (e.g. CAN), Round Robin (e.g. Token Bus), FDMA, TDMA (e.g. TTP), and Switched Ethernet. The propagation delay is ignored, since it is typically very small in a local area network. Only packet-level simulation is supported, i.e., it is assumed that higher protocol levels in the kernel nodes have divided long messages into packets.

Configuring the network block involves specifying a number of general parameters, such as transmission rate, network model, and probability for packet loss. Protocol-specific parameters that need to be supplied include, e.g., the time slot and cyclic schedule in the case of TDMA. For an example of how to configure the individual TrueTime nodes for network communication, see Listing 5.5.

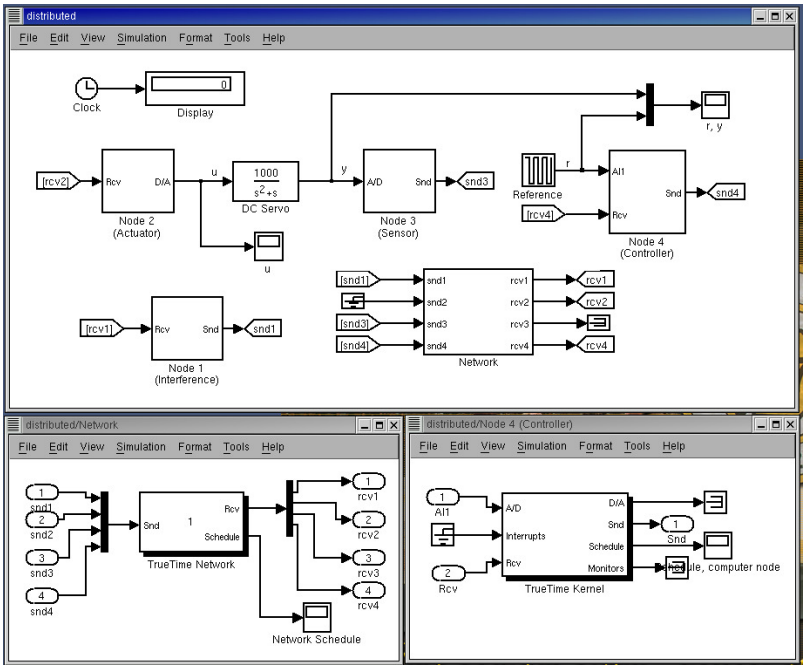

Figure 5.5 TrueTime simulation model of the networked control system.

5.5 Example: A Networked Control System

As an example of a simulation in TrueTime, we consider a general simulation of a distributed control system, wherein the effects of scheduling in the CPUs and simultaneous transmission of messages over the network can be studied in detail. The TrueTime model of the system is shown in Figure 5.5.

The model contains four computer nodes connected by one network block. The time-driven *sensor node* contains a periodic task, which periodically samples the process and transmits the sample package to the *controller node*. 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. The model also contains an *interference node* with a periodic task generating random interfering traffic over the network.

Initialization of the Actuator Node

As a complete initialization example, Listing 5.5 shows the code needed to initialize the actuator node in this particular example. The kernel block contains one task and one interrupt handler, and their execution is defined by the code functions `actCode` and `rcvCode`, respectively. The task and 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 (`'rcv_hdl'`) 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 simulated time for a full context switch (zero in this case). The built-in priority function `prioFP` specifies fixed-priority scheduling.

Simulations

In the following simulations, we will assume a CAN-type network where transmission of simultaneous messages is decided based on package priorities. The controller node contains a PD-controller task designed for a 10 ms sampling interval. The sampling interval is enforced by the time-driven task in the sensor node sending samples periodically to the controller node.

The execution time of the controller is 0.5 ms and the ideal transmission time from one node to another is 1.5 ms. The ideal round-trip delay 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 round-trip delay to be even longer on average and time-varying. The resulting degraded control performance can be seen in the simulated step response in the top plots of Figure 5.6. The execution of the tasks in the controller node and the transmission of messages over the network can be studied in detail in Figure 5.7.

Listing 5.5 Complete code for the actuator node in the networked 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; % read from network input buffer
    exectime = 0.0005;
  case 2,
    ttAnalogOut(1, data.u);
    exectime = -1;
end

%% Code function for the network interrupt handler
function [exectime,data] = rcvCode(segment,data)
ttCreateJob(ttCurrentTime, 'act_task');
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('rcv_hdl', 1, 'rcvCode');
ttInitNetwork(2, 'rcv_hdl'); % node number 2 in the network
```

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

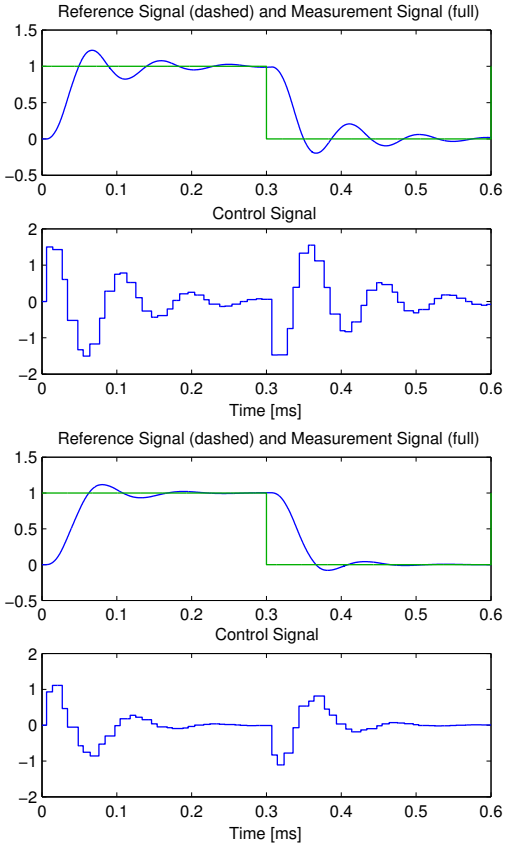

Figure 5.6 The top plot shows the degraded control performance resulting from interfering network messages and an interfering task in the controller node. The bottom plot shows the improved performance resulting from the implementation of delay-compensation in the controller node.

value of the delay from controller to actuator. The control signal is then calculated based on linear interpolation among a set of controller parameters pre-calculated for different delays. Using this compensation, better control performance is obtained, as seen in the bottom plots of Figure 5.6.

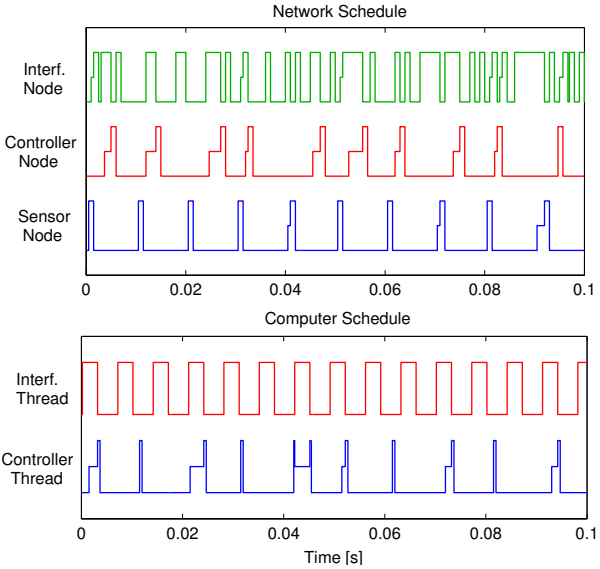

Figure 5.7 Close-up of schedules showing the allocation of common resources: network (top) and controller node (bottom). A high signal means sending or executing, a medium signal means waiting, and a low signal means idle.

5.6 Kernel Implementation Details

This section will give a brief description of the implementation of the TrueTime kernel. The main data structures will be described as well as the kernel implementation. It will also be shown how to achieve event-based simulation in Simulink, using the zero-crossing detection mechanism.

Kernel Data Structures

The main data structure of the TrueTime kernel is a C++ class called `RTsys`. An instance (`rtsys`) of this class is created in the initialization step of the kernel S-function. The `rtsys` object is stored in the `UserData` field of the kernel block between simulation steps. Among others, the `RTsys` class contains the following attributes:

```
class RTsys {
 public:
  double time;              // Current time in simulation

  double* inputs;           // Vector of input port values
  double* outputs;          // Vector of output port values

  Task* running;            // Currently running task

  List* readyQ;   // Contains tasks ready for execution
  List* timeQ;    // Contains tasks and timers

  List* taskList;    // A list containing all created tasks
  List* handlerList;
  List* monitorList;
  List* eventList;

  double (*prioFcn)(Task*); // Priority function
};
```

The ready queue and time queue are sorted linked list. The elements in the time queue (tasks and timers) are sorted according to release times and expiry times. The tasks in the ready queue are sorted according to the priority function `prioFcn`, which is a function that returns a (possibly dynamic) priority number from a `Task` instance.

The `Task` class contains the following basic attributes:

```
class Task {
 public:
  char* name;
  double wcExecTime;
  double deadline;
  double assignedPriority;
  double priority; // dynamic priority (priority inheritance)
  void *data;

  double (*codeFcn)(int, void*); // Code function written in C++
  char* codeFcnMATLAB;  // Name of m-file code function

  Handler* deadlineORhandler; // deadline overrun handler
  Handler* exectimeORhandler; // execution-time overrun handler
```

```
  Job* currentJob;  // Job currently served
  List* jobQ;       // List of pending jobs
};
```

The kernel implements priority inheritance to avoid priority inversion. Therefore each task has a dynamic priority value that may be raised while executing inside a monitor. The code function of the task is represented either as a function pointer in the C++ case or the name of a MATLAB m-file. The currently running job is given by the pointer `currentJob`. Pending jobs are stored in the job queue of the task.

The `Job` class contains the following basic attributes:

```
class Job {
 public:
  double execTime;  // remaining execution time of current segment
  double lastStart; // last time the job was resumed
  double absDeadline; // absolute deadline of job
  double release;   // release time of job
  double budget;    // remaining execution time budget
  int segment;      // current segment of the code function
};
```

All dynamic attributes of a task are contained in the `Job` class. The variable `lastStart` is used to store successive resume times of the job. This is used to set up the timers that are used to implement the execution-time overrun handling (see scheduling hooks below).

The `Handler` class contains the following basic attributes:

```
class Handler {
 public:
  char* name;
  double execTime;
  double priority;
  int segment;
  void *data;

  double (*codeFcn)(int, void*); // Code function written in C++
  char* codeFcnMATLAB;  // Name of m-file code function
};
```

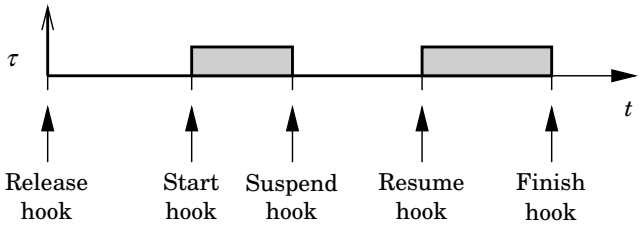

Figure 5.8 The various scheduling hooks that can be used to attach arbitrary functionality to the scheduling algorithm.

Writing a Priority Function

Based on the above data structures it is straight-forward to write priority functions implementing arbitrary scheduling schemes. E.g., the priority function implementing standard EDF scheduling is given as

```
double prioEDF(Task* task) {

  return task->currentJob->absDeadline;
}
```

Scheduling Hooks

To facilitate arbitrary dynamic scheduling mechanisms, it is possible to attach small pieces of user-defined code (*scheduling hooks*) to each task. These hooks are executed at different stages during the simulation of the task, as shown in Figure 5.8. The hooks can, e.g., be used to monitor different scheduling schemes and keep track of context switches and deadline overruns. By default, the hooks contain code to trigger the worst-case execution time and deadline overrun handlers possibly associated with the different tasks. This is summarized below.

- *Release hook:* If the released task has an associated deadline overrun handler, a timer is created. The expiry of this timer is set to the absolute deadline of the task. It may be the case that, because of previous overruns, the absolute deadline of the task has already expired when the task instance is released. In this case the overrun handler is activated immediately.
- *Start hook:* If the task has an associated execution-time overrun handler, another timer is created. The expiry of the timer is set

to the current time plus the remaining execution-time budget. The start time of the task is recorded.

- *Suspend hook:* The execution-time budget is decreased based on the time elapsed since the task last began execution. 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.

The Kernel Function

The TrueTime real-time kernel is implemented in a function `runKernel` that is called by the Simulink S-function call-back procedures at appropriate times during the simulation. See next section for timing implementation details.

This function manipulates the basic data structures of the kernel, such as the ready queue and the time queue. It is also from this function the code functions for tasks and interrupt handlers are called. The kernel keeps track of the current segment and updates it when the time associated with the previous segment has elapsed. The hooks mentioned above are also called from this function.

A simple model for how the kernel works is given by the pseudo code in Listing 5.6. Note that interrupt handlers are not treated in the pseudo code. However, they are handled essentially in the same way as the tasks.

Simulink Timing Details

The TrueTime blocks are event-driven and support external interrupt handling. Therefore, the blocks have a continuous sample time, and the timing of the block is implemented using the Simulink zero-crossing functionality [The Mathworks, 2001b].

Listing 5.6 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
        // (sorted after release and expiry)
        for (each task) {
            if (release time - currentTime == 0.0) {
                Move the task from time queue to the ready queue;
                Execute release-hook;
            }
```

Listing 5.6 (Continued)

```
        }
        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;
}
```

As seen in Listing 5.6, the next time the kernel should wake up (e.g., because a task is to be released from the time queue or a task has finished its execution) is denoted `nextHit`. If there is no known wake-up time, this variable is set to infinity. The basic structure of the zero-crossing function is

```
static void mdlZeroCrossings(SimStruct *S) {

   Store all inputs;
   if (any external interrupt input has changed value) {
      nextHit = ssGetT(S);
   }
   ssGetNonsampledZCs(S)[0] = nextHit - ssGetT(S);
}
```

This will ensure that the Simulink call-back function `mdlOutputs` executes every time an internal or external event has occurred. The kernel function (`runKernel`) is only called from `mdlOutputs` since this is where the outputs (D/A, schedule, network) can be changed.

Since several kernel and network blocks may be connected in a circular fashion, *direct feedthrough* is not allowed. We exploit the fact that, when an input changes as a step, `mdlOutputs` is called, followed by `mdlZeroCrossings`. Since direct feedthrough is not allowed, the inputs may only be checked for changes in `mdlZeroCrossings`. There, the zero-crossing function is changed so that the next major step occurs at the current time.

Commands and Real-Time Primitives

To give an overview of the functionality of TrueTime, a summary of the available functions and commands is given in Table 5.1. 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.

Based on the kernel data structures it is easy to extend the TrueTime functionality with new functions. As examples, follows below the TrueTime implementation of the standard real-time primitives, `sleepUntil`, `sleep`, and `setPriority`.

```
void ttSleepUntil(double time) {

  // Set new release time
  rtsys->running->currentJob->release = time;
  // Insert into time queue
  rtsys->timeQ->insertSorted(new TaskNode(rtsys->running));
  // Remove from ready queue (running is first in queue)
  rtsys->readyQ->deleteFirst();

  rtsys->running = NULL;
}

void ttSleep(double duration) {
  ttSleepUntil(rtsys->time + duration);
}

void ttSetPriority(double prio) {

  Task* task = rtsys->running;
  if (task->priority - task->assignedPriority < 0.0) {
    // Priority inheritance, do not change the priority now
  } else {
    task->priority = prio;
  }
  task->assignedPriority = prio;
  // Possible reordering of readyQ
  TaskNode* tn = rtsys->readyQ->getFirst();
  rtsys->readyQ->removeNode(tn);
  rtsys->readyQ->insertSorted(tn);
}
```

5.7 Summary

This chapter has described TrueTime, a MATLAB/Simulink toolbox that facilitates co-simulation of continuous plant dynamics, controller task execution in real-time kernels, and network transmissions. Arbitrary scheduling policies and various network protocols may be evaluated from a control performance perspective. The event-based kernel implementation was detailed.

Table 5.1 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`[2]	Attach an arbitrary task priority function to be used by the scheduler.
`ttAttachHook`[2]	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.

[2]Available in the C++ API only.

Table 5.1 (Continued)

Command	Description
`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.
`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.

6

Simulation Case Studies

6.1 Introduction

This chapter contains two simulation case studies performed using the TrueTime simulator. The first case study simulates communication and control of a three-joint robot system over TCP. The TCP communication layer is emulated on top of the link-layer protocols provided by the TrueTime network block. The second case study simulates a web server application where individual server threads are emulated using TrueTime tasks.

6.2 Network Communication and Control

This work was performed within the EU FP5 IST project Hard Real-time CORBA (HRTC). The objective of this project was to provide solutions that allow the distributed object model CORBA [Object Management Group, 2003] to be applied in hard real-time applications.

The work was focused on replacing the standard communication protocols used in ordinary CORBA and RT-CORBA with protocols that provide better temporal determinism. Ordinary CORBA communication is based on the IIOP (Internet Inter-operable Orb Protocol) which is layered on top of TCP/IP. Within the project IIOP was replaced by both TTP/C and by real-time switched Ethernet [Martinsson, 2002].

The CORBA object interface was restricted to one-way invocations only, i.e., a CORBA request from a client to a server does not generate any reply message back from the server to the client. In order to demonstrate the real-time properties of the different communication alternatives TrueTime was used. Networked control of a three-joint robot was used as the main example. In the sequel the experiment involving ordinary CORBA over IIOP (TCP/IP) is described.

Simulating TCP in TrueTime

The TrueTime network block simulates the basic properties of standard MAC (media access control) layer protocols. These protocols constitute the link layer in the Internet protocol stack, and are typically implemented in a network interface card, see [Kurose and Ross, 2001].

It is, however, straight-forward to also implement higher level protocols using TrueTime. Transport layer protocols, such as TCP and UDP, are usually implemented in software in the end systems, and may be emulated directly in the various TrueTime computer nodes using dedicated tasks or interrupt handlers.

A simple TCP implementation will be outlined below. In the simulation it is possible to specify TCP parameters such as sizes of the buffers at the receiving and sending ends, receive windows, maximum segment size (MSS), and acknowledgment time-outs. The receive windows are used to implement flow control. The window gives an indication of the free buffer space at the receiving side, and dictates how much data that can be transmitted on that specific connection. The window size is constantly updated by the receiving node, as messages are being read from the application layer. This information is sent back to the sender with each acknowledgment.

Opening a TCP Connection Since TCP is connection-oriented, a socket connection must be established before two nodes can start sending and receiving messages. When a connection is set up, sending and receiving buffers are created at each end of the connection. Special TrueTime sending and receiving tasks are also associated with each connection. Using tasks for the processing of incoming and outgoing TCP packets, it is possible to simulate overhead in the TCP layer. The functionality performed by these tasks will be described below.

Sending a TCP Data Message When sending a message over TCP it is divided in segments of size MSS which are sent in sequence to the receiving end, where the message is recreated. In addition to the data, each TCP data segment includes a header containing fields for source and destination identifiers, sequence number, acknowledgment number, and window size. When a segment is transmitted, a timer is created. If no acknowledgment has been received at the expiry of the timer, the segment is resent. The sending of a message is summarized in the following TrueTime pseudo-code

```
double TCPSend_code(int seg, void *data) {

  i = 0;
  ready = false;

  // Send all segments in send buffer and set up timers
  while (!ready) {
    // Take next segment from send buffer
    segment = sendBuffer->get(i);
    // Send if window allows
    if (segment->seqNbr <= sendWindow) {
      segment->ackNbr = lastRcv;
      ttSendMsg (segment->destination, segment, segment->size);
      time = ttCurrentTime() + TIMEOUT;
      Create timer for resending at t = time;
    } else {
      // Send window full, can not send
      ready = true;
    }
    // Increase buffer index
    i++;
    if (i == sendBuffer->currentSize()) {
      // No more segments in send buffer
      ready = true;
    }
  }
  return i * SND_OVERHEAD_TIME; // task execution time
}
```

Receiving a TCP Data Segment When a TCP segment arrives at a node, it is handled by a receiving task. An incoming TCP segment may be either a data segment or an acknowledgment of a previously transmitted segment. In the first case, it is checked if all preceding segments have been received. In this case the data is put in the receive buffer, otherwise the segment is discarded. An acknowledgment, with the latest received sequence number, is sent back to the source node. When all segments of a message have been received, the application layer is notified.

In the case that the incoming segment is a first-time acknowledgment, it works as a cumulative acknowledgment of all previous data, and the corresponding timers are removed. If we get a duplicate acknowledgment, however, this indicates that segments in between have been lost. In this case a fast re-transmit is performed, before the actual expiry of the timers of the segments. The implementation is summarized in the following TrueTime pseudo-code

```
double TCPReceive_code(int seg, void *data) {

  // Get segment from data link layer
  segment = ttGetMsg();

  if (segment contains data) {

    if (segment->seqNbr == lastRcv) {
      // have got all previous segments, put in buffer
      rcvBuffer->put(segment);
      lastRcv = segment->seqNbr + segment->size;
      Increase size of receive window;
    } else {
      // Out-of-order segment, ignore
    }
    // Send Ack
    ack->seqNbr = -1;
    ack->ackNbr = lastRcv;
    ack->window = rcvWindow;
    ack->source = segment->destination;
    ack->destination = segment->source;
    ttSendMsg(ack->destination, ack, ACKSIZE);
```

```
  } else {
    // We received an acknowledgement segment
    sendWindow = segment->window;
    if (segment->ackNbr > lastAck) {
       // new Ack
       lastAck = segment->ackNbr;
       Remove timeout timers;
       Delete segments from send buffer;
    } else {
      // same Ack as previously received
      Packets was lost, fast re-transmit;
    }
  }
  return RCV_OVERHEAD_TIME; // task execution time
}
```

Communicating with the Application Layer The sending task is triggered from the application layer when a user wants to send a message on the specific connection. Then the message is divided in segments and stored in the send buffer for subsequent transmission to the receiver. When the message is later reassembled at the receiving end the application layer is notified and the message can be read from the receive buffer.

The following example shows the code function for a controller node communicating with a sensor and actuator node over TCP.

```
double ctrl_code(int seg, void *data) {

  double *m;
  Task_Data* d = (Task_Data*) data;

  switch(seg) {
  case 1:
    ttTCPReceive(sensConn); // Blocks on sensor connection
                            // until a message arrives
                            // Notified from TCP layer
    return 0.0;
  case 2:
    m = (double*) ttTCPGet(sensConn); // Get TCP message data
                                      // from receive buffer
    d->y = *m;
```

```
    delete m;

    r = ttAnalogIn(1);
    // Compute control signal
    d->u = d->K*(r - d->y);

    return 0.0005;
  case 3:
    // Try to send a 4-byte message
    if (!ttTCPSend(4)) {
      // Send buffer full, can not send
      ttSetNextSegment(1); // Loop and wait for new message
    }
    return 0.0;
  case 4:
    m = new double;
    *m = d->u;
    ttTCPPut(actConn, m, 4); // Send 4-byte message to actuator
                             // Triggers the sending TCP task
    ttSetNextSegment(1); // Loop and wait for new message
    return 0.0;
  }
}
```

Simulations

The simulation setup included a sensor node, a controller node, and an actuator node communicating over TCP to control a three-joint robot system. The dynamics of the robot model was given by the standard model

$$M(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} + G(\theta) = \tau \tag{6.1}$$

where $\theta = (\theta_1, \theta_2, \theta_3)^T$ is the vector of joint angles of the robot, $M(\theta)$ is the mass matrix, $C(\theta, \dot{\theta})$ is the Coriolis matrix, and $G(\theta)$ is the gravity matrix. The simulation model also contained a friction term.

The robot was controlled using a computed-torque control law

$$\begin{aligned} \tau =& C(\theta, \dot{\theta})\dot{\theta} + G(\theta) + M(\theta)\ddot{\theta}_r \\ & - M(\theta)K_d(\dot{\theta} - \dot{\theta}_r) - M(\theta)K_p(\theta - \theta_r) \end{aligned} \tag{6.2}$$

The underlying network protocol used in the simulations was ordinary 100 Mbit Ethernet with the CSMA/CD protocol. This protocol

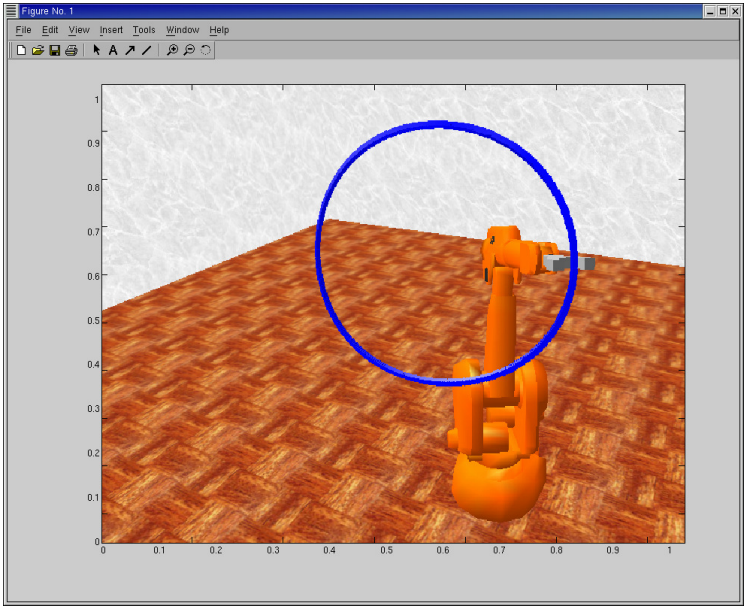

Figure 6.1 The result of the robot simulation without disturbing network traffic. Sensor values and control signals are communicated between a sensor node, controller node, and an actuator node over TCP.

uses exponential back-off in the case of collisions, which may cause long and unpredictable waiting times. The sampling frequency was 4 kHz, and enforced by the time-triggered task in the sensor node.

The results of the simulation were fed into the virtual robot environment presented in [Olsson, 2003] for visualization. The reference trajectory corresponded to a circle and the result of a simulation is shown in Figure 6.1.

By introducing disturbing traffic on the network, the effects of the unpredictability in the network communication can be studied. The schedule plots in Figure 6.2 and Figure 6.3 show the result of a network burst at time 0.2. As seen in Figure 6.2, the network traffic runs smoothly prior to the burst with periodic communication between the nodes. The communication in each period starts with the sensor node

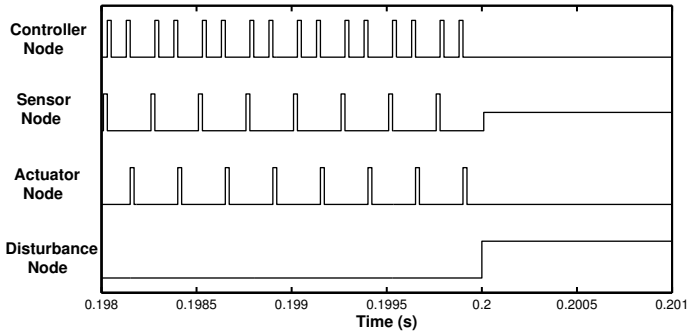

Figure 6.2 The network traffic just prior to the disturbance burst. Data messages and acknowledgments are communicated without interference with a sampling frequency of 4 kHz.

sending a data message to the controller node, which responds with an acknowledgment. The same pattern is seen in the communication between the controller and actuator nodes.

Figure 6.3 shows the network schedule during and after the burst. As a result of the collisions and the random waiting times of Ethernet, considerable jitter and delays are introduced in the communication. The resulting degraded control performance is shown in Figure 6.4, and it is seen how the disturbed communication causes deviations from the reference trajectory.

A communication scheme better suited for real-time traffic is switched scheduled Ethernet [Martinsson, 2002], which combines the attractive features of Ethernet (fast and inexpensive) with real-time guarantees. Using switched Ethernet each node has a full duplex connection to the switch, which isolates the collision domains. Still some non-determinism may be introduced by the switch, mainly caused by buffering. However, by making certain restrictions on how much traffic each node is allowed to generate, it is possible to compute upper bounds on the network latency. TrueTime was also used to simulate this communication strategy, and the effects of the disturbing traffic were eliminated.

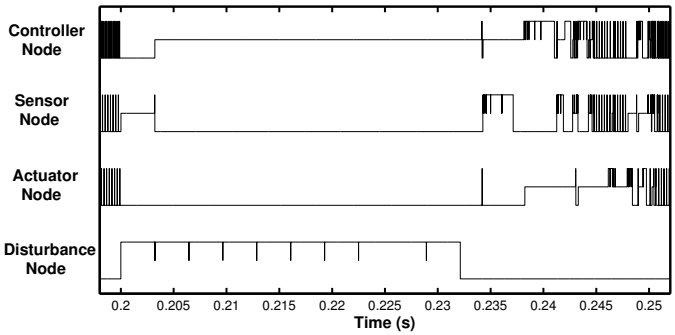

Figure 6.3 The resulting network traffic during and after a disturbance burst. Random delays and jitter are introduced by the network communication.

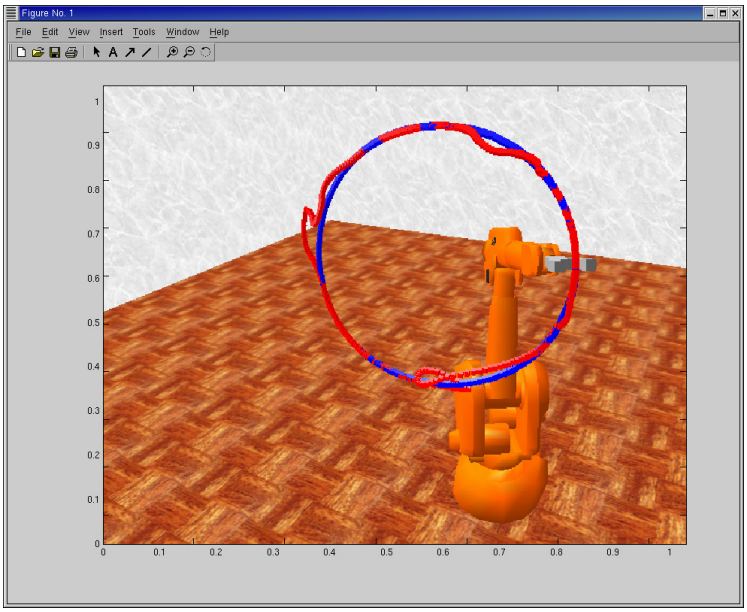

Figure 6.4 The control performance as a result of disturbing network traffic. Collisions causes delays that degrade the performance and leads to deviations from the reference trajectory.

6.3 A Web Server Application

The simulations described in this section were developed during a recent research visit to Tarek Abdelzaher at the University of Virginia. The group in Virginia are working on quality-of-service in web servers [Abdelzaher *et al.*, 2003] and schedulability analysis for aperiodic tasks [Abdelzaher and Lu, 2001]. This case study shows how these two areas may be integrated.

The use of control theory to achieve quality-of-service guarantees in modern web server applications, has been an active research area during recent years [Robertsson *et al.*, 2003; Sha *et al.*, 2002]. There are two main ways for the controller to influence the server load; by manipulating the arrival rate and by manipulating the service rate. The first approach is the most common, and is typically achieved by various admission control schemes.

However, in the following we will assume a web server where it is possible to modify the service rate by changing the CPU clock frequency and thereby the processing speed of the incoming request. The use of dynamic voltage scaling as an energy-saving mechanism in high-performance web servers was first presented in [Bohrer *et al.*, 2002].

The following case study will describe the simulation of a web server using TrueTime, and the use of feedback scheduling to achieve timing guarantees of incoming requests. The feedback scheduling scheme is based on the synthetic utilization concept introduced in Section 2.4.

Simulation in TrueTime

In the simulations, a client application sends web requests to a server emulating the basic properties of the HTTP/1.1 protocol. Using TrueTime, it is, e.g., possible to experiment with different processing times and priorities of the requests, different scheduling policies in the server node, and different control strategies to change the processing speed of incoming requests.

The Client The client node generates synthetic web requests that are sent to the server. The inter-arrival times, i.e., the time between the sending of each request on a connection, follows a bounded Pareto distribution. The Pareto distribution has been reported to fit measurements of real web traffic well [Crovella and Bestavros, 1997]. When

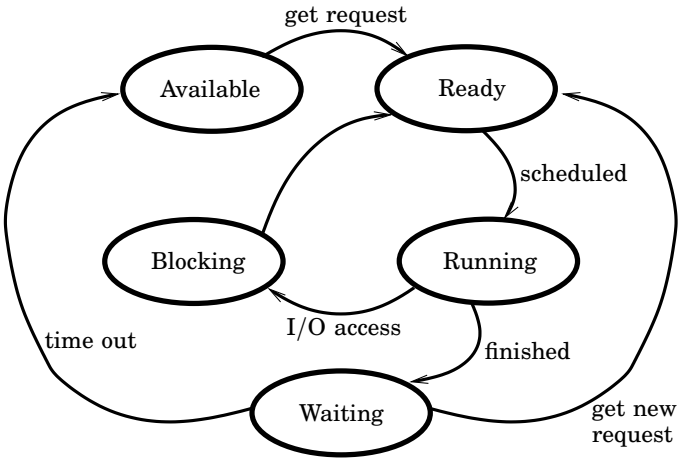

Figure 6.5 State diagram visualizing the processing of each web request by a server thread.

sending the first request on a closed connection (a connection is closed if no new requests are sent within the HTTP/1.1 TIMEOUT), the client awaits an acknowledgment from the server before sending new requests. Thereafter, requests are pipe-lined on the connection, i.e., multiple requests can be made without waiting for each response. Each request has associated simulated processing times and blocking times that it will consume on the server side. These are also Pareto distributed.

The Server The server node contains a number of server threads, and a high-priority thread that handles incoming requests. All incoming requests are time-stamped, and are then either forwarded to the socket queue of the server thread serving the connection, or put in a global request queue if there is no idle server thread available. When server threads are finished serving their current connection they check the global input queue for new requests.

Each server thread operates according to the state diagram shown in Figure 6.5. The times associated with the Running and Blocking states are Pareto distributed and specified in each request. When the server thread is finished it waits for a certain amount of time for new

requests, before closing the connection. The TrueTime code function implementing each server thread is given below

```
function [exectime, data] = threadcode(seg, data)

switch seg,

 case 1,
  request = ttTryFetch(data.socketQ);
  if (isempty(request))
    % Socket queue empty
    ttSetNextSegment(5);
    exectime = 0.0;
  else
    % Store request attributes
    data.C = request.C;
    data.B  = request.B;
    data.conn = request.conn;
    data.arrival = request.arrival;
    data.start = ttCurrentTime;
    % Simulate processing phase 1
    exectime = data.C;
  end
 case 2,
  % Simulate I/O access (not using CPU)
  ttSleep(data.B);
  exectime = 0.0;
 case 3,
  % Processing phase 2
  exectime = 0.001;
 case 4,
  % Request served
  ttSetNextSegment(1); % loop back and serve next request
  exectime = 0.0;
 case 5,
  % All requests served, block waiting for more
  % requests from the same connection
  now = ttCurrentTime;

  % Create timer, terminates job on expiry
  timeout = now + TIMEOUT;
```

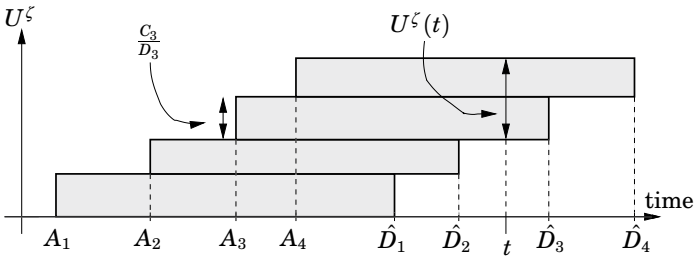

Figure 6.6 The synthetic utilization during a busy-period.

```
  ttCreateTimer(data.timer, timeout, data.timeoutHandler);
  % Block until new request arrives or timer expires
  ttWait(data.reqEvent);
  exectime = 0.0;
 case 6,
  % New request arrived before time-out, serve it
  ttRemoveTimer(data.timer)
  ttSetNextSegment(1);
  exectime = 0.0;
end
```

A Feedback Scheduling Scheme

The feedback scheduling scheme is aimed at controlling the average synthetic utilization around the schedulability bounds given by Equations (2.3) and (2.4) in Section 2.4. This way it is guaranteed that all requests meet their deadlines and fulfill their QoS specifications.

A schematic diagram showing the synthetic utilization evolving over time is shown in Figure 6.6. At each arrival, k, the synthetic utilization is increased by $\frac{C_k}{D_k}$. The absolute deadlines of each request are denoted $\hat{D}_k = A_k + D_k$. At the deadline of the request, the synthetic utilization is decreased by the same amount. The instantaneous synthetic utilization at any point in time, is given by the height of the shaded area. When the processor becomes idle, the synthetic utilization is reset to zero. The time from the first arrival to the time when the processor again becomes idle, is called a *busy-period*.

For control purposes we want to obtain a relationship between the processing speed and the average synthetic utilization, $\hat{U}^\zeta$, in the busy-

period. Since the contribution of each arriving request is $\frac{C_k}{D_k}$, the area of each rectangle in Figure 6.6 is exactly C_k. To obtain the average synthetic utilization we sum up the areas and divide with the base length, i.e., the difference between the largest absolute deadline and the first arrival time in the busy-period. For the example in the figure, we get

$$\hat{U}^{\zeta} = \frac{\sum_{i=1}^{i=4} C_i}{\hat{D}_4 - A_1} \tag{6.3}$$

We will consider a scheme where the control action is changed after each departed request. Therefore we need to update the average synthetic utilization between each departure. Assume that n request have arrived at time, t_1, of the first departure. Denoting the processor frequency, μ, the average synthetic utilization, $\hat{U}_1^{\zeta}$, at time t_1 can be written

$$\hat{U}_1^{\zeta} = \frac{\frac{1}{\mu}\sum_{i=1}^{i=n} C_i}{\max_{1<i\leq n} \hat{D}_i - A_1} \tag{6.4}$$

Then assume that m new requests arrive before the next departure time, t_2. The average synthetic utilization, $\hat{U}_2^{\zeta}$, at time t_2 can then be written

$$\hat{U}_2^{\zeta} = \frac{\frac{1}{\mu}\left(\sum_{i=1}^{i=n} C_i + \sum_{i=n+1}^{i=m} C_i\right)}{\max_{1<i\leq n+m} \hat{D}_i - A_1} \tag{6.5}$$

Combining Equations (6.4) and (6.5) we can write

$$\hat{U}_2^{\zeta} = \frac{\hat{U}_1^{\zeta} \cdot (\max_{1<i\leq n} \hat{D}_i - A_1) + \frac{1}{\mu}\sum_{i=n+1}^{i=m} C_i}{\max_{1<i\leq n+m} \hat{D}_i - A_1} \tag{6.6}$$

We thus arrive at a recursive update equation for the synthetic utilization

$$\hat{U}_k^{\zeta} = \beta_k \hat{U}_{k-1}^{\zeta} + u_k \gamma_k \tag{6.7}$$

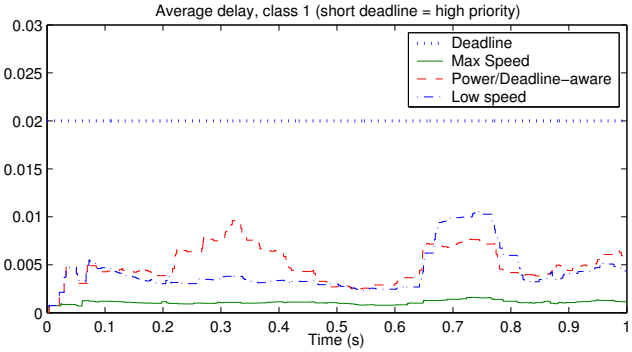

Figure 6.7 Average delay for the high-priority class.

where

$$\begin{aligned} \beta_k &= \frac{\max_{1<i<n} \hat{D}_i - A_1}{\max_{1<i<n+m} \hat{D}_i - A_1} \\ \gamma_k &= \frac{\sum_{i=n+1}^{n+m} C_i}{\max_{1<i<n+m} \hat{D}_i - A_1} \\ u_k &= \frac{1}{\mu_k} \in [0.2, 1] \end{aligned} \tag{6.8}$$

and n is the number of arrivals up until departure $k-1$, and m is the number of new arrivals between departure $k-1$ and k.

The controller then uses the update equation (6.7) to compute the average synthetic utilization at each invocation. An event-based P-controller may then be used to control the system

$$u_k = K \cdot (U_{bound} - \hat{U}_{k-1}) \tag{6.9}$$

The processing times, C_i, in the above equations will be estimated on-line using the recursive first-order filter

$$C_i = \lambda \cdot C_{i-1} + (1-\lambda) \cdot c_i \tag{6.10}$$

where c_i is the last measured processing time.

Simulations

A simulation with two classes of requests served by two server threads was run and the results are shown in Figures 6.7 and 6.8. The requests

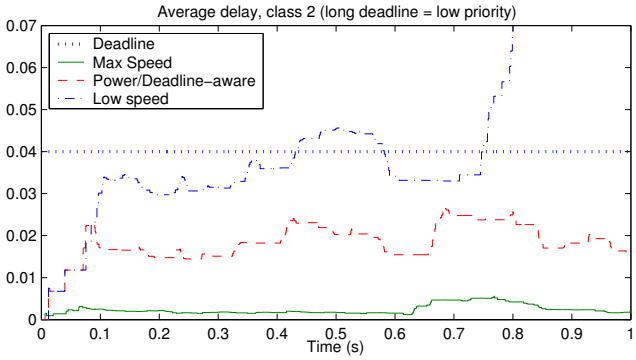

Figure 6.8 Average delay for the low-priority class.

for the two classes had different deadlines of 0.02 and 0.04 time units, respectively.

The average input load during the simulation corresponded to a 300 per cent CPU utilization when running at the lowest speed, $\mu = 1$, and 60 per cent utilization running at the highest speed, $\mu = 5$. The server threads processing the requests were scheduled using deadline-monotonic scheduling based on the request deadlines. Running at a constant low speed, the low-priority requests are processed at a lower rate than they arrive and thus the queues accumulate and the delay increases. However, running constantly at the highest speed, the average delay for both classes is well below the respective deadlines, and the system is clearly under-utilized.

The results using the proposed feedback scheduling technique are shown by the dashed curves, and the deadlines are met for both classes. The average synthetic utilization and the utilization set-point are shown in Figure 6.9. The synthetic utilization is reset to zero when the system becomes idle.

The margin to the deadline is quite large even for the low-priority class. This is due to the pessimism involved when applying the schedulability results to such a small number of tasks. The results are expected to provide better guidelines when the number of tasks increase.

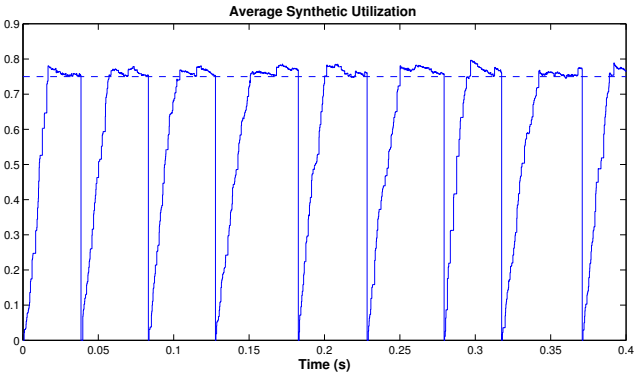

Figure 6.9 Average synthetic utilization and set-point.

6.4 Summary

In the first part of the chapter it was shown how TrueTime could be used to simulate the basic functionality of TCP. A simulation was described, where a robot system was controlled over TCP using ordinary Ethernet as the under-lying MAC protocol.

The second part described a feedback scheduling scheme used for web server delay control. The scheme was based on the concept of synthetic utilization for aperiodic tasks. TrueTime was used to simulate the web server system and evaluate the scheduling strategy.

7

Conclusions

7.1 Summary

This thesis has been dealing with flexible approaches in the design of real-time control systems. Two main contributions were presented: a flexible implementation scheme for model predictive controllers, and a simulator for real-time control system co-design. The thesis also contained a general discussion and overview of feedback scheduling techniques for real-time control systems. A summary of the contributions is given below.

Flexible Implementation of Model Predictive Control

Many control algorithms exhibit large variations in their execution time characteristics. One prominent example is model predictive control (MPC) algorithms, where the control signal in each sample is obtained from the on-line solution of a constrained optimization problem. The execution time of the optimization is often decided by external factors, such as disturbances and changing reference values.

Since the optimization algorithm is iterative and may be aborted, there exists a dynamic trade-off between the computational delay and the quality of the computed control signal. The standard cost function used in the MPC formulation was modified to also contain effects of the computational delay. By minimizing the new delay-dependent cost index, the computational requirement of the algorithm was adjusted

in a way that significantly improved the control performance.

Another observation is that the cost index can be used in a dynamic scheduling context. By always scheduling the MPC task with the highest value of its cost index, a fair arbitration of the computing resources is obtained. Given the highly changing computational requirements of MPC tasks, a standard fixed priority assignment is clearly insufficient.

Simulation with TrueTime

TrueTime is a toolbox for MATLAB/Simulink that extends the traditional control system simulation facilities with two Simulink blocks; a kernel block and a network block. The functionality provided by these blocks allows co-simulation of the continuous process dynamics, the controllers implemented as tasks in the real-time kernel, and the effects of network communication.

The flexibility of the kernel allows experimentation with dynamic compensation and scheduling schemes, while evaluating the result on the performance of the controlled plants. The evaluation of the MPC schemes was performed using TrueTime, and the simulator was also demonstrated in two simulation case studies.

The first case study described a simulation of TCP on top of the data-link protocols provided by the TrueTime network block. Dedicated tasks in the various nodes where used to implement the basic TCP functionality. Networked control of a robot system using TCP on top of ordinary Ethernet was described.

In the second case study TrueTime was used to simulate the basic properties of a web server. The web server node contained a number of server threads used to serve incoming requests. A feedback scheduling scheme based on schedulability results for aperiodic tasks was used to schedule the server threads and thereby guarantee that the timing requirements of individual connections were not violated.

7.2 Future Work

The work presented in this thesis can be extended in many ways. A few suggestions of possible extensions are given below.

General Feedback Scheduling Structures

The basic feedback scheduling structure could be extended to also include more involved combinations of tasks. One example would be to combine control tasks of anytime nature with ordinary control tasks where the control performance decreases monotonically with the input-output latency and sampling interval. In this case it is not trivial how to assign the computation resources. It could also be possible to directly control timing parameters such as delays and jitter.

Another possibility includes the direct approach to feedback scheduling, where the scheduling decisions are made based on the current cost of the different control tasks. What is the best way to design the cost functions and how should the resulting event-based system be analyzed?

MPC

The work on scheduling of MPC tasks may be extended in a number of ways. One question is what happens if the task has not been terminated at the deadline. Is it then better to abort and output the control signal or to continue the optimization into next sample. Another extreme is when there is no execution time available at all. In this case it may be possible to use the previous solution sequence shifted one step. The stability issue of MPC under varying time delays and using sub-optimal solutions is another interesting area.

Visual Servoing

Algorithms of anytime nature can be found in many other application areas than model predictive control. One example is vision-based control in, e.g., robotics. In tracking-based algorithms there exist an interesting trade-off between the computational delay of the image processing algorithm and the resulting quality of the image used for feedback control.

TrueTime Extensions

The TrueTime kernel could be extended and be made more realistic. Currently it is possible to simulate context switch overhead, but the kernel model could also include interrupt latencies and execution times associated with the various real-time primitives. One major limitation with TrueTime is the question of how to assign the execution times of tasks. One possibility would be to integrate TrueTime with available compiler/execution time analysis tools. Another obvious extension would be to include support for more network protocols, e.g., wireless communication protocols.

8

Bibliography

Abdelzaher, T. and C. Lu (2001): "Schedulability analysis and utilization bounds for highly scalable real-time services." In *Proceedings of the 22nd IEEE Real-time Systems Symposium*. London, UK.

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**.

Abdelzaher, T. F., J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu (2003): "Feedback performance control in software services." *IEEE Control Systems Magazine*, **23:3**, pp. 74–90.

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.

Årzén, K.-E. (1999): "A simple event-based PID controller." In *Preprints 14th World Congress of IFAC*. Beijing, P.R. China.

Åström, K. J. and B. Bernhardsson (1999): "Comparison of periodic and event based sampling for first-order stochastic systems." In *Preprints 14th World Congress of IFAC*, vol. J, pp. 301–306. Beijing, P.R. China.

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.

Bartlett, R. A., A. Wächter, and L. T. Biegler (2000): "Active set vs. interior point strategies for model predictive control." In *Proceedings of the American Control Conference*. Chicago, Illinois.

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.

Bemporad, A., L. Chisci, and E. Mosca (1994): "On the stabilizing property of SIORHC." *Automatica*, **30:12**, pp. 2013–2015.

Bemporad, A., M. Morari, V. Dua, and E. N. Pistikopoulos (2002): "The explicit linear quadratic regulator for constrained systems." *Automatica*, **38:1**, pp. 3–20.

Bohrer, P., E. Elnozahy, M. Kistler, C. Lefurgy, C. McDowell, and R. Mony (2002): *The Case for Power Management in Web Servers*. Power Aware Computing, Kluwer Academic Publications.

Bollella, G., B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull (2000): *The Real-Time Specification for Java*. 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*. Madrid, Spain.

Cannon, M., B. Kouvaritakis, and J. A. Rossiter (2001): "Efficient active set optimization in triple mode MPC." *IEEE Transactions on Automatic Control*, **46:8**, pp. 1307–1312.

Cervin, A. (2003): *Integrated Control and Real-Time Scheduling*. PhD thesis ISRN LUTFD2/TFRT--1065--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Cervin, A. and J. Eker (2004): "Control-scheduling codesign of real-time systems: The control server approach." *Journal of Embedded Computing*. To appear.

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

Chung, J.-Y., J. Liu, and K.-J. Lin (1990): "Scheduling periodic jobs that allow imprecise results." *IEEE Trans on Computers*, **39:9**.

Crovella, M. E. and A. Bestavros (1997): "Self-similarity in world wide web traffic: Evidence and possible causes." *ACM/IEEE Transaction on Networking*, **5:6**.

Dunbar, W. B. and R. M. Murray (2002): "Model predictive control of coordinated multi-vehicle formations." In *Proceedings of the 41st IEEE Conference on Decision and Control*. Las Vegas, NV.

Dunbar, W. B., M. B. William, R. Franz, and R. M. Murray (2002): "Model predictive control of a thrust-vectored flight control experiment." In *Proceedings of the 15th IFAC World Congress on Automatic Control*. Barcelona, Spain.

Eker, J. and A. Blomdell (2000): "A contract-based language for embedded control systems." In *Proc. 25th IFAC/IFIP Workshop on Real-Time Programming*.

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., P. Hagander, and K.-E. Årzén (2000): "A feedback scheduler for real-time control tasks." *Control Engineering Practice*, **8:12**, pp. 1369–1378.

El-khoury, J. and M. Törngren (2001): "Towards a toolset for architectural design of distributed real-time control systems." In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. London, England.

Fletcher, R. (1991): *Practical methods of optimization 2nd ed.* John Wiley & Sons Ltd.

Garcia, C. E., D. M. Prett, and M. Morari (1989): "Model predictive control: Theory and practice – a survey." *Automatica*, **25:3**, pp. 335–348.

Gestegård Robertz, S. (2003): "Flexible automatic memory management for real-time and embedded systems." Technical Report Licentiate Thesis. Department of Computer Science, Lund Institute of Technology, Lund, Sweden.

Hanselmann, H. (1987): "Implementation of digital controllers – a survey." *Automatica*, **23:1**, pp. 7–32.

Henriksson, D. and A. Cervin (2003): "TrueTime 1.13—Reference manual." Technical Report. Department of Automatic Control, Lund Institute of Technology.

Joseph, M. and P. Pandya (1986): "Finding response times in a real-time system." *The Computer Journal*, **29:5**, pp. 390–395.

Kopetz, H. (1997): *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, Boston, MA.

Kouvaritakis, B., M. Cannon, and J. Rossiter (2002): "Who needs QP for linear MPC anyway?" *Automatica*, **38:5**, pp. 879–884.

Kurose, J. F. and K. W. Ross (2001): *Computer Networking – A Top-Down Approach Featuring the Internet*. Addison-Wesley.

Lincoln, B. (2003): *Dynamic Programming and Time-Varying Delay Systems*. PhD thesis ISRN LUTFD2/TFRT--1067--SE.

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**, pp. 65–75.

Liu, J., K.-J. Lin, W.-K. Shih, A. Yu, J.-Y. Chung, and W. Zhao (1991): "Algorithms for scheduling imprecise computations." *IEEE Trans on Computers*.

Liu, J., W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung (1994): "Imprecise computations." *Proceedings of the IEEE*, **82:1**, pp. 83–94.

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, S. Son, and G. Tao (2002): "Feedback control real-time scheduling: Framework, modeling and algorithms." *Real-time Systems*, **23:1/2**, pp. 85–126.

Maciejowski, J. M. (2002): *Predictive Control with Constraints*. Prentice-Hall.

Martinsson, A. (2002): "Scheduling of real-time traffic in a switched ethernet network." Technical Report Masters thesis ISRN LUTFD2/TFRT--5683--SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Mayne, D. Q., J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert (2000): "Constrained model predictive control: Stability and optimality." *Automatica*, **36:6**, pp. 789–814.

Object Management Group (2003): "CORBA: Common Request Broker Architecture." Home page, http://www.omg.org/.

Olsson, T. (2003): "Virtual Environment for Robot Vision." Home page, http://www.control.lth.se/~tolsson/vr/vrobot.html.

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*. Orlando, Florida.

Qin, S. J. and T. A. Badgwell (2003): "A survey of industrial model predictive control technology." *Control Engineering Practice*, **11:7**, pp. 733–764.

Richalet, J. (1993): "Industrial application of model based predictive control." *Automatica*, **29**, pp. 1251–1274.

Robertsson, A., B. Wittenmark, and M. Kihl (2003): "Analysis and design of admission control in web-server systems." In *Proceedings of ACC'03*.

Schinkel, M., W.-H. Chen, and A. Rantzer (2002): "Optimal control for systems with varying sampling rate." In *Proceedings of American Control Conference*. Anchorage.

Scokaert, P. O. M., D. Q. Mayne, and J. B. Rawlings (1999): "Suboptimal model predictive control (feasibility implies stability)." *IEEE Transactions on Automatic Control*, **44:3**, pp. 648–654.

Sha, L., X. Liu, Y. Lu, and T. Abdelzaher (2002): "Queuing model based network server performance control." In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*. Austin, TX.

Shin, K. and C. Meissner (1999): "Adaptation and graceful degradation of control system performance by task reallocation and period modification." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 29–36. York, UK.

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*, pp. 160–169.

The Mathworks (2001a): *Real-Time Workshop; User's Guide*. The MathWorks Inc., Natick, MA.

The Mathworks (2001b): *Simulink: A Program for Simulating Dynamic Systems – User's Guide*. The MathWorks Inc., Natick, MA.

Tiller, M. (2001): *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers.

Tindell, K., A. Burns, and A. J. Wellings (1992): "Mode changes in priority preemptively scheduled systems." In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pp. 100–109.

Wright, S. J. (1997): *Primal-Dual Interior-Point Methods*. SIAM.