Merging Real-Time and Control Theory for Improving the Performance of Embedded Control Systems

Cervin, Anton

2004

Research Report:

Merging Real-Time and Control Theory for Improving the Performance of Embedded Control Systems

Anton Cervin

Department of Computer Engineering and Systems Science
University of Pavia, Italy

September 15, 2004

Project Supervisor: Giorgio Buttazzo

Sponsoring Organizations: EU/ARTIST, EU/ARTIST2

Abstract

This report describes the work carried out within the research project "Merging Real-Time and Control Theory for Improving the Performance of Embedded Control Systems". The overall objective of the work has been to develop integrated control and scheduling methods for improving the performance of real-time control systems with limited resources. The work has fallen into three categories. First, overrun methods for control tasks has been investigated. Specifically, a reservation-based scheduling concept called the *control server* has been further developed, and control experiments on a ball-and-place process have been performed. Second, the issue of jitter in real-time control systems has been explored. The concept of *jitter margin* has been introduced as a link between control stability theory and scheduling theory. In this context, best-case response-time analysis under earliest-deadline-first scheduling has been researched. Third, some development work on the S.Ha.R.K. real-time kernel has been performed. The rate-monotonic and earliest-deadline-first scheduling modules have been extended, and new modules for the elastic task model and the control server model have been implemented.

Contents

1. Introduction

It is generally agreed that the research area of real-time scheduling theory was born when Liu and Layland published their seminal work on periodic task scheduling in the early seventies [Liu and Layland, 1973]. What fewer people recall is that their work was motivated by and firmly based on the assumptions of digital process control. Sampled-data control theory had been developed during the fifties and sixties to cover the case where a continuous-time plant was controlled by a digital computer. The work of Liu and Layland facilitated the closing of multiple control loops over the same computer, thus allowing the computational resources to be used more efficiently. They also carefully pointed out that their proposed algorithms could introduce a computational delay of up to one sample in each control loop, and that this should be accounted for in the control design.

Much of the real-time theory developed since the seventies has focused on different task models rather than on specific applications. Hence, the controller timing aspects have tended to be forgotten. Similarly, the majority of recent developments in control theory has been far from implementation-oriented. This has created an increasing gap between control theory and real-time theory that has only recently begun to be recognized and addressed, e.g., [Seto *et al.*, 1996; Albertos and Crespo, 1997; Törngren, 1998; Årzén *et al.*, 1999].

It should be pointed out that the well-established theories of hard real-time systems and sampled-data systems are perfectly applicable as long as the computational resources are aplenty. It is then possible to use high sampling rates and to base the real-time design on worst-case assumptions on task activation rates and execution times. In many embedded applications, however, the cost of the computing hardware is an important factor, and the developer must strive to use as lightweight a platform as possible. In the case of limited computing resources, the choice of scheduling algorithm and other design issues in the implementation can have a large impact on the control performance.

Summary of Work and Outline

The common theme in this research project has been how to handle timing variations in real-time control. In Chapter 2, control algorithms with varying execution times are studied. The traditional approach to such tasks has been to design the real-time system in accordance with the worst-case execution times. Here, a dynamic approach is explored, where the occasional long execution times are treated as exceptions. The application can be informed about its timing status and adjust its own actions and activations based on this information. This work is an extension of the scheduling methods developed in [Abeni and Buttazzo, 1998; Caccamo *et al.*, 2000; Cervin *et al.*, 2003], coupled with the control analysis developed in [Lincoln and Cervin, 2002]. For experimental validation, control-scheduling experiments with a ball-on-place control application have been performed.

In the Chapter 3, the timing variations are seen from the point of view of the controller. The delay variations are not necessarily the result of varying execution times, but rather of the scheduling algorithm itself. Under the standard rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling algorithms, various amount of *input-output jitter* are introduced in the control loops. Using response-time analysis [Joseph and Pandya, 1986; Spuri, 1996; Redell and Sanfridson, 2002], the jitter may be quantified for each task. Here, the theory is extended to give a lower bound of the best-case response time under EDF scheduling. Also, based on a recent stability theorem for control systems with jitter [Kao and Lincoln, 2004], we propose the notion of *jitter margin* and show how it can be used to link the response-time analysis to control stability analysis.

Chapter 4 reports on the implementation and modification of various scheduling modules in the S.Ha.R.K. real-time operating system [Gai *et al.*, 2001]. One way to reduce jitter in control applications is to use dedicated tasks for input and output actions. To synchronize the different communicating tasks, offsets can be used. Hence, task release offsets have been added to the EDF and RM scheduling modules. These modules have also been extended to handle deadlines less than the period, also suitable for the scheduling of dedicated input and output tasks. For the overrun handling, a new scheduling module for the control server task model has been developed. For tasks with more slowly time-varying computational requirements changes, a scheduling module for the elastic task model [Buttazzo *et al.*, 1998] has been implemented.

Finally, in Chapter 5, the results are summarized and some suggestions for future work, both theoretical and practical, are given.

2. Overrun Handling in Real-Time Control Systems

2.1 Background and Motivation

In embedded control systems, the computational resources are limited and must be used as efficiently as possible. This can prevent the use of high sampling rates and a real-time design based on worst-case execution times. In this section, we study the particular problem of control applications with execution times that can vary from sample to sample.

At the heart of the problem lies a trade-off between the control task sampling period and the probability of execution overruns. The CPU utilization U_i of a task is given by

$$U_i = \frac{C_i}{T_i}, \tag{2.1}$$

where C_i is the worst-case execution time and T_i is the task period. It is seen that, for a given value of U_i, a large enough T_i must be chosen to accommodate the largest possible execution time. It is well known that an overly long sampling period leads to degraded control performance. Hence, it can be tempting to choose a smaller T_i than what is dictated by (2.1). The penalty that must be payed is that of occasional execution overruns. If the performance loss due to the overruns is smaller than the performance gain due to the shorter sampling period, then such a design could be considered "better" than the classical worst-case design.

The influence of the sampling interval on the control performance is relatively easy to understand and to compute. The consequence of execution overruns is considerably more difficult to predict. The result depends on a large number of factors: the basic scheduling algorithm, the specific overrun handling method, the execution-time characteristics of the control task, the controller and plant dynamics, whether or not dynamic control compensation is used, etc, etc. Hence, it is simply not possible to devise an overrun handling method that is "the best" for all control applications.

Varying execution times may stem from the control application itself or have their origin in the operating system or the computing hardware. Examples in the first category include discrete logic and data-dependencies in the control algorithm. In the second category we find unaccounted interrupts and hardware effects such as cache misses. We are concerned with occasional, "random" overruns that deviate from the *nominal execution-time*, denoted by C_{nom}.

One way to try to model the variations in execution time is to use a probability distribution function. Figure 2.1 shows an example of what such a function might look like. In the example, the probability function is built from a point distribution at C_{nom} and a uniform distribution between C_{nom} and the true worst-case execution time, C_{max}. It should be noted that, in a real system, the execution times from sample to sample are typically not independent, and it can be very hard to estimate the maximum execution time.

The overrun handling methods explored in this chapter are based on *reservation-based* scheduling algorithms, where a given fraction of the CPU is assigned to each task. Such algorithms provide *temporal isolation* and make it easier to reason about each task separately. Also, we assume *local overrun handling*, which means that no global resource repartitioning takes place in the case of an overrun.

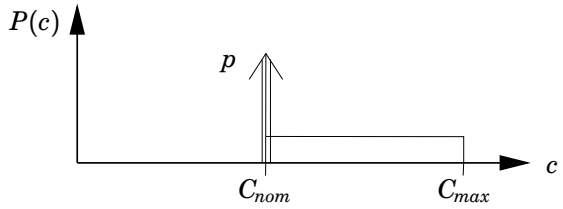

Figure 2.1 Example of an execution-time probability density function that could be used to model control tasks with varying execution time. In this example, a point distribution and a uniform distribution and a total of three parameters are used.

The novelty of the work lies in the controller timing analysis and the introduction of feedback from the scheduler to the application regarding the current timing status. This enables the controller to compensate for the overruns.

Server-Based Scheduling

In the hard-real-time scheduling literature, execution-time and deadline overruns are seldom mentioned. This is quite natural, since the definition of a hard real-time system implies that such overruns may never occur. It is well known, however, that ordinary hard scheduling algorithms such as RM and EDF do a poor job in the case of overruns. Under either strategy, an overrun in one task may lead to unpredictable deadlines misses in the other tasks [Buttazzo, 2003].

If it is known that overruns may occur, a good design methodology demands that they are explicitly handled by the real-time system. One way to do so is to schedule tasks with variable computation time using *servers*. The server concept was originally introduced for handling aperiodic tasks (i.e., tasks without a known minimum interarrival rate) in hard real-time systems. The aperiodic servers were first developed for fixed-priority scheduled systems [Lehoczky *et al.*, 1987]. Later, the ideas were also extended to deadline-scheduled systems [Spuri and Buttazzo, 1996].

A server that was designed to handle both aperiodic tasks and tasks with unknown and variable execution times is the *constant bandwidth server* (CBS) [Abeni and Buttazzo, 1998]. A CBS creates the abstraction of a virtual CPU with a given capacity (or *bandwidth*) U_s. A task executing within the CBS cannot consume more than the reserved capacity. Hence, from the outside, the CBS will appear as an ordinary EDF task with a maximum utilization of U_s. The time granularity of the virtual CPU abstraction is determined by the *server period* T_s.

The Control Server

The overrun handling methods explored in this work are based on a further development of the constant bandwidth server, called the *control server* [Cervin and Eker, 2003]. The control server is a scheduling mechanism tailored to control tasks that combines three different ideas:

- *Reservation-based scheduling.* Each task is scheduled by a modified constant-bandwidth server, where a dynamic server period is used.
- *Subtask scheduling.* A task may be divided in several *segments* that are scheduled as subtasks. Scheduling the two main parts of a control algorithm (Calculate and Update) as subtasks, the input-output latency of the controller can be reduced [Cervin, 1999].
- *Time-triggered I/O.* Inputs can be read and outputs can be written at predefined points in time by the kernel, minimizing the jitter in the control actions [Halang, 1993; Henzinger *et al.*, 2001].

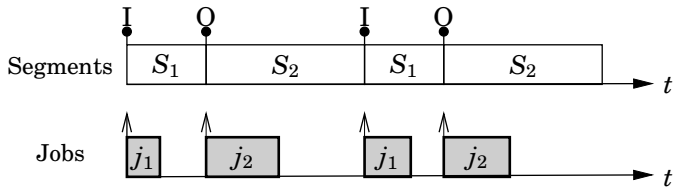

Figure 2.2 A periodic Control Server task with two segments.

An illustration of a periodic control server task is given in Figure 2.2. The task is divided into two segments, S_1 and S_2. The segments can be viewed as a static schedule for the input and output operations. At the beginning of S_1, an input is read (I), and at the end of S_1, an output is written (O). At the beginning of each segment, a job associated with the segment is released.

Formally, a control server task τ_i is described by

- a CPU share U_i,
- a period T_i,
- a release offset ϕ_i,
- a set of $n_i \geq 1$ segments $S_i^1, S_i^2, \ldots, S_i^{n_i}$ of lengths $l_i^1, l_i^2, \ldots, l_i^{n_i}$ such that $\sum_{j=1}^{n_i} l_i^j = T_i$,

Associated with each segment S_i^j are

- an optional input hook I_i^j,
- a code function f_i^j, and
- an optional output hook O_i^j.

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

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

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

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

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

In the case of overruns, the control server, as defined above, queues any pending jobs. Also, the definition above assumes periodic tasks only. In this work, the server has been modified to also handle aperiodic tasks. Furthermore, the possibility to skip jobs has been introduced by new functions in the control server API.

Related Work

Scheduling of systems that allow skips is treated in [Koren and Shasha, 1995] and [Ramanathan, 1997]. The latter paper considers scheduling that guarantees that at least k out of n instantiations will execute. A slightly different motivation for skipping samples is presented in [Caccamo and Buttazzo, 1997]. Here the main objective is to use the obtained execution time to enhance the responsiveness of aperiodic tasks.

A variant of the constant-bandwidth server specifically designed to handle overruns in real-time control systems is presented in [Caccamo *et al.*, 2002]. The proposed server, called CBS^{hd}, differs from the original CBS by postponing the deadline only by the amount needed to complete the job. In this way, the task can be scheduled more efficiently and finish earlier. The approach assumes that the worst-case execution time of the task is known. Unfortunately, the paper's control-theoretical foundation is quite weak. It is not explained when sampling and actuation are performed, the input-output latency is ignored, and it is assumed that the controller can change sampling rate with zero penalty. Furthermore, in the performance evaluation, a steady-state performance index is used to compute the performance of a controller that switches between different sampling intervals.

2.2 Analysis of Basic Overrun Strategies

In this section, we will explore three basic overrun strategies for control tasks, called Queue, Abort, and Skip. It is assumed that measurement samples arrive periodically, each triggering the release of a job. A very simple model will be used, where the control task is assumed to be running in isolation on a dedicated processor. This approximates the behavior of a task running in a bandwidth server when the rest of the processor is being fully utilized. It also models the case where a task executes in a server with *hard reservations*, i.e., a system where the task cannot consume more than its budget in each period.

For the output action, two different models will be analyzed. In the first model, the output is written at the end of the period. If no control signal has been computed at this point, the previous output value is held. This is the default behavior of a control server task with one segment. In the second model, the output is written no sooner than at the end of the period. In the case of an overrun, the output is be written as soon as the task finishes (unless the task has been aborted, of course). This behavior can be be approximated using a control server task with a short extra segment at the end for the output operation. (Note that this scheme will also introduce a small jitter in the output.)

The Queue Strategy. The queue strategy is the default strategy in the control server. The strategy is illustrated in Figure 2.3. When an overrun occurs, the following job is queued and can start once the first instance completes.

Allowing the first job to complete, the second job will be delayed, introducing extra input-output latency. Also, since the second job is released late, it is less likely be able to complete before the third job is released. If several long execution times occur in a row, a long queue of jobs may build up in the server.

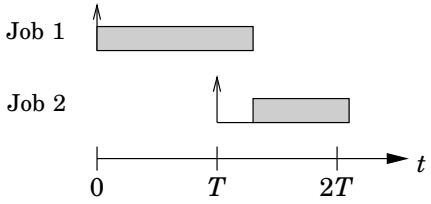

Figure 2.3 The Queue strategy. The second job is queued and can start once the first job completes.

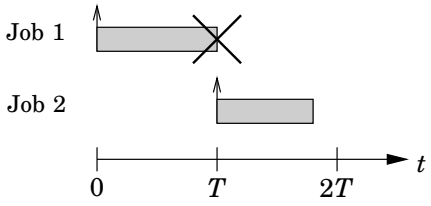

Figure 2.4 The Abort strategy. When the first job overruns, it is aborted, allowing the second job to start immediately.

As with the jobs, the samples themselves may or may not be queued. If they are queued, the input-output latency will grow longer as more and more jobs are queued. If they are not queued, several instances of the control algorithm will act on the same measurement data, which may be unnecessary.

To prevent the server queue from growing indefinitely, the server can be modified so that only a single job is queued, while the rest are discarded. The same also makes sense for the measurement samples. Acting on old data when fresh samples are available is not a good idea for feedback applications. Hence, for control tasks, it is reasonable to queue only one job and one sample. We call this modified strategy Queue(1).

The Abort Strategy. In the Abort strategy, only the execution-measurement part of the server is used. The strategy is illustrated in Figure 2.4. When an overrun occurs, the job is aborted, allowing the next job to be released on time.

This strategy only makes sense as long as the overrun is not task-dependent, but caused by external, random events such as cache misses or bursty interrupts. Aborting the current job, the next job can start over with fresh measurement data, possibly decreasing the input-output latency. If the job execution times are not independent random variables, however, several overruns in a row can prevent the controller from updating its output during a long interval.

A problem with the Abort strategy is that it can be difficult to implement. Generally, operating systems do not support asynchronous transfer of control (program flow). Hence, a task can typically not be aborted until a real-time primitive is called. One possibility is to insert extra checkpoints in the code. Another possibility is to lower the priority (or equivalent) of the task and let it run in the background until it has finished. Meanwhile, a new task, taken from a task pool, is used to execute the next job. Such a scheme will introduce additional overhead. Care must also be taken such that the task data is in a consistent state throughout.

The Skip Strategy. In the Skip strategy, subsequent jobs are skipped as long as the current instance has not completed. The strategy is illustrated in Figure 2.4. If the overrun covers several periods, many jobs may have to be skipped.

Compared to queuing, skipping can have the advantage of avoiding a domino effect of overruns due to a single long execution time. An obvious disadvantage

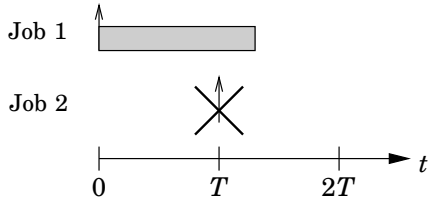

Figure 2.5 The Skip strategy. When the first job overruns, the second job is skipped.

of the skip strategy is that even a small overrun will cause the next job to be dropped—even if it might have a chance to complete in time.

Performance Analysis. For a given example, performance analysis of the different basic overrun strategies can be carried out using Jitterbug [Lincoln and Cervin, 2002]. Jitterbug is a MATLAB toolbox for control performance analysis in the presence of delays and jitter. The performance is measured by a quadratic cost function,

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

Here, x is a vector collecting all states and signals in the control system, and Q is a positive semidefinite weighting matrix.

For each case, a Markov chain describing the state of the scheduler is built. The Abort and Skip strategies are very simple to model (requiring only 2 Markov states), while the Queue(1) strategy is more complicated.

The analysis is exemplified on an integrator process,

$$G(s) = \frac{1}{s}.$$

The controller is designed to minimize the cost function

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

i.e., a minimum-variance controller [Åström and Wittenmark, 1997]. The sampling interval h and a computational delay of h is assumed in the design.

An execution-time distribution is assumed according to Figure 2.1, where $p = 0.8$, $C_{nom} = 1$, and $C_{max} = 2$. The performance for each strategy is computed for different sampling intervals between $h = 1$ and $h = 2$. Note that, in the case $h = 2$, no overruns occur and all strategies should behave the same.

The results are displayed in Figure 2.6, where the cost is plotted as a function of the sampling period. As the period is decreased from C_{max}, the cost of the Abort strategy initially decreases, but then increases again as the period approaches C_{nom}. With the Skip strategy, the results are reversed—the cost initially increases, but then decreases as the period becomes shorter. The Queue strategy increases the cost monotonically with shorter periods. The results in the different cases are the results of the interplay between the decreased sampling period (good for performance) and the cost of overruns (bad for performance).

In this example, the Skip strategy has the best performance, assuming that the period can be chosen freely. In some cases, however, the sampling period may be dictated by the application, and then the Abort strategy may give better performance.

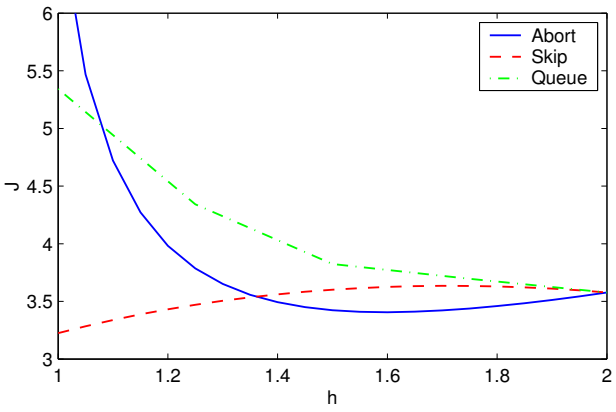

Figure 2.6 Comparison of costs in different basic overrun handling strategies.

2.3 Overrun Handling in the Control Server

The default overrun behavior in the control server is to queue any pending jobs in the case of overruns. As seen in the example in the previous section, this does not always give the best performance. It is, however, easy to extend the API of the control server to facilitate the Skip strategy. This is done by the introduction of a new primitive which indicates whether the task is late or not:

```
/* Returns 1 if the task is late, 0 otherwise */
int CS_task_late();
```

"Late" should here be interpreted as follows. When a task has an execution overrun, its deadline is postponed, and the task is marked as late. The task returns to the "not late" state when a job task arrives at time r and $d_i \leq r$.

Period and Segment Skipping

Using the `CS_task_late` primitive, the user can choose to skip a whole period or just a segment in the case of overruns. For instance, a control task may divided into one segment that does the control, while the other segment displays some graphics on the screen. In this case, the second segment can be skipped without any impact on the control performance. The principle is illustrated in the example below:

```
while (1) {
    // segment 1
    do_control();
    CS_task_endsegment();

    // segment 2
    if (!CS_task_late()) {
      draw_graphics();
    }
    CS_task_endsegment();
}
```

Formally, the segment is not really skipped, but it is replaced by an empty segment with a very short execution time. This scheme gives full flexibility to the programmer while keeping the internal implementation of the control server simple.

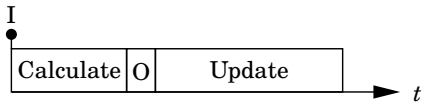

Figure 2.7 Handling late outputs using an extra output segment.

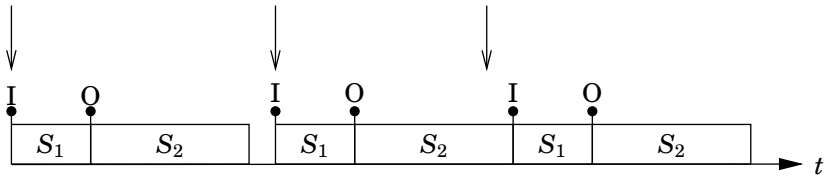

Figure 2.8 An aperiodically triggered control server task.

Alternative Handling of Late Outputs

In the control server, outputs are written at the end of a segment. This creates a constant input-output latency in a control task, as long as no overruns occur. The constant latency can be easily be compensated for in the controller design.

If an overrun occurs, however, the control signal is delayed a whole period. In this case, the fixed output point *increases* the output jitter rather than decreases it. Intuitively, a better solution would be to release the output "as soon as possible" if the output instance is missed. This can be achieved in the control server by introducing a dedicated "Output" segment, see Figure 2.7. Placing the output action in an extra segment will introduce a jitter of at most l_i^j when the task is on time.

2.4 Aperiodic Control Server Tasks

The control server was originally designed for periodic tasks only. Hence, it was assumed that all tasks would be periodically triggered by an internal timer in the kernel. There exist some applications, however, where the sampling is triggered by external events or hardware, and not by the kernel. An example of such a system is a frame grabber that delivers new frames in a buffer at a given approximate rate. The sampling process is in this case almost periodic, but may experience a small drift or release jitter compared with the real-time clock in the computer. To keep the task synchronized with the hardware, it is necessary to support aperiodically triggered tasks.

An aperiodic control server task is declared in the same way as a periodic task, but the period is interpreted as a minium-interarrival time instead. Should a new job arrive before the end of the last segment of the recent invocation, the new job is queued until the output action of the last segment has been executed, see Figure 2.8.

It is important to select an appropriate period for the aperiodic server. If the period is chosen longer than the minimum interarrival time of the task, the queue of jobs may grow indefinitely, and the task will loose synchronization with the arriving samples. As an example, consider a frame grabber that delivers images at a rate of about 24.9–25.1 frames per second. Too be on the safe side, the server period could be chosen as 39 ms (rather than the average value of 40 ms).

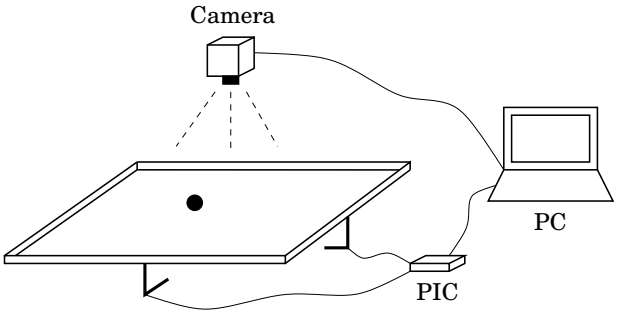

Figure 2.9 Schematic picture of the vision-based ball-and-plate control system.

2.5 A Control Application

As a testbench for the new control server scheduling module, a vision-based ball-and-plate application was used. Some system identification was performed, and an LQG controller with adaptive feedforward was designed. Finally, experiments with different overrun handling methods were performed.

The Ball and Plate Process

A schematic diagram of the ball-and-plate application is displayed in Figure 2.9. The object of the control is to make the ball follow a given reference trajectory on the plate. Feedback regarding the position of the ball is provided by a black-and-white video camera mounted above the plate. The plate can be tilted in the x and y directions by sending commands over a serial connection to a PIC microcontroller, which in turn gives commands to the servo motors mounted under the plate.

The process has been successfully controlled in previous projects by two PID controllers—one for each axis. The dynamics in each direction are essentially described by a double integrator. Hence, a PD controller should be sufficient to stabilize the process. Due to friction, the ball can get stuck close to the setpoint. Therefore, it is also good to add integral action in the controller.

The Vision System

The camera is connected to a frame grabber in the PC, which produces 25 pictures per second. The image size is 320 times 200 pixels, and each pixel has a value between 0 (black) and 255 (white). To simplify the image processing, only a rectangle of 170 times 150 pixels in the middle of the plate (away from the borders) was used.

To locate the ball on the plate, a simple center-of-mass calculation can be performed. First, the image is thresholded to separate the dark ball from the light background. Then, the center of the dark area is computed. The size of the ball used was about 65 pixels.

To speed up the image processing, a smaller search window can be used. If the location of the ball in the previous image (40 ms ago) is known, it is likely that the ball remains close by in this image. It was found that a window of 30 times 30 pixels around the previous position was sufficient never to lose the ball during normal operation. The execution time for a complete scan of the plate was about 5 ms, while the smaller window could be searched in about 1.4 ms.

Controller Design

Although the process had previously successfully been controlled using PID controllers, some system identification experiments were performed to see if there was additional dynamics. Further dynamics could possibly be introduced by the

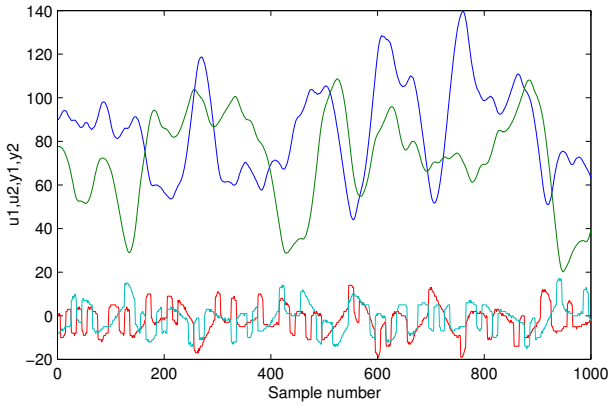

Figure 2.10 Some of the data from the system identification experiment. On the bottom are the two control inputs; on the top are the two measurement outputs.

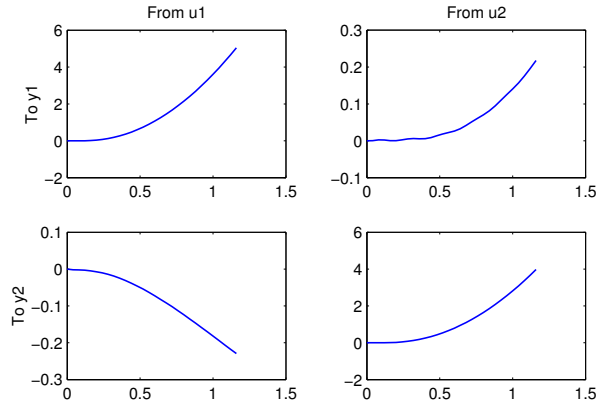

Figure 2.11 Step responses of the identified MIMO model.

servo motors and the time delays present in the serial communication and the frame grabber.

System Identification. Since the process is open-loop unstable, the system identification was performed in closed loop. Two detuned PID controllers were used to barely stabilize the process, and random disturbances were added to the control signal. The data from such an experiment is shown in Figure 2.10. The data was analyzed using the System Identification Toolbox in MATLAB. An 6th order multiple-input multiple-output (MIMO) state-space model was deemed sufficient to capture the dynamics. The poles of the model could be interpreted as a double integrator and a low-pass filter in each dimension. The time delay was found to be (approximately) one sample (i.e., 40 ms). The step responses of the 6th order MIMO model are shown in Figure 2.11. It is seen that the two dimensions have slightly different gains, and that there is some cross-coupling between the axis. This was probably the result of the board being slightly rotated a few degrees compared to the camera.

Control Design. Based on the 6th order MIMO model of the process, a MIMO LQG controller [Åström and Wittenmark, 1997] was designed using the Control Systems Toolbox in MATLAB. The controller was then augmented with an additional integrator for each output to eliminate steady-state errors. The gain of the feedback from the integrator states and the feedforward from the reference to the

control signal were handtuned to achieve good tracking performance.

To achieve integral action, the model was augmented with an additional integrator on each output. The resulting controller was of 8th order. The complete controller (Kalman filter, state feedback, integrators and reference feedforward) can be written as a standard linear filter on the form

$$\begin{aligned} x(k+1) &= Ax(k) + By(k) + B_r r(k) \\ u(k) &= Cx(k) + Dy(k) + D_r r(k) \end{aligned} \tag{2.2}$$

where y is the measurement signals, r is the reference signals, x is the controller state vector, and u is the control signals. The coefficients of the matrices A, B, B_r, C, D, and D_r were exported in a C file for use in the real-time control program.

Experiments

In the experiments, we consider a heavily loaded system, where there is not CPU resources to do a complete scan of the plate image in every sample. The control task consists of two segments. In the first segment, the image processing and control takes place. If the ball was located in the previous period, then a small search window around that position is used in this period. If the ball is not located in the window, the entire plate must be searched. In the case of overrun, the second segment may be skipped. The pseudo-code of the control task is shown below:

```
while (1) {
  // segment 1: locate ball, do control
  if (mode == 1) {
    search_small_window();
    if (!found) {
      mode = 0;
    }
  if (mode == 0) {
    search_entire_plate();
    if (found) {
      mode = 1;
    }
  }
  if (mode == 1) {
    do_control();
  }
  CS_segment_end();

  // segment 2: draw graphics
  if (!CS_task_late()) {
    draw_ball_and_plate();
  }
  CS_segment_end();
}
```

An aperiodic task was used, where the task is triggered by the arrival a new image from the frame grabber. In the input hook of the first segment, a request for a new image acquisition is sent to the frame grabber. This way, the task is constantly synchronized with the frame grabber and the input-output latency from image to control is minimized.

From measurements it was found that the execution times of the local window scan plus control, the full scan plus control, and the graphics display were 1.4 ms, 5.0 ms, and 9.3 ms respectively. Allocating $U = 0.27$ of the processor for the control task, segment lengths were chosen as $l^1 = 1.4/0.22 = 5.2$ ms and $l^2 = 9.3/0.27 = 34$ ms. This means that a full scan will cause an execution overrun.

The performance of the overrun strategies of Queue and Skip was compared in control experiments were the view of the ball was blocked for a few frames. This caused the global scan to be invoked, causing repeated execution overruns in the control task. The performance of the controller was measured by a quadratic cost function, which was integrated over dozens of repeated experiments. In the end, no significant difference in control performance between the two strategies could be observed.

In this application, the consequence of queuing a few jobs is simply that the sampling period will be slightly longer for a few periods. This may not have a very great impact on the performance if the sampling rate is high compared to the bandwidth of the closed-loop system.

2.6 Conclusion

Overrun handling can be an issue for control tasks with varying execution times. Examples of such tasks are optimization-based control algorithms (for instance model-predictive control) or vision-based algorithms with dynamically-sized search windows. Assuming reservation-based scheduling (approximating the case of a single task on a single processor), three basic overrun strategies have been identified: Abort, Queue, and Skip. For simple examples, the performance of each strategy may be computed analytically, given linear dynamical models of the controller and the plant and an execution-time distribution. Calculations on an integrator process showed that the Skip strategy had the best performance for that particular example.

The Queue and Skip strategies have been implemented in the control server scheduling module in the S.Ha.R.K. real-time operating system, and their performance have been compared in a ball-and-plate control application. The results from these experiments showed that Queue and Skip strategies performed equally well.

3. The Jitter Margin

3.1 Background and Motivation

In classical feedback control theory (e.g., [Franklin *et al.*, 2002]), notions such as *phase margin* and *gain margin* are used to describe how sensitive a control loop is towards various uncertainties in the plant. Nonnegative margins are required to ensure the stability of the closed-loop system. The margins are also used as practical stability measures, and there are various rules of thumb associated with them. For instance, it is typically recommended to have a phase margin of at least 30°–45° to ensure some degree of robustness and performance of the system.

When a controller is implemented as a task in a real-time system, a new kind of uncertainty is introduced—an *implementation uncertainty*. Here, we will focus on the specific problem of *output jitter*. Variability in the task execution time and preemption from other tasks can cause the controller to experience a different amount of input-output delay in each period. It is well known that such a jitter can degrade the control performance and in extreme cases even cause instability of the control loop (e.g., [Törngren, 1998]). Although the present work only considers jitter due to CPU scheduling, some of the results also carry over to networked control systems, where jitter due to variable transmission times is a major issue.

The majority of previous work on jitter in real-time control systems has focused on either scheduling theory or control theory. In the few instances where an integrated approach has been taken, the control analysis has been somewhat underdeveloped. By contrast, our analysis yields hard results and should hence be applicable to a wide range of systems, including safety-critical applications.

Recently, a new stability theorem for control loops with time-varying input-output delays has been developed [Kao and Lincoln, 2004]. Based on this theorem, we propose the notion of *jitter margin* for control tasks. The jitter margin can be combined with real-time scheduling theory to guarantee the stability and performance of the controller in the target system. The jitter margin can also be used as a tool for assigning meaningful deadlines to control tasks.

It is noted that the jitter analysis can be improved if *best-case response times*, as well as worst-case response times, can be computed. For this purpose, we propose a lower bound on the best-case response time under EDF scheduling, where no such results are known to exist.

When designing a real-time control system, information about the task timing is needed in the control design, and information about the controller timing sensitivity is needed in the real-time design. Based on this insight, we propose an iterative control–scheduling codesign procedure, where the jitter margin is used as a central tool.

Related Work

Several works have considered scheduling solutions to reduce output jitter in general. In [Locke, 1992] and [Klein *et al.*, 1993], it is suggested to use dedicated high-priority output tasks to reduce the jitter. This has the disadvantages of a more complex implementation and longer delays on average. [David *et al.*, 2001] considers jitter reduction under deadline-monotonic and EDF scheduling. Output jitter reduction under EDF is also the topic of [Baruah *et al.*, 1999] and [Kim *et al.*, 2000]. It can be noted that, in these papers, the jitter is defined between successive periods, rather than over the lifetime of the system (as in this work).

There have also been some efforts to specifically minimize jitter in control tasks. The papers [Crespo *et al.*, 1999; Balbastre *et al.*, 2000] define the *control action interval*, which is just another term for output jitter. The proposed solution introduces high-priority tasks for the input and output actions. Again, this has the disadvantage of longer delays on average. Also, the resulting control performance is not analyzed. [Cervin, 1999] proposes a subtask scheduling method for control tasks, where the main part of the control algorithm are scheduled at different priorities. The scheme attempts to reduce both the delay and the jitter. The performance improvements are verified by simulations.

Jitter compensation in control has been the subject of much research. In [Nilsson, 1998], an optimal jitter-compensating LQG controller is derived in the context of networked control loops. The controller uses timestamps to track the sensor-to-controller and controller-to-actuator delays. The performance is measured by a quadratic cost function and is evaluated by stochastic analysis. [Marti *et al.*, 2001] considers jitter compensation in state feedback controllers. No specified scheduling algorithm is considered, but it is assumed that the delays are known a-priori. Also, full state information is assumed. The performance improvements are verified by simulations. In [Lincoln, 2002] a more realistic approach is taken, where the output jitter experienced in one period is compensated for in the next period. The resulting jitter-compensating controller can be viewed as a generalization of the well-known Smith predictor.

In the area of control–scheduling codesign, [Shin *et al.*, 1985] studies computational delays in computer-controlled systems. Hard constraints on the controlled variables (e.g., physical constraints) are used to derive maximum allowable control latencies in different regions of the state space. It is noted that the hard deadline may be a random variable due to stochastic disturbances acting on the process. The approach is extended in [Shin and Kim, 1992] where the stability of the closed-loop system is also considered. Sampling period selection for control tasks is the topic of [Seto *et al.*, 1996]. The performance of the control loops are described using cost functions, and the period assignment problem is formulated as an optimization problem. The combined effect of period and delay on control performance is studied in [Ryu *et al.*, 1997], where simulations are used to evaluate the performance. None of these papers considers jitter, however.

3.2 The Jitter Margin

Preliminaries

Computer-controlled systems (e.g., [Åström and Wittenmark, 1997]) are typically designed assuming periodic sampling and either zero or a constant computational delay. A real implementation, however, will introduce jitter at various points in the control loop.

Here, for analysis purposes, we will assume that the sampling is jitter-free, while the input-output delay may be time-varying. Jitter-free sampling can be achieved by programming the A-D converter to take samples periodically, or by requesting the A-D conversion when the control task is released.

The assumed control loop is shown in Figure 3.1. The plant is described by the linear continuous-time system $P(s)$, and the plant output is sampled with the constant interval h. The controller is described by the linear discrete-time system $K(z)$. Following the zero-order hold, there is a time-varying delay Δ before the control signal is applied to the input of the plant.

Exact stability analysis of the closed-loop system is trivial if the delay Δ is either constant or varying according to a known, periodic pattern. If the delay

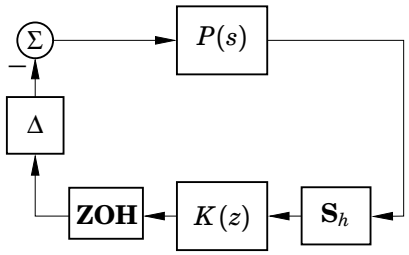

Figure 3.1 Computer-controlled system with continuous-time plant $P(s)$, periodic sampler $\mathbf{S}_h$, discrete-time controller $K(z)$, zero-order hold, and time-varying delay Δ.

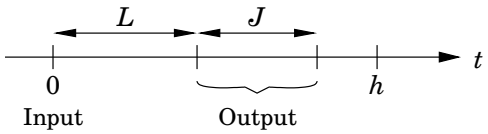

Figure 3.2 The input-output delay can be divided into a constant delay, L, and a jitter, J.

varies randomly among a set of known delays, Lyapunov theory can be used to verify the stability of the closed-loop system. For freely time-varying delays, the analysis is considerably more difficult. The following theorem from [Kao and Lincoln, 2004] is only sufficient, but it guarantees stability for *any* delays in a given interval, including constant, periodic, and random delays:

THEOREM 3.1—STABILITY UNDER OUTPUT JITTER
The closed-loop system in Figure 3.1 is stable for any time-varying delays $\Delta \in [0, Nh]$, where $N > 0$ is a real number, if

$$\left| \frac{P_{\text{alias}}(\omega)K(e^{i\omega})}{1 + P_{\text{ZOH}}(e^{i\omega})K(e^{i\omega})} \right| < \frac{1}{\tilde{N}\left|e^{i\omega} - 1\right|}, \quad \forall \omega \in [0, \pi], \tag{3.1}$$

where $\tilde{N} = \sqrt{\lfloor N \rfloor^2 + 2\lfloor N \rfloor g + g}$ and $g = N - \lfloor N \rfloor$; $P_{\text{ZOH}}(z)$ is the zero-order hold discretization of $P(s)$, and

$$P_{\text{alias}}(\omega) = \sqrt{\sum_{k=-\infty}^{\infty} \left| P\left(i(\omega + 2\pi k)\frac{1}{h} \right) \right|^2}. \tag{3.2}$$

Proof. See [Kao and Lincoln, 2004]. □

Definitions and Properties

We now consider a periodic control task with the period $T = h$, executing in a real-time system. The plant is assumed to be sampled when the task is released, and the control signal is actuated when the task finishes.

The input-output delay experienced by the controller can be divided into two parts: a constant part, $L \geq 0$, and a time-varying part (the jitter), $J \geq 0$, see Figure 3.2. The minimum possible delay is hence given by L, and the maximum possible delay is given by $L + J$.

We will first recall the definition of the classical *delay margin* for the jitter-free case ($J = 0$):

DEFINITION 3.1—DELAY MARGIN
Given the system in Figure 3.1, the delay margin is defined as the largest number L_m for which closed-loop stability is guaranteed assuming a *constant delay* $\Delta = L_m$. □

Remark. For continuous-time control systems, the delay margin can be computed as

$$L_m = \varphi_m / \omega_c, \tag{3.3}$$

where φ_m is the *phase margin* and ω_c is the *crossover frequency* of the system. Due to aliasing effects, the exact computation is more complicated for computer-controlled systems (see [Åström and Wittenmark, 1997]). □

In systems with jitter, the delay and the jitter will both contribute to the destabilization of the system. Hence, we give the following definition of the *jitter margin*:

DEFINITION 3.2—JITTER MARGIN
Given the system in Figure 3.1, the jitter margin is defined as the largest number $J_m(L)$ for which closed-loop stability is guaranteed for any time-varying delay $\Delta \in [L,\, L + J_m(L)]$. □

Remark. Since Theorem 1 is only sufficient, it can only be used to compute a lower bound on the jitter margin. The theorem is not very conservative, however. To apply the theorem, we replace the plant $P(s)$ by its time-delayed version $P(s)e^{-sL}$ and let $N = J/h$. □

The reason for defining the jitter margin as a function of L is to make the stability test less conservative whenever a lower bound on L is available. It is obvious that, if a system is stable for any time-varying delay $\Delta \in [0,\, J]$, it must also be stable for any time-varying delay $\Delta \in [L,\, J]$, $0 < L \leq J$. Furthermore, in the latter case, the system might also be stable for longer delays. Based on this argument, the following properties of the jitter margin can be derived (the proofs are omitted):

PROPERTY 3.1
$J_m(L) = 0, \quad L \geq L_m$. □

PROPERTY 3.2
$J_m(L) \leq L_m, \quad \forall L$. □

PROPERTY 3.3
$J_m(L) + L$ is an increasing function of L. □

EXAMPLE 3.1—JITTER MARGIN
Figure 3.3 reports the jitter margin as computed by Theorem 1 for the plant $P(s) = 1000/(s(s+1))$ and two different controllers. Both controllers are designed with the sampling interval $h = 10$ [ms]. In (a), a PID controller is used. The delay margin is $L_m = 7.8$, and the jitter margin has the maximum value $J_m(0) = 3.7$. In (b), an LQG controller designed for a constant delay $L = 5$ is used. Here, the delay margin is $L_m = 15.5$, and the jitter margin has the maximum value $J_m(4.8) = 7.1$. It can be seen that the jitter-margin function can have different shapes for different controllers, but the maximum total delay, $J_m(L)+L$, is always an increasing function. □

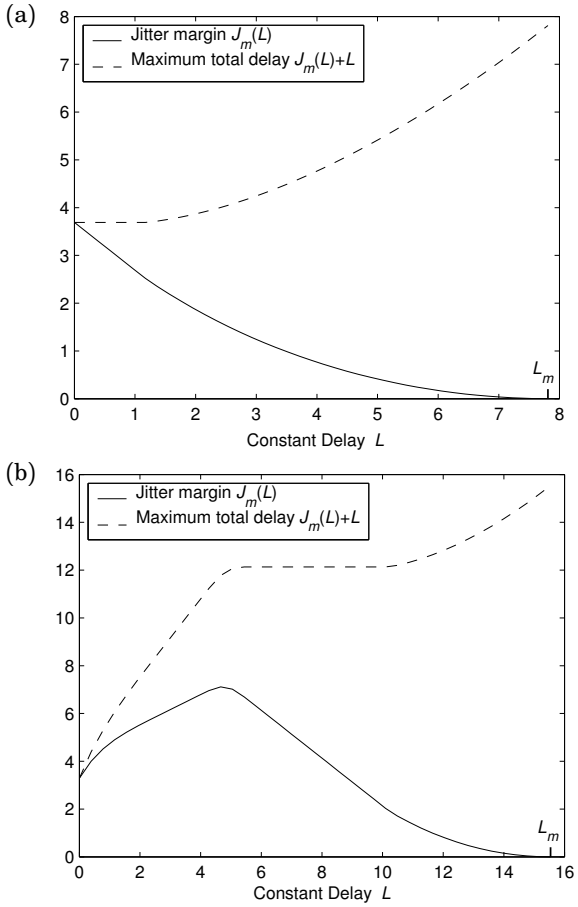

Figure 3.3 Example of jitter margins $J_m(L)$: (a) PID controller with $h = 10$, (b) LQG controller with $h = 10$, designed for the delay $L = 5$. (All units are in ms.)

Verifying Stability and Performance

If we know the constant delay L and the jitter J of a control task, stability of the closed-loop system is guaranteed if

$$J_m(L) > J. \tag{3.4}$$

Often, it is not enough to just guarantee stability—there must also be some margins that guarantee performance. In classical control theory, the phase margin is sometimes used as a performance and robustness measure. Unfortunately, the phase margin is only defined for systems without jitter. It is, however, possible to generalize the concept via an extended definition of the delay margin. Hence, we start by defining a delay margin for systems with delay and jitter:

DEFINITION 3.3—DELAY MARGIN FOR SYSTEMS WITH DELAY AND JITTER
Given the system in Figure 3.1, assuming some constant delay L and jitter J, the delay margin is defined as the largest number L_m for which closed-loop stability is guaranteed for any time-varying delay $\Delta \in [L + L_m, L + L_m + J]$. □

Remark. For systems without jitter, this definition is equivalent to Definition 1. □

Expressed in terms of the jitter-margin function $J_m(L)$, the delay margin is given

by the smallest L_m that solves

$$J_m(L + L_m) = J. \tag{3.5}$$

For the control designer, it is often more convenient to think in terms of phase margin, since that measure is independent of time. For systems without jitter, the relationship between phase margin and delay margin is approximately given by (3.3). Based on this observation, we propose the notion of *apparent phase margin:*

DEFINITION 3.4—APPARENT PHASE MARGIN
Given the system in Figure 3.1, assuming the constant delay L and the jitter J, the apparent phase margin is defined as the largest number $\hat{\varphi}_m$ for which closed-loop stability is guaranteed for any time-varying delay $\Delta \in [L+\hat{\varphi}_m/\omega_c,\, L+\hat{\varphi}_m/\omega_c+J]$, where ω_c is the crossover frequency of the system if assuming only the constant delay L. □

Similar to above, expressed in terms of the jitter-margin function $J_m(L)$, the apparent phase margin is given by the smallest $\hat{\varphi}_m$ that solves

$$J_m(L + \hat{\varphi}_m/\omega_c) = J. \tag{3.6}$$

A system with the apparent phase margin $\hat{\varphi}_m \leq 0^\circ$ can be interpreted as a system for which stability cannot be guaranteed, while any $\hat{\varphi}_m > 0^\circ$ can be interpreted as a performance guarantee. For systems without jitter, the apparent phase margin is equal to the classical phase margin.

Deadline Assignment

In the real-time literature, task deadlines are often considered as given parameters. Using the jitter margin, we can derive *real* hard deadlines that guarantee closed-loop stability. For instance, given that we have a lower bound on the constant delay L in the target system, we can guarantee stability by assigning the relative deadline

$$D = L + J_m(L). \tag{3.7}$$

(It is of course also required that all deadlines are really met during run-time.) Note that, if no estimate of L is available, assuming $L = 0$ yields a more conservative deadline.

Similarly, we can assign deadlines that guarantee a certain apparent phase margin in the target system. Given a lower bound on the constant delay L in the target system and a desirable apparent phase margin $\hat{\varphi}_m < \omega_c(L_m - L)$, we can guarantee a level of performance by assigning the deadline

$$D = L + J_m(L + \hat{\varphi}_m/\omega_c). \tag{3.8}$$

EXAMPLE 3.2—DEADLINE ASSIGNMENT
Consider the LQG controller in Example 1, whose jitter margin is shown in Figure 3(b). Without jitter, assuming $L = 5$, the phase margin is $\varphi_m = 34.9^\circ$ and the crossover frequency is $\omega_c = 57.9$ rad/s. Suppose that we require an apparent phase margin of $\hat{\varphi}_m = 20^\circ$. The allowable jitter is then given by

$$J_m(5 + 20^\circ/57.9 \text{ rad}) = J_m(11.0) = 1.4,$$

and we should hence assign the relative deadline

$$D = L + J_m(11.0) = 6.4.$$

□

An interesting problem here is that, depending on the scheduling policy, the constant delay might depend on the deadline which we are trying to compute. For instance, under deadline-monotonic scheduling, the assigned deadline will affect the priority of the task, which might in turn affect the constant delay. The problem could possibly be addressed using an iterative deadline assignment procedure, but this is left as future work.

3.3 Review of Response-Time Analysis

In order to apply the stability and performance analysis of the previous section, we need to be able to compute the constant delay and the jitter for each control task in the system. This can be done using response-time analysis. Let R_i and R_i^b denote, respectively, the worst-case and best-case response times of task i. The constant delay L_i and the jitter J_i are then given by

$$L_i = R_i^b, \tag{3.9}$$

$$J_i = R_i - R_i^b. \tag{3.10}$$

Often, the true values of R_i and R_i^b cannot be obtained. First, if the task phasing is unknown, one must assume *worst-case phasing* when computing R_i and *best-case phasing* when computing R_i^b. It is not certain that R_i and R_i^b can *both* occur during the lifetime of the system. Second, depending on the scheduling policy and the task set, exact analysis for the worst-case and the best-case response times may not be available.

From a stability perspective, it is always safe to overestimate R_i and to underestimate R_i^b. This will make L_i smaller and J_i larger, causing the apparent phase margin to decrease.

Below, a brief outline of the available results in response-time analysis under fixed-priority and EDF scheduling is given. For EDF, a new lower bound on best-case response times is proposed.

Worst-Case Response Time Analysis

Under fixed-priority scheduling, assuming $D_i \leq T_i$, the worst-case response time of task i is given by the well-known equation [Joseph and Pandya, 1986]

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{3.11}$$

Exact analysis also exists for task sets with release offsets as well as deadlines $D > T$ [Audsley *et al.*, 1993; Tindell *et al.*, 1994].

Under EDF scheduling, worst-case response-time analysis is more complicated. Assuming $D_i \leq T_i$, the worst-case response time of task i is given by [Spuri, 1996; George *et al.*, 1996; Stankovic *et al.*, 1998]

$$R_i = \max\Big\{ C_i, \max_{a \geq 0} \{ L_i(a) - a \} \Big\}, \tag{3.12}$$

where the busy interval $L_i(a)$ is given by the equation

$$L_i(a) = W_i\big(a, L_i(a)\big) + \left(1 + \left\lfloor \frac{a}{T_i} \right\rfloor \right) C_i, \tag{3.13}$$

and the higher-priority workload $W_i(a,t)$ is given by

$$W_i(a,t) = \sum_{j \neq i,\, D_j \leq a+D_i} \min \left\{ \left\lceil \frac{t}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor \right\} C_j. \tag{3.14}$$

It should be noted that only a finite number of values of a must be checked when evaluating (3.12). The analysis has also been generalized to arbitrary deadlines [George *et al.*, 1996].

Best-Case Response Time Analysis

Under fixed-priority scheduling, exact best-case analysis has recently been developed for the case $D \leq T$ [Redell and Sanfridson, 2002]. The best-case response time of task i is given by the equation

$$R_i^b = C_i^b + \sum_{j \in hp(i)} \left\lceil \frac{R_i^b}{T_j} - 1 \right\rceil C_j^b, \tag{3.15}$$

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

Under EDF scheduling, no exact best-case analysis is known to exist. In the next section, a lower bound on the the best-case response time under EDF scheduling is developed.

3.4 Best-Case Response-Time Analysis Under EDF

We will consider a set of periodic tasks scheduled under EDF. Each task i has a period T_i, a relative deadline $D_i \leq T_i$, and a best-case execution time C_i^b. It is assumed that the task set is schedulable.

A trivial lower bound $\underline{R}_i$ on the best-case response time of task i is given by

$$\underline{R}_i^b = C_i^b. \tag{3.16}$$

This is actually a quite good bound for the shortest-period tasks. The longest-period tasks can, however, have much longer best-case response times, especially if the system load is high.

A tighter lower bound on the best-case response time can be obtained by interference analysis. Let R_i be the response time of an instance of task i that is released at time 0, and let task j be a potentially interfering task. We will construct a lower bound on R_i by shifting each task j such that minimum interference is obtained.

First, consider a task j with $D_j \geq R_i$. It is obvious that the task can be phased such that it does not interfere with task i.

For each task j with $D_j < R_i$ we must consider two different cases, see Figure 3.4. In case (a), $D_i - D_j > R_i$, and each instance of task j released within the interval $[0, R_i]$ will have higher priority than task i. Minimum interference is obtained when task j is phased such that one release occurs at time R_i. The number of complete preemptions from task j is hence given by $\lceil R_i/T_j - 1 \rceil$.

In case (b), $D_i - D_j \leq R_i$, and instances of task j will only have higher priority if released within the interval $[0, D_i - D_j]$. Minimum interference is obtained when task j is phased such that one release occurs at time $D_i - D_j$. The number of complete preemptions from task j is hence given by $\lceil (D_i - D_j)/T_j - 1 \rceil$.

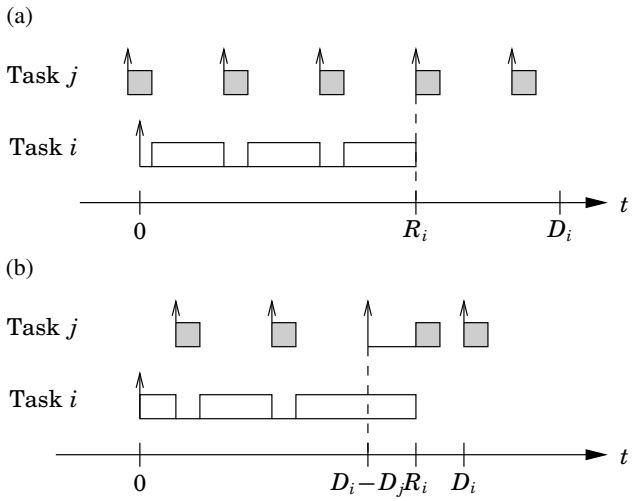

Figure 3.4 Different cases where task j causes minimum interference for task i: (a) $D_i - D_j > R_i$, (b) $D_i - D_j \leq R_i$.

Each complete preemption from task j will contribute C_j to the response time. Combining the two cases above, a lower bound, $\underline{R}_i^b$, on the minimum response time of task i is given by

$$\underline{R}_i^b = C_i^b + \sum_{\forall j: D_j < \underline{R}_i^b} \left\lceil \frac{\min\left\{\underline{R}_i^b,\, D_i - D_j\right\}}{T_j} - 1 \right\rceil C_j^b \tag{3.17}$$

This expression provides only a lower bound, since it does not take any initial (partial) interference from task j into account (see for instance Figure 3.4(a)). It is possible to improve the formula slightly by including some obvious cases where initial interference must occur. It is conjectured, however, that the expression for the *exact* best-case response time is as complex as the formula for exact worst-case response time under EDF.

The results obtained with the proposed bound have been compared to results obtained by simulation, where the shortest response time of each task was recorded. (Note that the latter constitutes an *upper bound* on the real best-case response time.) The bounds were evaluated for loads ranging from $U = 0.5$ to $U = 0.99$. For each load case, 100 random task sets were generated. The number of tasks in each set was integer-uniformly distributed between 2 and 10. The task periods were exponentially distributed with mean 1, and the fraction of the execution time to the period was uniformly distributed between 0 and 1. The execution times were uniformly rescaled to give the task set the desired utilization. Throughout, $D_i = T_i$ and $C_i^b = C_i$ were assumed.

For each task set, the system was simulated for 1000 s, and the minimum response time of the longest-period task (task n) was recorded. The result was compared with the bounds (3.16) and (3.17). Figure 3.5 shows the mean of R_n^b/C_n over the task sets for different bounds and different loads. It is seen that the proposed bound performs quite well up to a load of $U = 0.95$. The bound is not tight since it does not consider initial interference.

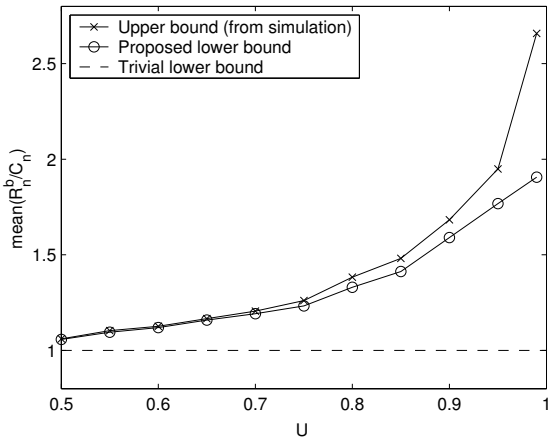

Figure 3.5 Comparison of bounds on the best-case response time under EDF. The results are shown for the longest-period task.

3.5 A Codesign Procedure

To illustrate how the jitter margin could be applied in the design of real-time control systems, we describe an iterative control–scheduling codesign procedure.

It is assumed that a set of independent controllers should be implemented in the same processor. The controllers are designed in continuous time, and should be discretized and implemented as periodic tasks with different periods. The goal of the codesign procedure is to choose sampling periods such that the controllers will experience the same relative performance degradation in the target system, taking the jitter into account. The performance of the continuous-time controller is measured by its original phase margin φ_m, and the performance of the control task is measured by its apparent phase margin $\hat{\varphi}_m$ (see Section 2.3). The goal of the procedure is to make the ratio $\hat{\varphi}_m/\varphi_m$ as equal as possible among the tasks.

The inputs to the codesign procedure are a set of n continuous-time plants, $P(s)$, a set of n continuous-time controllers, $K(s)$, estimates of the best-case and worst-case execution times of the control algorithms, C and C^b, and a scheduling policy where worst-case as well as best-case response time analysis is available.

The procedure is outlined is the following steps:

1. Initialize by assigning initial (nominal) sampling periods h for the controllers. (A common rule of thumb [Åström and Wittenmark, 1997] is to choose the sampling period such that $\omega_b h \in [0.2, 0.6]$, where ω_b is the bandwidth of the closed-loop continuous system.)

2. Rescale the periods linearly such that the task set becomes schedulable under the given scheduling policy. (Here, a suitable sufficient schedulability test can be used.)

3. Discretize the controllers using the assigned sampling periods, yielding the set of discrete-time controllers $K(z)$.

4. For each task, compute worst-case and best-case response times, R and R^b. (Here, the analysis in Section 3 is applicable.)

5. For each task, compute the jitter margin using Theorem 1 and the apparent phase margin $\hat{\varphi_{m}}_i$ from (3.6), assuming the constant delay $L_i = R_i^b$ and the jitter $J_i = R_i - R_i^b$.

6. For each task, compute the relative performance degradation $r_i = \hat{\varphi_{m_i}}/\varphi_{m_i}$. Also, compute their mean value, $\bar{r} = \sum r_i/n$.

7. For each task, adjust the period according to

$$h_i := h_i + kh_i(r_i - \bar{r})/\bar{r},$$

where $k < 1$ is a gain parameter.

8. Repeat from 2 until no further improvement is given. A suitable stop criterion is when sum of the performance differences, $\sum |r_i - \bar{r}|$, is no longer decreasing.

The period adjustment mechanism in step 7 is intended to decrease the periods of controllers with bad performance, and to increase the periods of controllers with good performance. Choosing the gain parameter can be difficult. A small k will give slow adaptation, while a large k can cause instability.

The iterative procedure tries to solve a highly nonlinear optimization problem. Hence, it is not certain that it will converge to an optimal solution. For instance, under rate-monotonic scheduling, a small period adjustment may change the task priorities, and this can in turn have a huge impact on the jitter. Neither is it certain that a completely equal performance degradation can be achieved.

Example 3.3—Codesign

We consider an example where three controllers should be implemented in a single CPU. Both rate-monotonic and EDF scheduling is considered. The execution times of the control algorithms are assumed to be equal and constant and are given by $R = R^b = 0.15$ [ms]. The plants to be controlled are given by

$$\begin{aligned} P_1(s) &= \frac{8\cdot 10^5}{s(s+1000)}, \\ P_2(s) &= \frac{4\cdot 10^4}{(s-200)(s+200)}, \\ P_3(s) &= \frac{5\cdot 10^7}{s(s^2+100s+2.5\cdot 10^5)}, \end{aligned} \tag{3.18}$$

and the continuous-time controllers are given by

$$\begin{aligned} K_1(s) &= \frac{4.88\cdot 10^3(s+2\cdot 10^5)(s+1295)}{(s+5000)(s^2+7.325\cdot 10^4 s+2.573\cdot 10^9)}, \\ K_2(s) &= \frac{2.57\cdot 10^3(s+2\cdot 10^5)(s+259.1)}{(s+3000)(s^2+1.645\cdot 10^4 s+1.35\cdot 10^8)}, \\ K_3(s) &= \frac{478(s+2\cdot 10^5)(s^2+160.6s+1.655\cdot 10^5)}{(s+2740)(s+1000)(s^2+2494s+7.109\cdot 10^6)}. \end{aligned} \tag{3.19}$$

Table 3.1 reports the bandwidth ω_b and the original phase margin φ_m of each control loop. It is seen that the loops have different bandwidths, which suggests that the controllers would require different sampling intervals. The differences in bandwidth are also visible in Figure 3.6, which shows the system responses for the different continuous-time loops.

To initialize the procedure, nominal sampling periods are chosen by the rule of thumb $\omega_b h = 0.2$. This results in a CPU utilization of $U = 1.30$. Hence, slower sampling must be used in the target system. For the controller discretization, the Tustin method is used.

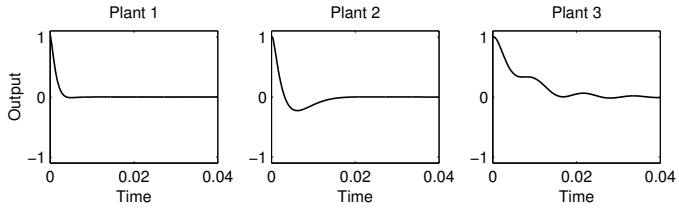

Figure 3.6 System responses of the original continuous-time control loops.

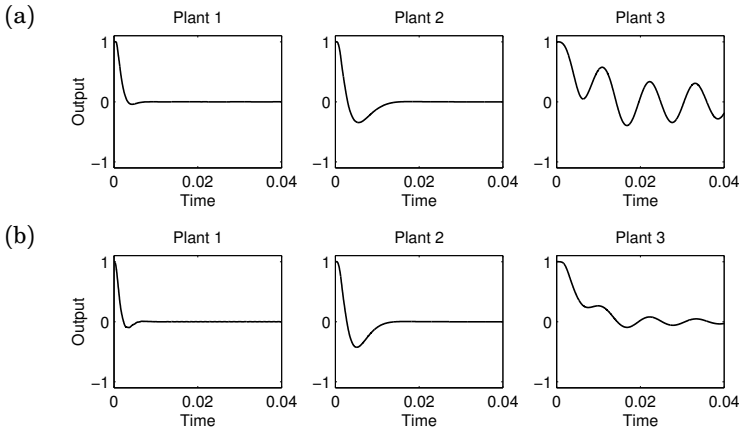

Figure 3.7 Control system responses under rate-monotonic scheduling: (a) after one iteration, (b) after ten iterations.

First, rate-monotonic scheduling is assumed. The target utilization is chosen as $U = 0.78$. The adaptation gain is chosen as $k = 0.2$. The results of the codesign procedure after one and ten iterations are shown in Table 3.2. After the initial iteration, where the nominal sampling periods have been simply rescaled, loop 3 has a small negative apparent phase margin. That means that stability cannot be guaranteed for that loop. After ten iterations, the periods have been adjusted such that they are nearly equal, resulting in a somewhat more equal performance degradation (as measured by the ratio $\hat{\varphi}_m/\varphi_m$).

To verify the results of the procedure, the complete real-time system (including plants, controllers, and scheduler) was also simulated using the MATLAB/Simulink toolbox TrueTime [Henriksson *et al.*, 2002]. The actual control system responses after one and ten design iterations are shown in Figure 3.7. It is seen that, after one iteration, loop 3 is close to unstable, as predicted by the negative apparent phase margin. After ten iterations, the performance degradation of loop 3 is visibly smaller.

Next EDF scheduling is assumed. The target utilization is chosen as $U = 0.95$. The results of the codesign procedure after one and ten iterations are shown in Table 3.3. After the initial iteration, task 3 has a large negative apparent phase

Table 3.1 Bandwidths and phase margins of the original continuous-time control loops

Loop	ω_b	φ_m
$P_1(s), K_1(s)$	960 rad/s	74.1°
$P_2(s), K_2(s)$	599 rad/s	49.5°
$P_3(s), K_3(s)$	179 rad/s	69.7°

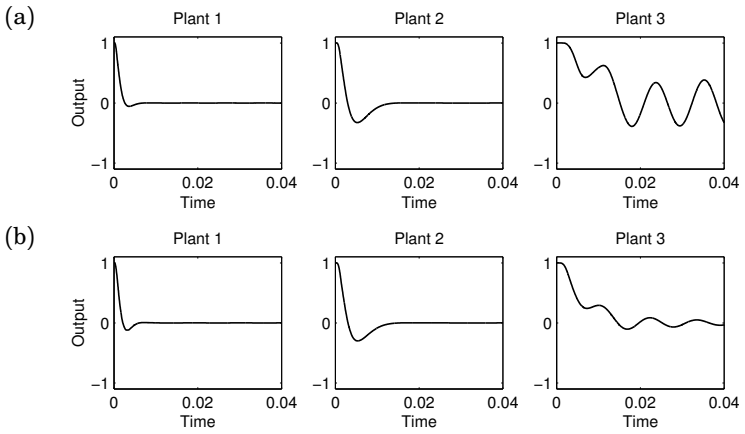

Figure 3.8 Control system responses under EDF scheduling: (a) after one iteration, (b) after ten iterations.

margin, implying that the control loop might be unstable. After ten iterations, the performance degradation is quite even among the controllers. Again, the results were also verified in simulations. Figure 3.8 shows the system response after one and ten iterations.

The final design results under rate-monotonic scheduling and EDF scheduling are quite similar. Under EDF, slightly shorter periods could be used, due to the higher level of schedulability of EDF. It can also be noted that, under EDF, the jitter is more evenly distributed among the tasks. This makes it possible to achieve a more even performance degradation among the control loops. □

3.6 Conclusion

This chapter has proposed the notion of *jitter margin* and showed how it can be applied in the design of real-time control systems. The stability test is based on worst-case assumptions about the jitter, and hence produces hard stability results. We have also linked the control analysis to scheduling analysis, showing how output jitter analysis can be used together with the jitter margin. An extensive codesign example has been presented, where many of the proposed concepts have been applied.

Table 3.2 Codesign results under rate-monotonic scheduling: (a) after one iteration, (b) after ten iterations.

(a)

Task	h	R	R^b	J	$J_m(R^b)$	$\hat{\varphi}_m$	$\hat{\varphi}_m/\varphi_m$
1	0.35	0.15	0.15	0	1.08	60.8°	0.82
2	0.56	0.30	0.15	0.15	1.17	27.9°	0.56
3	1.87	0.90	0.15	0.75	0.47	−4.8°	−0.07

(b)

Task	h	R	R^b	J	$J_m(R^b)$	$\hat{\varphi}_m$	$\hat{\varphi}_m/\varphi_m$
1	0.56	0.15	0.15	0	0.96	56.5°	0.76
2	0.57	0.30	0.15	0.15	1.17	27.7°	0.56
3	0.60	0.45	0.15	0.30	1.18	27.9°	0.40

Table 3.3 Codesign results under EDF scheduling: (a) after one iteration, (b) after ten iterations.

(a)

Task	h	R	R^b	J	$J_m(R^b)$	$\hat{\varphi}_m$	$\hat{\varphi}_m/\varphi_m$
1	0.28	0.16	0.15	0.01	1.11	64.0°	0.86
2	0.46	0.34	0.15	0.19	1.21	33.4°	0.67
3	1.53	1.35	0.60	0.75	0.03	−18°	−0.27

(b)

Task	h	R	R^b	J	$J_m(R^b)$	$\hat{\varphi}_m$	$\hat{\varphi}_m/\varphi_m$
1	0.40	0.31	0.15	0.16	1.04	43.1°	0.58
2	0.50	0.40	0.15	0.25	1.19	26.7°	0.54
3	0.54	0.45	0.15	0.30	1.20	29.8°	0.43

This work has only treated output jitter. In some applications, sampling jitter is also an issue. We are investigating if the stability analysis can be extended to also handle this case.

The topic of best-case response-time analysis needs to be investigated further. For instance, exact best-case response-time analysis under EDF could be developed. It would also be interesting to consider jitter analysis where the same task phasing is assumed for the best-case and the worst-case response-time analysis.

The suggested codesign approach is only one of many possible. It would be interesting to also consider direct digital design, where the controller is designed to compensate for the constant delay. In this case, a quadratic cost function is probably a better performance measure than the apparent phase margin.

4. S.Ha.R.K. Scheduling Modules

4.1 Background

S.ha.R.K. [Gai *et al.*, 2001] is a modular, open-source real-time kernel developed at Real-Time Systems Laboratory at the Scoula Superiore S. Anna di Pisa, Italy. The kernel supports hierarchical scheduling and a number of different resource-access protocols. There is also support for a large number of peripheral units, including many Linux 2.6 drivers for keyboard, framegrabber, network, etc. When building an application in S.ha.R.K., the user selects which scheduling modules, resource protocols, and hardware drivers to use by including them in the initialization function. The application is compiled, together with relevant the kernel files, into an executable file.

A typical hierarchy of scheduling modules is shown in Figure 4.1. At the top, at the highest priority, is a server mechanism for the scheduling of the device driver interrupts. At the priority below follows the first user-level module, the EDF scheduling module. Here, tasks with hard deadlines are scheduled. At the next layer is the CBS module, which is used to schedule soft and aperiodic tasks. This module inserts jobs into the EDF module, as indicated in the figure. At the lowest layers we find the round robin module for scheduling of the main() function and the dummy layer (for the dummy task). At each scheduling point, the kernel schedules the highest-priority ready task in the highest layer for execution. If no task is eligible for execution in layer 0, the kernel proceeds to layer 1, and so on.

A new scheduling module can be created by implementing a set of kernel interface functions. It is the responsibility of the module to keep track of the task data structures, the sorting of the ready queue (if any), and to post and handle any timers that are needed by module. Via the kernel interface, the module is informed about when a task is created, activated, preempted, blocked, finished, etc. Special kernel calls can be used to insert individual jobs into other scheduling modules.

During this project, the EDF and RM scheduling modules have been extended to handle task offsets and deadlines less than the task period. Some different overrun handling options have also been included. A new module has been written to implement control server scheduling, including some different options for overrun handling. Finally, a new module for the elastic task model has also been implemented, together with Giacomo Guidi and Mauro Marinoni.

4.2 Improved EDF and RM Scheduling Modules

The EDF (earliest-deadline-first) and RM (rate-monotonic) scheduling modules were originally designed to schedule hard periodic tasks according to the classical

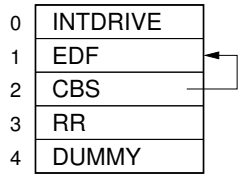

Figure 4.1 A typical hierarchy of scheduling modules in S.Ha.R.K.

Liu and Layland algorithms [Liu and Layland, 1973]. In the model, each task was described by a period T and a worst-case computation time C. The relative deadline was assumed to be equal to the period, i.e., $D = T$. There was also support for sporadic tasks, where T is interpreted as a minimum interarrival time. If a task violated its computation time, deadline, or minimum interarrival time, an exception was generated.

The new version of the modules introduces the possibility to have relative deadlines smaller than the period and release offsets. The HARD_TASK_MODEL included a field for the deadline, which was previously ignored by the EDF and RM modules. A new field was added to support release offsets. The updated task model structure looks as follows:

```
typedef struct {
  TASK_MODEL t;
  TIME mit;
  TIME drel;
  TIME wcet;
  int periodicity;
  TIME offset;
} HARD_TASK_MODEL;
```

If the deadline attribute is not specified, $D = T$ is assumed. If $D < T$ is used, the utilization factor of the task is computed as $U = C/D$. Unfortunately, the schedulability guarantee test in S.Ha.R.K. only uses the utilization factor, so this can result in a pessimistic design with low average utilization.

The task offsets can be used in two ways. First, they can be used to synchronize two or more tasks that are part of the same *transaction*. In this model, an event triggers a chain of tasks that should be executed at points in time relative to the event. The first task in the chain is typically given the offset 0. A transaction is defined using a *task group* in S.Ha.R.K., as in the following example:

```
  hard_task_def_offset(task1,0);
  hard_task_def_group(task1,1);
  hard_task_def_offset(task2,10000);
  hard_task_def_group(task2,1);
  ...
  group_activate(1); // activate task1 now and task2 10000 us later
```

The second way to use offsets is to assign an absolute release time to a task (or a group of tasks). This can be done using the two new primitives `task_activate_at` and `group_activate_at`:

```
/*+ Activate a task specified via pid from task_create at time t +*/
int task_activate_at(PID pid, struct timespec *t);

/*+ Activate the task group g at time t +*/
int group_activate_at(WORD g, struct timespec *t);
```

The EDF and RM modules can generate exceptions in the case of execution overruns, deadline overruns, and too frequent activation. Some additional flags have been added to control the behavior in the case of overruns. They are summarized in Table 4.1.

To implement the release offsets, a new transition was introduced in the state transition diagram, see Figure 4.2. When a task is activated with an offset, the task is put in the IDLE state, and a timer is posted to wake up the task at the correct time. To keep track of deadlines less than the period, an additional deadline timer was introduced. The timer is disabled when the task finished (i.e., when task_endcycle is called).

Table 4.1 Exception flags in the new EDF and RM modules.

(No flags enabled)	Deadline and wcet overruns are ignored. Pending periodic jobs are queued and are eventually scheduled with correct deadlines according to their original arrival times. Sporadic tasks that arrive to often are simply dropped.
EDF_ENABLE_DL_CHECK	When a deadline overrun occurs, a deadline miss counter for the task is increased. Same behavior for pending jobs as above.
EDF_ENABLE_WCET_CHECK	When a wcet overrun occurs, a WCET miss counter for the task is increased. Same behavior for pending jobs as above.
EDF_ENABLE_DL_EXCEPTION	When a deadline overrun occurs, an exception is raised.
EDF_ENABLE_WCET_EXCEPTION	When a wcet overrun occurs, an exception is raised.
EDF_ENABLE_ACT_EXCEPTION	When a periodic or sporadic task is activated too often, an exception is raised.

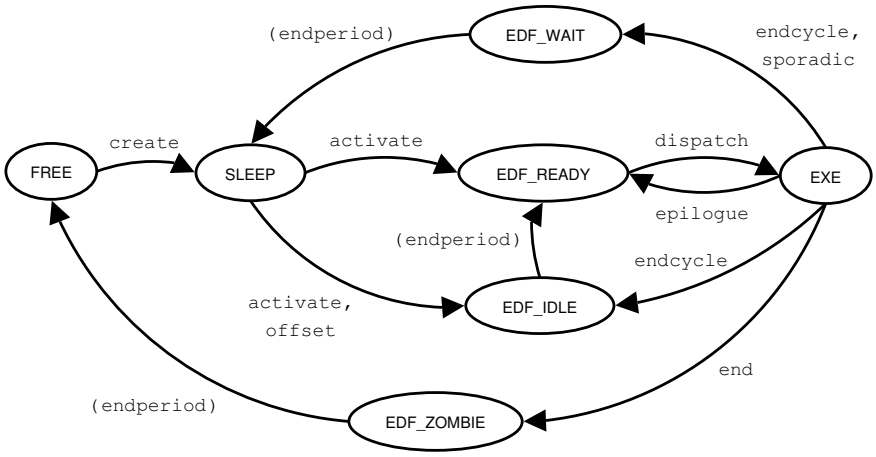

Figure 4.2 Task state transition diagram for the new EDF module.

4.3 The Control Server Scheduling Module

The Control Server is a variant of the constant bandwidth server (CBS), specifically designed to schedule control tasks that are sensitive towards jitter. A task may be divided into several segments, and time-triggered I/O points may be defined. The scheduling algorithm has been described in Chapter 2 of this report. The Control Server scheduling module in S.Ha.R.K. schedules tasks that are defined using the CS_TASK_MODEL, as defined below:

```
typedef struct {
  TASK_MODEL t;
  TIME offset;              /* release offset             */
  int nbr_segs;             /* the number of segments     */
  TIME *seg_lens;           /* pointer to segment lengths */
  void (**seg_inhooks)();   /* pointer to input hooks     */
  void (**seg_outhooks)();  /* pointer to output hooks    */
  bandwidth_t bandwidth;    /* task bandwidth             */
  int periodicity;
```

```
} CS_TASK_MODEL;
```

An absolute time release offset may be given to synchronize the task to other tasks in the system (including other modules). The number of and lengths (in microseconds) of the segments must be specified, together with optional input and output hooks for each segment. The reserved bandwidth of the task must be given, and the task may be declared as periodic or aperiodic.

As with the other scheduling modules, a set of macros are provided to simplify the initialization of a task structure. The example below shows how a task with two segments is specified:

```
static TIME seg_lens0[] = { 1000000, 2000000 };
static void (*seg_inhooks0[])() = { inhook1, inhook2 };
static void (*seg_outhooks0[])() = { outhook1, outhook2 };
...
cs_task_default_model(m);
cs_task_def_nbr_segs(m,2);
cs_task_def_seg_lens(m,seg_lens0);
cs_task_def_seg_inhooks(m,seg_inhooks0);
cs_task_def_seg_outhooks(m,seg_outhooks0);
cs_task_def_bandwidth(m, MAX_BANDWIDTH/2);
```

Here, the segment lengths are given in microseconds, and the input and output hooks pointers to functions that should perform functions such as reading samples or writing control signals.

A control server task is implemented by the user as an infinite loop, where a special primitive `CS_task_endsegment` is called after each segment (equivalent to `task_endcycle` for regular tasks). An example with two segments is given here:

```
void *cstask_code(void *p)
{
  while (1) {
    /* First segment */
    // do work here
    CS_task_endsegment();
    /* Second segment */
    // do some more work here
    CS_task_endsegment();
  }
}
```

Another special primitive, CS_task_late, can be used to check if the task has exhausted its budget and is currently executing with a postponed deadline. To implement period skipping in the case of overruns, the following scheme can be used:

```
void *cstask_code(void *p)
{
  while (1) {
    if (CS_task_late()) {
      // Late, skipping a period
      CS_task_endsegment();
      CS_task_endsegment();
      continue;
    }
     /* First segment */
    // do work
    CS_task_endsegment();
```

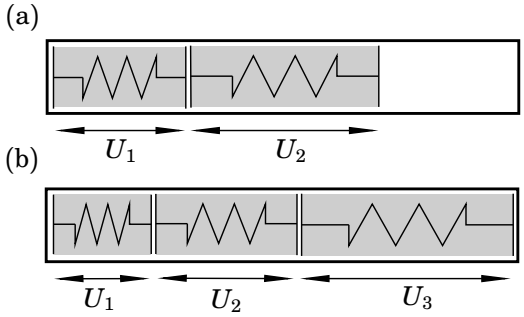

Figure 4.3 Illustration of the utilization rescaling in the elastic task model: (a) Tasks 1 and 2 are executing at their nominal utilization. (b) Tasks 1 and 2 are compressed to make room for the new task 3.

```
    /* Second segment */
    // do some more work
    CS_task_endsegment();
  }
}
```

Internally, each control server task is driven by a timer. The timer is set up to expire between each segment. In the timer handler, the following operations are carried out:

- The output hook of the previous segment is called (if not null).
- The input hook of the next segment is called (if not null).
- A new job is released.

The jobs are scheduled according to the CBS algorithm, except that the server period is dynamic (and equal to the current segment length). The jobs are inserted as guests in the EDF module (which must be present in the system). A counter is used to keep track of queued jobs in the case of overruns.

4.4 The Elastic Scheduling Module

The idea of the elastic task model [Buttazzo *et al.*, 1998] is to treat task with flexible resource requirements as springs that can be compressed and decompressed, see Figure 4.3. This can be useful to model for instance control tasks that can give acceptable performanace over a range of different sampling periods. By rescaling the task periods when tasks enter or leave the system, the elastic manager can make sure that the task set is schedulable at all times.

In the model, each task is described by a worst-case execution time C, a minimum (nominal) period $T_{\min}$, a maximum period $T_{\max}$, and an elasticity coefficient E. Whenever possible, the tasks execute at their maximum rate (i.e., with the period $T_{\min}$). If that is not possible, then the task periods are rescaled in proportion to the elasticity of the tasks. A task can never be forced to execute with a longer period than its declared $T_{\max}$, however. In S.Ha.R.K., an elastic task is described by the following model structure:

```
typedef struct {
  TASK_MODEL t;
  TIME Tmin;
  TIME Tmax;
  TIME C;
  int  E;
```

```
  int  beta;
  int  arrivals;
} ELASTIC_TASK_MODEL;
```

Here, the `beta` attribute is reserved for future extensions where the task might be rescaled either by its period or by its computation time.

In the scheduling module, the elastic compression algorithm is implemented in a function which must be called with interrupts disabled. The function is called whenever a task is created or destroyed, or whenever a relevant task attribute is modified by the user.

If, during the compression, any task periods are shrunk, the periods of those tasks are changed at their next release. If periods are prolonged, they are immediately changed. This involves removing the task from the EDF level, changing the deadline, and inserting the task again. This way, the task set remains schedulable at all times.

5. Conclusion

This report has presented the work carried out within the six-month research project “Merging Real-Time and Control Theory for Improving the Performance of Embedded Control Systems”. The project has focused on handling timing variations in real-time control.

The problem of overruns due to varying control task execution times has been investigated. Theoretical analysis of simple models has shown that the performance can be improved by sampling fast but allowing occasional overrun. The overrun handling has been implemented in the control server scheduling module and practical experiments on a vision-based ball-and-plate control application have been performed. The practical experiments were inclusive, however.

From a control design point of view, the problem of response-time jitter in control tasks has been researched. By calculating the jitter for each task in the system, the control stability can asserted. The A new lower bound for the best-case response time under EDF scheduling has been developed.

Finally, some scheduling modules in the S.Ha.R.K. real-time kernel has been implemented or improved. The rate-monotonic and EDF modules have been extended to handle offsets. The control server scheduling mechanism has been implemented in a new module. Also, an elastic scheduling module has been implemented.

References

Abeni, L. and G. Buttazzo (1998): "Integrating multimedia applications in hard real-time systems." In *Proc. 19th IEEE Real-Time Systems Symposium*. Madrid, Spain.

Albertos, P. and A. Crespo (1997): "Real-time control of unconventionally sampled data systems." In *Proc. 4th IFAC Workshop on Algorithms and Architectures for Real-Time Control, Algarve, Portugal*.

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

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems*. Prentice Hall.

Audsley, N., K. Tindell, and A. Burns (1993): "The end of the line for static cyclic scheduling." In *Proc. 5th Euromicro Workshop on Real-Time Systems*.

Balbastre, P., I. Ripoll, and A. Crespo (2000): "Control task delay reduction under static and dynamic scheduling policies." In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*.

Baruah, S., G. Buttazzo, S. Gorinsky, and G. Lipari (1999): "Scheduling periodic task systems to minimize output jitter." In *Proc. 6th International Conference on Real-Time Computing Systems and Applications*.

Buttazzo, G. (2003): "Rate monotonic vs. EDF: Judgment day." In *Proceedings of the 3rd ACM International Conference on Embedded Software (EMSOFT'03)*.

Buttazzo, G., G. Lipari, and L. Abeni (1998): "Elastic task model for adaptive rate control." In *Proc. 19th IEEE Real-Time Systems Symposium*, pp. 286–295.

Caccamo, M. and G. Buttazzo (1997): "Exploiting skips in periodic tasks for enhancing aperiodic responsiveness." In *Proc. 18th IEEE Real-Time System Symposium*.

Caccamo, M., G. Buttazzo, and L. Sha (2000): "Elastic feedback control." In *Proc. 12th Euromicro Conference on Real-Time Systems*, pp. 121–128. Stockholm, Sweden.

Caccamo, M., G. Buttazzo, and L. Sha (2002): "Handling execution overruns in hard real-time control systems." *IEEE Transactions on Computers*, **51:7**.

Cervin, A. (1999): "Improved scheduling of control tasks." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 4–10. York, UK.

Cervin, A. and J. Eker (2003): "The Control Server: A computational model for real-time control tasks." In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pp. 113–120. Porto, Portugal.

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.

Crespo, A., I. Ripoll, and P. Albertos (1999): "Reducing delays in RT control: The control action interval." In *Proc. 14th IFAC World Congress*, pp. 257–262.

David, L., F. Cottet, and N. Nissanke (2001): "Jitter control in on-line scheduling of dependent real-time tasks." In *Proc. 22nd IEEE Real-Time Systems Symposium*.

Franklin, G., D. Powell, and A. Emami-Naeini (2002): *Feedback Control of Dynamic Systems*, 4th edition. Prentice Hall.

Gai, P., L. Abeni, M. Giorgi, and G. Buttazzo (2001): "A new kernel approach for modular real-time systems development." In *Proc. 13th Euromicro Conference on Real-Time Systems*.

George, L., N. Rivierre, and M. Spuri (1996): "Preemptive and non-preemptive real-time uniprocessor scheduling." Technical Report 2966. Institut National de Recherche en Informatique et en Automatique.

Halang, W. (1993): "Achieving jitter-free and predictable real-time control by accurately timed computer peripherals." *Control Engineering Practice*, **1:6**, pp. 979–987.

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.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch (2001): "Giotto: A time-triggered language for embedded programming." In *Proc. First International Workshop on Embedded Software*.

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

Kao, C.-Y. and B. Lincoln (2004): "Simple stability criteria for systems with time-varying delays." *Automatica*. To appear in September 2004. Preprint available at http://www.control.lth.se.

Kim, T., H. Shin, and N. Chang (2000): "Deadline assignment to reduce output jitter of real-time tasks." In *Proc. 16th IFAC Workshop on Distributed Computer Control Systems*.

Klein, M. H., T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Härbour (1993): *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publisher.

Koren, G. and D. Shasha (1995): "Skip-over: Algorithms and complexity for overloaded systems that allow skips." In *Proc. IEEE Real-Time Systems Symposium*.

Lehoczky, J., L. Sha, and J. Strosnider (1987): "Enhanced apriodic responsiveness in hard real-time environment." In *Proc. 8th IEEE Real-Time Systems Symposium*.

Lincoln, B. (2002): "Jitter compensation in digital control systems." In *Proceedings of the 2002 American Control Conference*.

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

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.

Locke, C. D. (1992): "Software architecture for hard real-time applications: Cyclic vs. fixed priority executives." *Real-Time Systems*, **4**, pp. 37–53.

Marti, P., G. Fohler, K. Ramamritham, and J. M. Fuertes (2001): "Jitter compensation for real-time control systems." In *Proc. 22nd IEEE Real-Time Systems Symposium*.

Nilsson, J. (1998): *Real-Time Control Systems with Delays*. PhD thesis ISRN LUTFD2/TFRT--1049--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

Ramanathan, P. (1997): "Graceful degradation in real-time control application using (m,k)-firm guarantee." In *Proc. 27th Annual International Symposium on Fault-Tolerant Computing*.

Redell, O. and M. Sanfridson (2002): "Exact best-case response time analysis of fixed priority scheduled tasks." In *Proc. 14th Euromicro Conference on Real-Time Systems*. Vienna, Austria.

Ryu, M., S. Hong, and M. Saksena (1997): "Streamlining real-time controller design: From performance specifications to end-to-end timing constraints." In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium*, pp. 91–99.

Seto, D., J. P. Lehoczky, L. Sha, and K. G. Shin (1996): "On task schedulability in real-time control systems." In *Proc. 17th IEEE Real-Time Systems Symposium*, pp. 13–21. Washington, DC.

Shin, K. G. and H. Kim (1992): "Derivation and application of hard deadlines for real-time control systems." *IEEE Transactions on Systems, Man, and Cybernetics*, **22:6**, pp. 1403–1413.

Shin, K. G., C. M. Krishna, and Y.-H. Lee (1985): "A unified method for evauating real-time computer controllers and its applications." *IEEE Transactions on Automatic Control*, **30:4**, pp. 357–366.

Spuri, M. (1996): "Analysis of deadline scheduled real-time systems." Technical Report 2772. INRIA, France.

Spuri, M. and G. Buttazzo (1996): "Scheduling aperiodic tasks in dynamic priority systems." *Real-Time Systems*, **10:2**, pp. 179–210.

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.

Tindell, K., A. Burns, and A. J. Wellings (1994): "An extendible approach for analyzing fixed priority hard real-time tasks." *Real-Time Systems*, **6:2**, pp. 133–151.

Törngren, M. (1998): "Fundamentals of implementing real-time control applications in distributed computer systems." *Real-Time Systems*, **14:3**.