The Control Server Model for Co-Design of Real-Time Control Systems

Cervin, Anton; Eker, Johan

Published in:
ARTES -- A network for Real-Time research and graduate Education in Sweden 1997--2006

2006

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

The Control Server Model for Codesign of Real-Time Control Systems

Anton Cervin* and Johan Eker†

Abstract

The paper presents the control server, a real-time scheduling mechanism tailored to control and signal processing applications. A control server creates the abstraction of a control task with a specified period and a fixed input-output latency shorter than the period. Individual tasks can be combined into more complex components without loss of their individual guaranteed fixed-latency properties. I/O occurs at fixed predefined points in time, at which inputs are read or controller outputs become visible. The control server model is especially suited for codesign of real-time control systems. The single parameter linking the scheduling design and the controller design is the task utilization factor. The proposed server is an extension of the constant bandwidth server, which is based on the earliest-deadline-first scheduling algorithm. The server has been implemented in a real-time kernel and has also been validated in control experiments on a ball and beam process.

1 Introduction

The design of a real-time control system is essentially a codesign problem. Decisions made in the real-time design affect the control design, and vice versa. For instance, the choice of scheduling policy influences the latency distributions in the control loops, and, ideally, this should be taken into account in the control design. At the same time, the performance requirements of the individual control loops place demands on the real-time system with regard to sampling periods, latencies, and jitter.

Traditional scheduling models give poor support for codesign of multi-threaded real-time control systems. One difficulty lies in the nonlinearity in scheduling mechanisms such as rate-monotonic (RM) or earliest-deadline-first (EDF) scheduling: a small change in a task parameter—e.g., period, execution time, deadline, or priority—may give rise to unpredictable results in terms of input-output latency and jitter. This is crucial, since the performance of a controller depends not only on its sampling period, but also on the latency and the jitter. In the control design, it is straight-forward to account for a constant latency, while it is difficult to address varying or unknown delays.

*A. Cervin is with the Department of Automatic Control, Lund Institute of Technology, SE-221 00 Lund, Sweden, email: anton@control.lth.se

†J. Eker is with Ericsson Mobile Platforms, SE-221 83 Lund, Sweden, email: johan.eker@emp.ericsson.se

In the seminal Liu and Layland paper [1], it is assumed that I/O is performed periodically by hardware functions, introducing a one-sample delay in all control loops closed over the computer. This scheme does provide a quite nice separation between the scheduling design and the control design. From a scheduling perspective, the controller can be described by a periodic task with a period T, a computation time C, and a deadline $D = T$. From a control perspective, the controller will have a sampling period of T and a constant latency $L = T$. This allows the control design and the real-time design to be carried out in relative isolation.

However, the one-sample latency degrades the control performance and is ultimately a waste of resources. A common alternative implementation is therefore to perform the I/O requests within the task loop and output the control signal as soon as possible in each period (e.g., [2, 3]). At this point, however, the design problem becomes very complicated. The I/O jitter and latency of a controller are now affected by variations in its own execution time as well as interference from higher-priority tasks (which in turn depend on the variations in the task execution times, the phasing of the periodic tasks, the arrival pattern of sporadic tasks, etc.). In the best case, it may be possible to derive formulas for the worst-case and best-case response times of the tasks (e.g., [4, 5]), but this information is still not sufficient to accurately predict the performance of the controllers. Furthermore, as argued in [6], with standard RM and EDF scheduling it can be difficult to map task importance into priorities and/or deadlines. These algorithms also perform poorly if tasks deviate from their assumed behavior or if the CPU should become overloaded.

1.1 Model Overview

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

(R1) isolation between unrelated tasks,

(R2) short input-output latencies,

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

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

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

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

Requirement (R1) is fulfilled by the use of constant bandwidth servers (CBSs) [7]. The servers make each task appear as if it was running on a dedicated CPU with a given fraction of the original CPU speed. To facilitate short latencies (requirement (R2)), a task may be divided into a number of *segments*, which are scheduled individually. A task may only read inputs (from the environment or from other tasks) at the beginning of a segment and write outputs (to the environment or to other tasks) at the end of a

segment. All communication is handled by the kernel and is hence not prone to jitter (requirement (R3)).

Requirements (R4)–(R6) are addressed by the combination of bandwidth servers and statically scheduled communication points. For periodic tasks with constant execution times, the model creates the illusion of a perfect division of the CPU, equivalent to the Generalized Processor Sharing (GPS) algorithm [8]. The model makes it possible to analyze each task in isolation, from both scheduling and control points of view. Like ordinary EDF, schedulability of the task set is simply determined by the total CPU utilization (ignoring context switches and the I/O operations performed by the kernel). The performance of a controller can also be viewed as a function of its alloted CPU share. These properties make the model very suitable for feedback scheduling applications.

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

1.2 Related Work

The constant bandwidth server (CBS) [7] was originally proposed as a means to bound the utilization of soft or aperiodic real-time A CBS creates the abstraction of a virtual CPU with a given capacity (or *bandwidth*) U_s. Tasks executing within the CBS cannot consume more than the reserved capacity. Hence, from the outside, the CBS will appear as an ordinary EDF task with a maximum utilization of U_s. The time granularity of the virtual CPU abstraction is determined by the *server period* T_s.

In [9], a variant of the CBS server, called $\mathrm{CBS}^{\mathrm{hd}}$, is used to schedule control tasks with varying execution times. In the case of an execution overrun, the current period is extended and the CBS budget is recharged in small increments until the task finishes.

Minimizing jitter using high-priority tasks or interrupt handlers has been suggested in various settings, e.g., [10, 2, 11]. Disadvantages of the approach include a more complex implementation and more run-time overhead. Also, reducing jitter means increasing the average input-output latency. In [12], a design procedure that minimizes input-output jitter using high-priority input and output tasks is presented. Task attribute assignment under both FP and EDF scheduling is considered. Another option to reduce input-output jitter is to use non-preemptive scheduling. Given that the control algorithm has a constant execution time, this will make the input-output latency constant. The drawback is that the scheduling design becomes more complicated.

Giotto [13] is an abstract programming model for the implementation of embedded control systems. Similar to our model, I/O and communication are time-triggered and assumed to take zero time, while the computations inbetween are assumed to be scheduled in real-time. A serious drawback with the model is that a minimum of one sample input-output latency is introduced in all control loops. Also, Giotto does not address the scheduling problem.

Within the Ptolemy project, e.g., [14], a computational domain called Timed Multitasking (TM) has been developed [15]. In the model, tasks (or *actors* in the terminology

of Ptolemy) may be triggered by both periodic and aperiodic events. Inputs are read when the task is triggered and outputs are written at the specified task deadline. The computations inbetween are assumed to be scheduled by a fixed-priority dispatcher. In the case of a deadline overrun, an overrun handler may be called. Again, the scheduling problem is not explicitly addressed by the model.

2 The Model

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

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

2.1 Control Server Tasks

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 $\sum_{j=1}^{n_i} l_i^j = T_i$,
- a set of inputs I_i (associated with physical inputs or shared variables), and
- a set of outputs O_i (associated with physical outputs or shared variables).

Associated with each segment S_i^j are

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

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

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

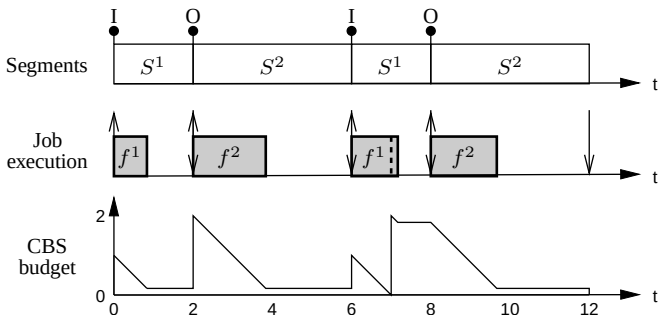

Figure 1: Example of a control server task executing alone. The up arrows indicate job releases and the down arrows indicate deadlines. The overrun at $t = 7$ causes the deadline to be postponed to the end of the next segment.

- 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, $m_i := \text{mod}(m_i, n_i) + 1$,
   - the deadline is moved, $d_i := d_i + l_i^{m_i}$, and
   - the budget is recharged to $c_i := U_i\, l_i^{m_i}$.

The rules are somewhat simplified compared to the original CBS rules [7] due to the predictable pattern of release times and deadlines. The only real difference from an ordinary CBS is that here a "dynamic server period", equal to the current segment length, $l_i^{m_i}$, is used.

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

2.2 Aborted Computations

The default behavior of the control server is to allow budget recharging across the task period. In the case of constant overruns, this will cause the task deadline to be postponed repeatedly and eventually reach infinity. For some applications, a better choice may be to abort the task when the total budget in the period has been exhausted. The choice of whether to abort the task in case of a period budget overrun may be specified separately for each control server task.

2.3 Communication and Synchronization

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

Communication between tasks is handled via shared variables. If an input is associated with a shared variable, the value of the variable is copied to the input buffer at the beginning of the relevant segment. Similarly, if an output is associated with a shared variable, the value in the output buffer is copied to the shared variable at the end of the relevant segment. Interrupts are assumed to be disabled when accessing buffers and shared data.

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

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

2.4 Scheduling Properties

From a schedulability point of view, a control server task with the CPU share U_i is equivalent to a CBS server with the bandwidth U_i. In [16], it is shown that a CBS can never demand more than its reserved bandwidth. By postponing the deadline when the budget is exhausted, the loading factor of the jobs served by the CBS can never exceed U_i. The same argument holds for the modified CBS used in the control server model.

A set of CBS and control server tasks is thus schedulable if and only if

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

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

$$l_i^j = C_i^j / U_i, \tag{2}$$

where C_i^j denotes the worst-case execution time (WCET) of the code function f_i^j, overruns will never occur (i.e., the budget will never be exhausted before the end of the segment), and all latencies will be constant. For tasks with large variation in their execution time, it can sometimes be advantageous to assign segment lengths that are shorter than those given by (2). This means that some deadlines will be postponed and that the task may not always produce a new output in time, delaying the output one or more periods. An example of when this can actually give better control performance (for a given value of U_i) is given later.

3 Control and Scheduling Codesign

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

3.1 Control Performance

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

$$J = J(T,\, L_{io},\, J_s,\, J_{io}). \tag{3}$$

Assuming that the first segment contains the Calculate Output part of the control algorithm, and that the segment lengths are chosen according to (2), execution under the Control Server implies

$$\begin{aligned} T &= \sum l^k = \sum C^k / U, \\ L_{io} &= l^1 = C^1 / U, \\ J_s &= 0, \\ J_{io} &= 0. \end{aligned} \tag{4}$$

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

$$J = J(U). \tag{5}$$

Assuming a linear controller, a linear plant, and a quadratic cost function, the performance of the controller for different values of U can easily be computed using the Matlab toolbox Jitterbug [17].

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

3.2 Codesign Example 1: Importance of Reducing Latency

Consider optimal control of the integrator process

$$\frac{dx(t)}{dt} = u(t) + v_c(t). \tag{6}$$

Here, x is the state (which should be controlled to zero), u is the control signal, and v_c is a continuous-time white noise disturbance with zero mean and unit variance. A discrete-time controller is designed to minimize the continuous-time cost function

$$J = \lim_{t\to\infty} \frac{1}{t} \int_0^t x^2(s)\, ds. \tag{7}$$

Dividing the control computations into two segments and choosing the segment lengths in proportion to WCET of the parts, the control server model will generate equidistant sampling with the interval T and a constant latency L. The cost for the optimal, delay-compensating controller can be shown to be

$$J(T,\, L) = \frac{3+\sqrt{3}}{6} T + L \;(\approx 0.79T + L). \tag{8}$$

(For details, see Appendix A.) It can be noted that, in this case, the cost grows linearly with both the sampling interval and the latency. Furthermore, for a fixed value of J (i.e., a specified level of performance), T is determined by L. This implies that a controller with a short latency will be less CPU-demanding than a controller with a long latency. In Table 1, the relative CPU demand of the integrator controller has been computed for different values of the relative latency L/T. The case $L/T = 1$ corresponds to a Liu and Layland implementation with a one sample delay. As the latency is reduced (by, e.g., a suitable division of the control algorithm into a Calculate Output segment and an Update State segment), the CPU demand of the controller can be decreased.

Table 1: Relative CPU demand of the integrator controller for different relative latencies.

L/T	CPU demand
1	1.00
0.5	0.72
0.25	0.58
0.1	0.50

3.3 Codesign Example 2: Optimal Period Selection

In this example we study the problem of optimal sampling period selection for a set of control loops. This type of codesign problem first appeared in [18]. In those cases, however, the scheduling-induced latency and jitter was ignored.

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

$$\frac{dx(t)}{dt} = u(t) + v_c(t), \tag{9}$$

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

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T x^2(t)\, dt. \tag{10}$$

assuming the sampling period h and a constant input-output latency L. The cost of the optimal controller is given by

$$J(h,\, L) = \frac{3+\sqrt{3}}{6} h + L \tag{11}$$

(see Appendix I). The assumed design goal is to select sampling periods h_1, h_2, h_3 such that a weighted sum of the cost functions,

$$J_{tot} = w_1\, J(h_1,\, L_1) + w_2\, J(h_2,\, L_2) + w_3\, J(h_3,\, L_3), \tag{12}$$

is minimized subject to the utilization constraint

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

Here, C is the (constant) total execution time of the control algorithm. Assigning segment lengths proportional to the parts of the algorithm, the control server model implies the same relative latency $a = L/h$ for all controllers. Using (11) the objective function (12) can be written

$$J_{tot} = \left(\frac{3+\sqrt{3}}{6} + a\right)(w_1 h_1 + w_2 h_2 + w_3 h_3).$$

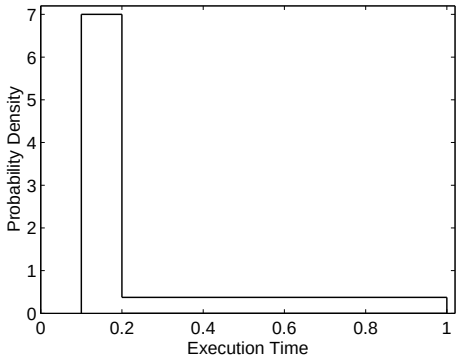

Figure 2: Assumed execution time probability distribution of the integrator controller.

The solution to the optimization problem is

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

where $b = C(\sqrt{w_1}+\sqrt{w_2}+\sqrt{w_3})$. (For more general problems numerical optimization must be performed.) Contrary to [18] (where RM or EDF scheduling is assumed), our model allows for the latency and the (non-existent) jitter to be accounted for in the optimization.

3.4 Codesign Example 3: Overrun Handling

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

Again consider the integrator controller. For simplicity, it is assumed that the controller is implemented as a single segment, i.e., we have $L_{io} = T$ if no overrun occurs, and that the assigned CPU share is $U = 1$. The controller is designed for a constant latency of $L_{io} = T$. Now assume that the execution time of the controller is given by the probability distribution in Fig. 2.

Two cases are compared. First, budget recharging across the period is allowed. If an overrun occurs, the computation continues in the next period, where an output is eventually produced. In the second case, it is the task is aborted in the case of an overrun. This means that no new output is produced, and a new computation is started in the next period The control performance in the different cases has been computed using Jitterbug [17]. In Fig. 3, the cost (7) has been computed for different values of the task period. When period overruns are allowed, the optimal cost is lower, $J = 1.55$, and occurs when $T = 0.5$. If task abortions are used, the optimal cost $J = 1.67$ occurs for $T = 0.76$. In this example, the default control server behavior with budget recharging across the task period yields the best control performance.

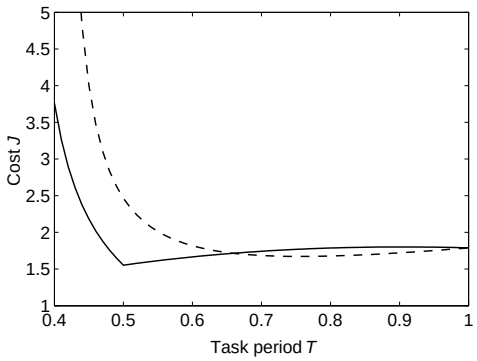

Figure 3: Cost as a function of the task period for the integrator controller, in the case of budget recharging across the period (full) and aborted computations (dashed). In both cases, assigning a task period shorter than the worst-case execution time can give better control performance.

4 Control Server Tasks as Real-Time Components

As argued in the previous section, given a control algorithm with known execution time C (divided into one or several segments), the sampling period T, the latency L_{io}, and the control performance J can be expressed as functions of the CPU share U. The predictable control and scheduling properties allows a control server task to be viewed as a scalable real-time component.

Consider for instance the PID (proportional-integral-derivative) controller component in Fig. 4. The controller has two inputs: the reference value r and the measurement signal y, and one output: the control signal u. The U knob determines the CPU share. An ordinary software component (see, e.g., [19]) would only specify the functional behavior, i.e., the PID algorithm. The specification for our real-time component includes the resource usage and the timely behavior. Assuming an implementation where the execution time of the Calculate Output part is $C^1 = 3.3$ ms and the execution time of the Update State part is $C^2 = 10.0$ ms, the specification of the PID component could look something like this:

- Parameters: U (CPU share), K, T_i, T_d (PID parameters)
- Execution time: $C = 13.3$ ms

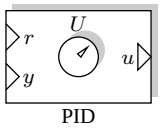

Figure 4: A PID controller component. The U knob determines both the scheduling and the control properties of the component.

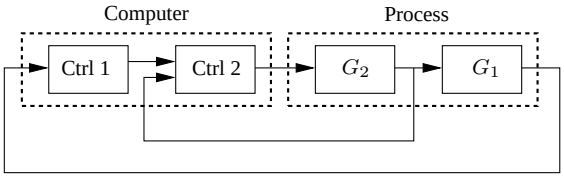

Figure 5: Cascaded controller structure.

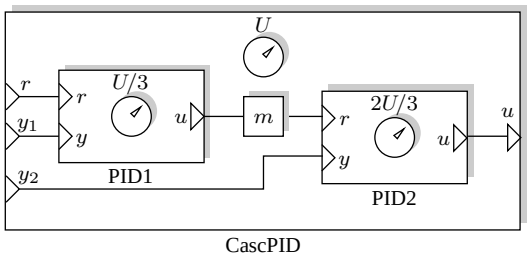

Figure 6: A cascaded PID controller component.

- Task period: $T = C/U$
- Latency: $L_{io} = C^1/U = T/4$
- Algorithm: $u = K\left((r-y) + \int \frac{1}{T_i}(r-y)dt + T_d\frac{-dy}{dt}\right)$, discretized using backward difference with interval T

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

Next, consider the composition of two PID controllers in a cascaded controller structure, see Fig. 5. In this very common structure, the inner controller is responsible for controlling the (typically) fast process dynamics G_2, while the outer controller handles the slower dynamics G_1. A cascaded controller component can be built from two PID components as shown in Fig. 6. In this case, it is assumed that the inner controller should have twice the sampling frequency of the outer controller (reflecting the speed of the processes). This is achieved by assigning the shares $U/3$ to PID1 and $2U/3$ to PID2, U being the CPU share of the composite controller. The end-to-end latency in the controller can be minimized by a suitable segment layout, see Fig. 7.

The schedulability and performance of the cascaded controller will, again, only depend on the total assigned CPU share U. The resulting controller is a multi-rate controller and its performance can be computed using Jitterbug [17]. As an example, in Fig. 8, the cost J as a function of the CPU share U has been computed for a cascaded PID controlling a ball and beam process.

Note that such composition is not possible with ordinary threads, i.e., two communicating threads cannot be treated as one, neither from schedulability nor control perspectives.

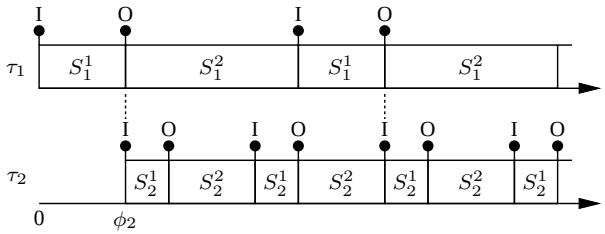

Figure 7: Segment layout in the cascaded PID controller. Task τ_2 is given an offset $\phi_2 = l_1^1$ such that the value written by S_1^1 is immediately read by S_2^1.

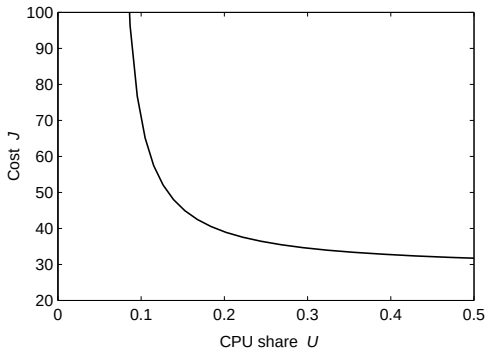

Figure 8: Control performance as a function of the CPU share for a cascaded PID controller.

Figure 9: Pseudo code for the modified real-kernel.

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

5 Implementation

As a proof of concept, the computational model has been implemented in the public-domain real-time kernel STORK [20], developed at the Department of Automatic Control, Lund Institute of Technology. The original kernel is a standard priority-preemptive real-time kernel written in Modula-2, running on multiple platforms. For this project, the Motorola PowerPC was chosen because of its high clock resolution (40 ns on a 100 MHz processor).

The kernel was modified to use EDF as the basic scheduling policy, and high-resolution timers (hardware clock interrupts that trigger user-defined handlers) were introduced. For tracing purposes, the kernel measures the execution-time of each task. An outline of the kernel code is shown in Fig. 9.

A number of data structures for CBS servers, control server tasks, segments, inputs, and outputs, etc., were introduced, see the UML diagram in Fig. 10. The tasks in the ready queue are sorted according to their absolute deadlines. Tasks that are associated

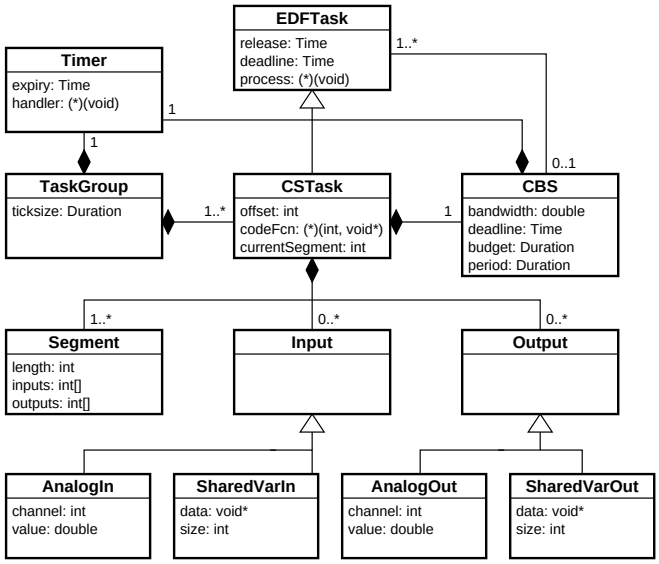

Figure 10: The various data structures in the implementation.

with a CBS inherit the deadline of the CBS. Note that several tasks may be served by the same CBS.

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

5.1 Task Group Timing

For synchronization reasons, communicating control server tasks must share a common timebase and are gathered in task groups. Each task group uses a timer to trigger the reading of inputs, writing of outputs, and release of segments of tasks within the group. The structure of the task group timer interrupt handler is shown in Fig. 11.

Associated with each control server task is a semaphore that is used to handle the release of the segment jobs. Internally, every control server task is implemented as a simple loop, see Fig. 12.

5.2 API

The kernel provides a number of primitives for defining task groups, EDF tasks, CBS servers, control server tasks, inputs and outputs, etc. The code of a control server task is written according to a special format, here illustrated with a PID controller (written in Modula-2), see Fig. 13. In the code, the kernel primitives `ReadInput` and `WriteOutput` are used to access the inputs and outputs associated with the segment.

Figure 11: Pseudo code for the task group timing.

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

Figure 12: Pseudo code for the internal implementation of a control server task.

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

Figure 13: Code function in Modula-2 representing a control server task.

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

Figure 14: The ball and beam process used in the control experiments.

To prevent shared data from being corrupted, interrupts are disabled in the read and write primitives.

6 Control Experiments

The implementation of the control server was validated in a number of real-time control experiments on the ball and beam process, see Fig. 14. The objective of the control is to move the ball to a given position on the beam. The input to the process is the beam motor voltage, and the outputs are voltages representing the beam angle and the ball position.

The process is controlled by a multirate cascaded PID controller, where the inner controller executes at twice the frequency of the outer controller (see Figs. 5–7). The

composite controller is assigned the CPU share $U = 0.5$. The execution time of the PID control algorithm is $C = 13.3$ ms, divided into a Calculate Output segment with $C^1 = 3.3$ ms and an Update State segment with $C^2 = 10$ ms. (To generate a high CPU load on the Power PC, busy cycles were inserted into the code.) The resulting sampling periods are $T_1 = 80$ ms for the outer PID controller and $T_2 = 40$ ms for the inner PID controller.

Also executing in the system is a sporadic task with an *assumed* minimum interarrival time of $T_{spor} = 20$ ms, and a worst-case execution time of $C_{spor} = 10$ ms. The actual execution time of the task is random and uniformly distributed between 2 and 10 ms. In the time interval 0 to 20, the actual interarrival time of the task is uniformly distributed between 20 and 40 ms. The average utilization of the sporadic task in this interval is 0.21. After $t = 20$ s, the interarrival time of the task suddenly decreases (hence violating the design assumptions) and is from then on uniformly distributed between 5 and 10 ms. The new average utilization of the sporadic task is 0.83, causing the CPU to be overloaded.

The behavior of the system under rate-monotonic scheduling, earliest-deadline-first scheduling, and control server scheduling was studied. Under all policies, PID2 was released with an offset of 20 ms compared to PID1. The same random sequence of execution times for the sporadic task were used in all experiments. For each run, the execution trace (i.e, the task schedule) was logged, together with measurements of the ball position and the control signal.

6.1 Rate-Monotonic Scheduling

Under rate-monotonic scheduling, the sporadic task has the shortest assumed minimum interarrival time and is assigned the highest priority, while PID1 and PID2 are assigned low and medium priorities respectively. The task set is schedulable according to rate-monotonic theory.

The control performance under rate-monotonic scheduling is shown in Fig. 15, and a close-up of the execution trace is shown in Fig. 16. As expected, the control performance is good up to $t = 20$. When the sporadic task starts to misbehave, PID1 becomes completely blocked and the controller stops to work. The result is that the ball rolls off the beam.

6.2 Earliest-Deadline-First Scheduling

Under earliest-deadline-first scheduling, the tasks are scheduled according to their absolute deadlines. The PID tasks are assigned relative deadlines equal to their periods, while the sporadic task is assigned a relative deadline equal to its assumed minimum interarrival time.

The control performance under earlist-deadline-first scheduling is shown in Fig. 17, and a close-up of the execution trace is shown in Fig. 18. Again, the control performance is good up to $t = 20$. After that, the step responses are slower and more oscillatory, but the control system is still stable. The CPU overload caused by the sporadic task causes the sampling periods of the controller to be rescaled. This property of control tasks under overloaded EDF scheduling was first explained in [21].

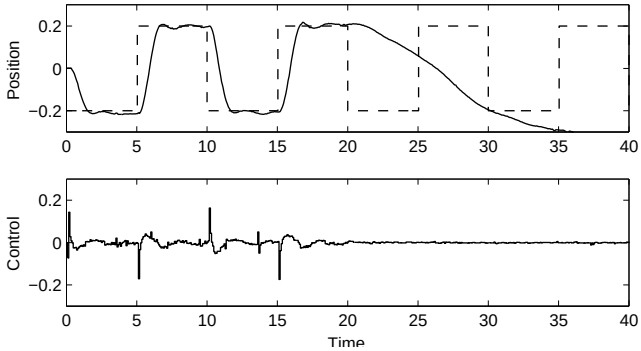

Figure 15: Control performance under rate-monotonic scheduling. The controller stops to work after $t = 20$.

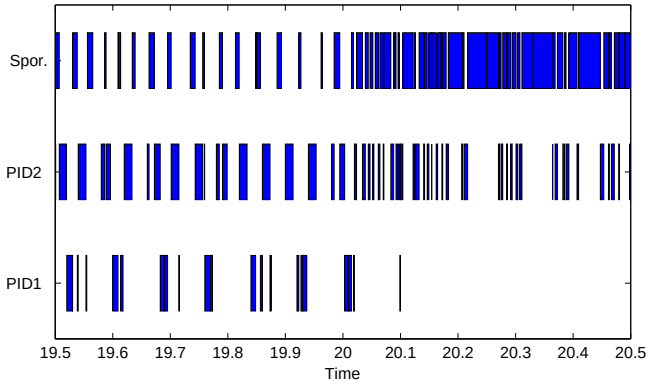

Figure 16: Execution trace under rate-monotonic scheduling. The sporadic task blocks the controller after $t = 20$.

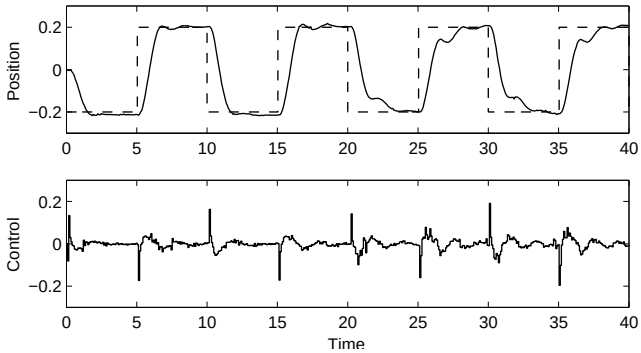

Figure 17: Control performance under earliest-deadline-first scheduling.

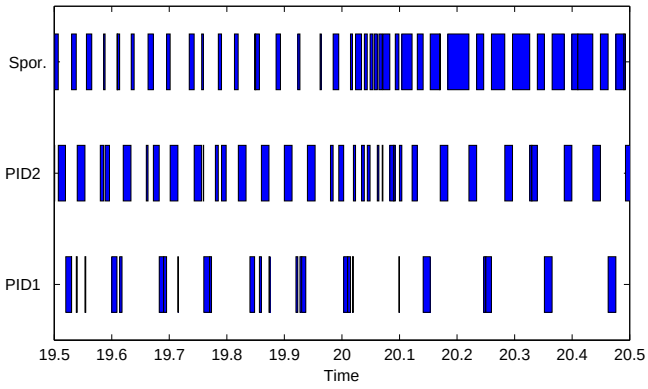

Figure 18: Execution trace under earliest-deadline-first scheduling.

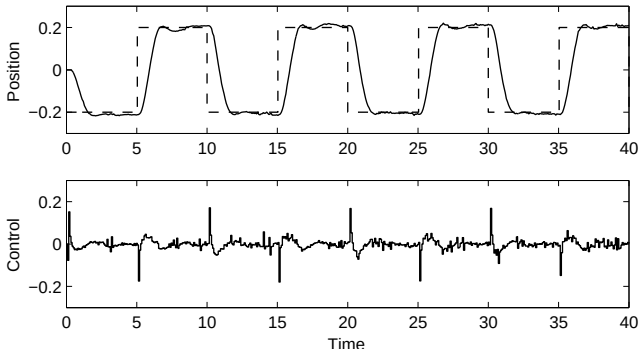

Figure 19: Control performance under control server scheduling. The performance is identical before and after $t = 20$.

6.3 Control Server Scheduling

Under control server scheduling, the cascade controller is assigned 50% of the processor, while the utilization of the sporadic task is bounded to 50% using a constant bandwidth server. The resulting control performance is shown in Fig. 19, and a close-up of the execution trace is shown in Fig. 20. Since the controller is no longer disturbed by the sporadic task, control performance is good throughout, and identical before and after $t = 20$. The schedule trace shows that the controller is no longer disturbed by the misbehaving sporadic task. Not shown in the trace is the fact that there is also no longer any I/O jitter.

7 Conclusion and Discussion of Future Work

This paper has presented the control server model, suitable for the implementation of feedback control tasks in embedded systems. Features of the model include small

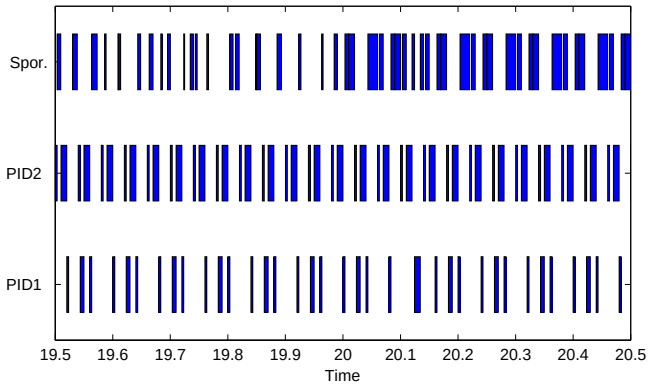

Figure 20: Execution trace under control server scheduling.

latency and jitter, something that is valuable from a control perspective. Based on the constant bandwidth server, the control server also provides isolation between unrelated tasks. A nice property of the control server is that both schedulability and control performance of a control task can be expressed through the task CPU share. The control server has been implemented and tested in a real control application with good results.

The current work can be extended in many directions. One topic that needs further investigation is that of overrun handling. In Section 3.4, an example was given where budget recharging across the periods was shown to give better control performance than aborted computations. It is also possible to find examples where the opposite is true. Can any general conclusions can be drawn? The problem of overrun handling becomes especially intricate when the control algorithm has been divided into segments. For instance, should the Update State part be executed even if the Calculate Output part was aborted (or did not finish in time)? Are some controller realizations more sensitive to aborted computations than others? Can controllers be designed to be robust against period overruns?

To provide better performance, the constant bandwidth servers used could be modified to use a slack stealing algorithm. When the system is under-utilized, tasks would then be able to exceed their budgets without unnecessary deadline postponements. Interesting possibilities for slack stealing include the CASH algorithm [22] and the GRUB algorithm [23].

The analysis in the paper builds on the simplified assumption that all communication (including interrupt handlers and I/O) takes zero time. Possibilities for more detailed analysis of interrupt times under EDF scheduling are found in [1] (“mixed scheduling”) and in [24]. It would also be interesting to develop a variant of the control server for distributed systems. Components of the same controller could then be located at different nodes in a network. The communication times would now have to be taken into account, and a suitable network scheduling policy would have to be used.

The proposed server is based on EDF scheduling. Unfortunately, very few commercial real-time kernels support EDF scheduling today. For backwards compatibility and industrial acceptance, it would be useful to develop a version of the control server

which is based on priority-based scheduling. In fact, aperiodic task scheduling servers were originally developed for fixed-priority systems, [25], and a wealth of other algorithms have been developed since. Since EDF is an optimal scheduling policy, it cannot be expected that all results carry over to a fixed-priority setting.

Finally, automatic tuning of the control server parameters should be studied. It is often very hard to obtain a good estimates of the worst-case execution times of tasks. Using an on-line feedback mechanism, the segment lengths could be automatically adjusted such that overruns occur optimally often. Also, the division of the CPU among several competing control tasks could be performed by a feedback scheduler, as proposed in [21].

A Cost Calculation in Codesign Example 1

The delayed integrator process can be written

$$\frac{dx(t)}{dt} = u(t-L) + v_c(t), \qquad 0 \le L \le T, \tag{13}$$

where L is the input-output latency, and T is the sampling interval. The cost function to be minimized can be written

$$J = \frac{1}{T}\,\mathbf{E}\left\{\int_0^T x^2(t)\,dt\right\}. \tag{14}$$

Inserting the process description (13) into (14) gives

$$\begin{aligned} J = \frac{1}{T}\,\mathbf{E}\Bigg\{ & \int_0^L \left(x(kT) + tu(kT-T) + \int_0^t v_c(s)\,ds\right)^2 dt \\ & + \int_L^T \Bigg(x(kT) + Lu(kT-T) + (t-L)u(kT) \\ & \qquad + \int_0^t v_c(s)\,ds\Bigg)^2 dt\Bigg\} \\ = \frac{1}{T}\,\mathbf{E}\left\{ \begin{bmatrix} x(kT) \\ u(kT-T) \\ u(kT) \end{bmatrix}^T \begin{bmatrix} Q_1 & Q_{12} \\ Q_{12}^T & Q_2 \end{bmatrix} \begin{bmatrix} x(kT) \\ u(kT-T) \\ u(kT) \end{bmatrix} \right\} \\ + J_{samp}, \end{aligned} \tag{15}$$

where

$$Q_1 = \begin{bmatrix} T & \frac{L^2}{2} + (T-L)L \\ \frac{L^2}{2} + (T-L)L & \frac{L^3}{3} + (T-L)L^2 \end{bmatrix},$$

$$Q_{12} = \begin{bmatrix} \frac{(T-L)^2}{2} \\ \frac{(T-L)^2}{2}L \end{bmatrix}, \quad Q_2 = \frac{(T-L)^3}{3},$$

$$J_{samp} = \frac{1}{T}\int_0^T \int_0^t 1 \, ds \, dt = T/2.$$

Sampling the process (13) with the interval T gives

$$\begin{bmatrix} x(kT+T) \\ u(kT) \end{bmatrix} = \Phi \begin{bmatrix} x(kT) \\ u(kT-T) \end{bmatrix} + \Gamma u(kT) + v(kT), \tag{16}$$

where

$$\Phi = \begin{bmatrix} 1 & L \\ 0 & 0 \end{bmatrix}, \qquad \Gamma = \begin{bmatrix} T-L \\ 1 \end{bmatrix},$$

and v is a discrete-time white noise process with covariance

$$R = \begin{bmatrix} T & 0 \\ 0 & 0 \end{bmatrix}.$$

Introduce the positive definite matrix S. The controller that minimizes the cost satisfies the algebraic Riccati equation (e.g. [3])

$$\begin{aligned} S = {} & \Phi^T S \Phi + Q_1 \\ & - \Big(\Phi^T S \Gamma + Q_{12}\Big)\Big(\Gamma^T S \Gamma + Q_2\Big)^{-1}\Big(\Gamma^T S \Phi + Q_{12}^T\Big), \end{aligned} \tag{17}$$

with the solution

$$S = \begin{bmatrix} L + \frac{\sqrt{3}\,T}{6} & \frac{L}{6}\Big(3L - \sqrt{3}\,T\Big) \\ \frac{L}{6}\Big(3L - \sqrt{3}\,T\Big) & \frac{L^3}{3} - \frac{\sqrt{3}\,L^2 T}{6} \end{bmatrix}.$$

The optimal control law is

$$u(kT) = -K \begin{bmatrix} x(kT) \\ u(kT-T) \end{bmatrix}, \tag{18}$$

where

$$\begin{aligned} K &= \Big(Q_2 + \Gamma^T S \Gamma\Big)^{-1}\Big(\Gamma^T S \Phi + Q_{12}^T\Big) \\ &= \begin{bmatrix} \frac{1}{T}\frac{\sqrt{3}+3}{2+\sqrt{3}} & \frac{L}{T}\frac{\sqrt{3}+3}{2+\sqrt{3}} \end{bmatrix}, \end{aligned}$$

and the optimal cost is given by

$$J = \frac{1}{T}\operatorname{tr} SR + J_{samp} = \frac{3+\sqrt{3}}{6}T + L. \tag{19}$$

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 40–61, 1973.

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

[3] K. J. Åström and B. Wittenmark, *Computer-Controlled Systems*. Prentice Hall, 1997.

[4] N. Audsley, K. Tindell, and A. Burns, "The end of the line for static cyclic scheduling," in *Proc. 5th Euromicro Workshop on Real-Time Systems*, 1993.

[5] O. Redell and M. Sanfridson, "Exact best-case response time analysis of fixed priority scheduled tasks," in *Proc. 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[6] K. Jeffay and S. Goddard, "Rate-based resource allocation models for embedded systems," in *Proc. First International Workshop on Embedded Software*, 2001.

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

[8] A. Parekh and R. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single node case," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 344–357, 1993.

[9] M. Caccamo, G. Buttazzo, and L. Sha, "Elastic feedback control," in *Proc. 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000, pp. 121–128.

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

[11] W. Halang, "Achieving jitter-free and predictable real-time control by accurately timed computer peripherals," *Control Engineering Practice*, vol. 1, no. 6, pp. 979–987, 1993.

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

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

[14] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[15] J. Liu and E. Lee, "Timed multitasking for real-time embedded software," *IEEE Control Systems Magazine*, vol. 23, no. 1, Feb. 2003.

[16] L. Abeni, "Server mechanisms for multimedia applications," Scuola Superiore S. Anna, Pisa, Italy, Tech. Rep. RETIS TR98-01, 1998.

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

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

[19] I. Crnkovic and M. Larsson, Eds., *Building Relable Component-Based Software Systems*. Artech House Publishers, 2002.

[20] L. Andersson and A. Blomdell, "A real-time programming environment and a real-time kernel," in *National Swedish Symposium on Real-Time Systems*, ser. Technical Report No 30 1991-06-21, L. Asplund, Ed. Uppsala, Sweden: Dept. of Computer Systems, Uppsala University, 1991.

[21] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén, "Feedback-feedforward scheduling of control tasks," *Real-Time Systems*, vol. 23, no. 1, July 2002.

[22] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *Proc. IEEE Real-Time Systems Symposium*, Orlando, Florida, 2000.

[23] G. Lipari and S. Baruah, "Greedy reclamation of unused bandwidth in constant-bandwidth servers," in *Proc. Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, 2000.

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

[25] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems*, vol. 1, no. 1, 1989.