The Control Server: A Computational Model for Real-Time Control Tasks

Cervin, Anton; Eker, Johan

Published in:
Proceedings 15th Euromicro Conference on Real-Time Systems, 2003.

DOI:
10.1109/EMRTS.2003.1212734

2003

The Control Server: A Computational Model for Real-Time Control Tasks

Anton Cervin

Department of Automatic Control
Lund Institute of Technology
Box 118, SE-221 00 Lund, Sweden
anton@control.lth.se

Johan Eker

Research Department
Ericsson Mobile Platforms AB
SE-221 83 Lund, Sweden
johan.eker@emp.ericsson.se

Abstract

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

1. Introduction

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 (in short, *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.

In the seminal Liu and Layland paper [17], 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 scheduling and 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 (more on this later). 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., [14, 4]). 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., [5, 21]), but this information is still not sufficient to accurately predict the performance of the controllers. Furthermore, as argued in [12], 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

The computational model we propose combines elements from the synchronized I/O model of Giotto [11] with the CPU resource reservation model of the constant bandwidth server (CBS) [1]. 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. 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

Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)
0-7695-1936-9/03 $17.00 © 2003 IEEE

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 [20]. 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.

In the end, we believe that the model will be a suitable platform for adaptation to varying task sets and CPU loads, i.e., feedback scheduling. As new control tasks are activated or old controllers change mode, the computing resources should be redistributed to provide optimal control performance for the overall system. This topic will be treated in subsequent papers.

1.2 Related Work

Giotto [11] 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, a computational domain called Timed Multitasking (TM) has been developed [18]. 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.

The Constant Bandwidth Server [1] was originally proposed as a means to bound the utilization of soft real-time tasks with varying or unknown computational demands. A variant called CBS$^{\text{hd}}$ was introduced to schedule control tasks with varying execution times in [7]. The idea was to extend the sampling period of the controller by adding small chunks of budget to the task in the event of an overrun. The problems of I/O jitter and latency were not considered, however.

The idea of reducing jitter using dedicated, high-priority tasks or interrupts handlers for input and output operations has been proposed many times before, e.g., [19, 14, 8, 2].

1.3 Outline

The rest of this paper is outlined as follows. In the next section, the model is stated in more formal terms. Section 3 deals with the control and scheduling codesign problem. Section 4 discusses the possibility of viewing control tasks as real-time components. The model has been implemented in a real-time kernel and this is reported in Section 5. The results of some experiments on a control application are given in Section 6. Finally, Section 7 gives the conclusions and suggestions for future work.

2. The Model

The Control Server (CS) model assumes preemptive deadline scheduling of tasks in a uniprocessor system. To guarantee isolation, all tasks in the system must belong to either one of two categories:

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

2.1 CS Tasks

A CS task τ_i is described by

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

Associated with each segment S_i^j are

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

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

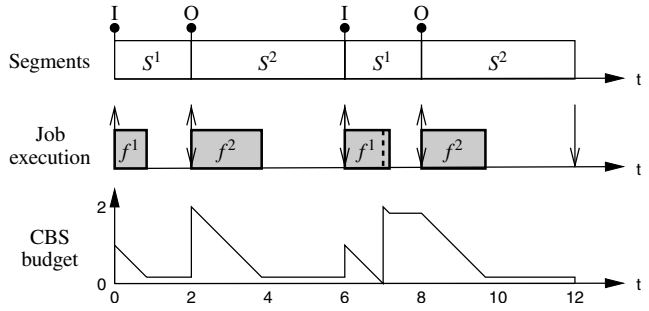

Figure 1 Example of a CS task executing alone. The up arrows indicate job releases and the down arrows indicate deadlines. The overrun at $t = 7$ causes the deadline to be postponed to the end of the next segment.

executing f_i^j is released. At the end of the segment, i.e., when $t = \phi_i + \sum_{k=1}^{j} l_i^k \pmod{T_i}$, the outputs O_i^j are written.

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

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

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

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

The rules are somewhat simplified compared to the original CBS rules [1] due to the predictable pattern of release times and deadlines. The only real difference from an ordinary CBS is that here a "dynamic server period", equal to the current segment length, $l_i^{m_i}$, is used.

Figure 1 shows an example of a CS task with two segments executing alone. This is a typical model of a control algorithm, which has been split into 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.

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

2.2 Communication and Synchronization

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

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

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

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

2.3 Scheduling Properties

From a schedulability point of view, a CS task with the CPU share U_i is equivalent to a CBS server with the bandwidth U_i. By postponing the deadline when the budget is exhausted, the loading factor of the jobs served by the CBS can never exceed U_i. The same argument holds for the modified CBS used in the CS model. A set of CBS and CS tasks is thus schedulable if and only if

$$\sum U_i \leq 1.$$

If the segment lengths of a CS 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 Eq. (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

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

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

$$J = J(T, L, j). \tag{3}$$

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

$$\begin{aligned} T &= \sum l^k = \sum C^k / U, \\ L &= l^1 = C^1 / U, \\ j &= 0. \end{aligned} \tag{4}$$

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, e.g., the Jitterbug toolbox [15].

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.

The disadvantage of eliminating the jitter is that the latency may increase, and latency also has a negative impact on the control performance. Our model, however, allows a control algorithm to be split into segments, and this can be used to reduce the latency. The importance of this feature is illustrated in the first example below.

3.1 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 [9].) 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 (assuming a fixed level of control performance).

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

3.2 Example 2: Optimal Period Selection

In this example we study the problem of optimal period selection for a set of control loops. This type of codesign problem first appeared in [22]. In that paper, however, the scheduling-induced latency and jitter was ignored.

Suppose for instance that we want to control three identical integrator processes (6). The assumed design goal is to select sampling periods T_1, T_2, T_3 such that a weighted sum of the cost functions, e.g.,

$$J_{tot} = J(T_1, L_1) + 2J(T_2, L_2) + 3J(T_3, L_3), \qquad (9)$$

is minimized subject to the utilization constraint

$$U = \frac{C}{T_1} + \frac{C}{T_2} + \frac{C}{T_3} \leq 1. \qquad (10)$$

Here, C is the (constant) execution time of the control algorithm. Dividing the algorithm into two segments, our model will imply the same relative latency $a = L_i/T_i$ for all controllers. Using (8) the objective function (9) can be written

$$J_{tot} = \left(\frac{3+\sqrt{3}}{6} + a\right)(T_1 + 2T_2 + 3T_3), \qquad (11)$$

and the solution to the optimization problem is

$$T_1 = b, \quad T_2 = b/\sqrt{2}, \quad T_3 = b/\sqrt{3} \qquad (12)$$

where $b = C(1+\sqrt{2}+\sqrt{3})$. (For more general problems numerical optimization must be performed.) Contrary to [22] (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.

3.3 Example 3: Allowing Overruns

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

Again consider the integrator controller. For simplicity, it is assumed that the controller is implemented as a single segment, i.e., we have $L = T$ if no overrun occurs, and that the assigned CPU share is $U = 1$. Now assume that the execution time of the controller is given by the probability density function in Figure 2. Choosing a period less than the WCET means that some outputs will be missed and that the actual latency will vary randomly between T, $2T$, $3T$, etc., according to a Markov chain. The resulting control performance for such a model can be computed using the Jitterbug toolbox [15]. In Figure 3 the cost (7) has been computed for different values of the task period. The optimal cost $J = 1.67$ is obtained for $T = 0.76$. For that period, overruns will occur in 9% of the periods (introducing a latency of $2T$ or more). The example shows that our model can be used to "cut the tail" off execution time distributions with safe and predictable results.

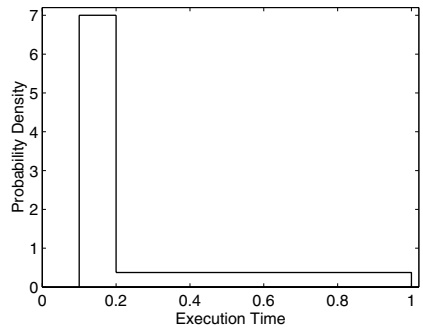

Figure 2 Assumed execution time probability density function of the integrator controller.

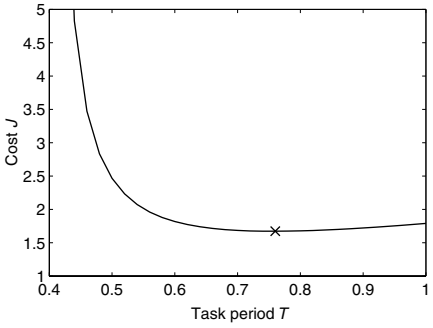

Figure 3 Cost as a function of the task period for the integrator controller with varying execution time.

4. CS Tasks as Real-Time Components

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

Consider for instance the PID (proportional-integral-derivative) controller component in Figure 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 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 and could for instance look like this:

- Algorithm: $u = K(r-y) + \ldots$
- Parameters: $U, K, \ldots$

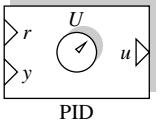

Figure 4 A PID controller component. The U knob determines both the schedulability and the control performance.

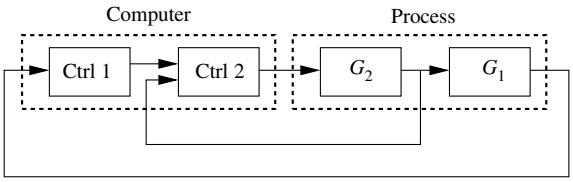

Figure 5 Cascaded controller structure.

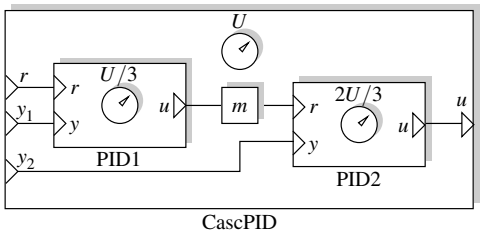

Figure 6 A cascaded PID controller component.

- $C = 1$ ms (on a given processor)
- $T = C/U$
- $L = T/4$
- $J = J(U)$ (given as function or diagram)

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

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

The schedulability and performance of the cascaded controller will, again, only depend on the total assigned CPU share U. The controller will have a predictable input-output pattern, and its performance can be computed using

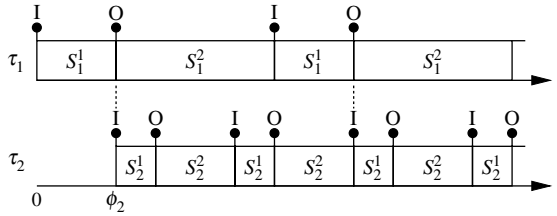

Figure 7 Segment layout in the cascaded PID controller. Task τ_2 is given a phase $\phi_2 = l_1^1$ such that the value written by S_1^1 is immediately read by S_2^1.

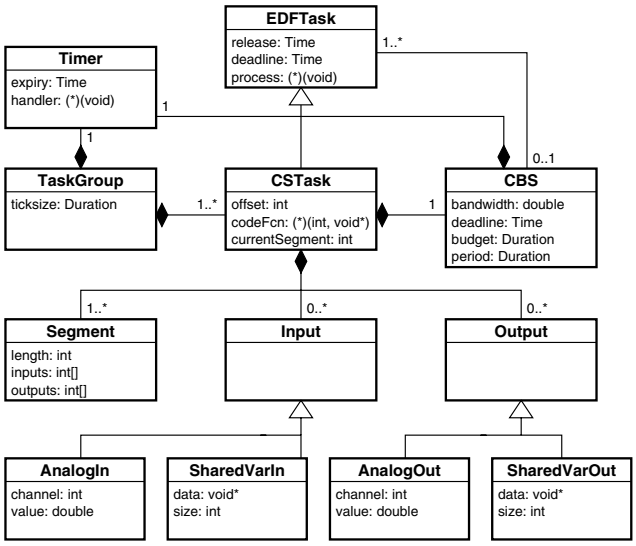

Figure 8 The various data structures in the implementation.

the Jitterbug toolbox [15, 9]. 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.

5. Implementation

The task model has been implemented in the STORK real-time kernel [3], 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. A number of data structures for CBS servers, CS tasks, segments, inputs, and outputs, etc., were introduced, see Figure 8. For synchronization reasons, communicating CS tasks must share a common timebase and are gathered in task groups.

5.1 Task Group Timing

Each task group uses a timer to trigger the reading of inputs, writing of outputs, and release of segments of tasks within the group. The structure of the task group timer interrupt handler is shown in Listing 1. The average execution time of the handler was about 5 μs in the implementation.

Associated with each CS task is a semaphore that is used to handle the release of the segment jobs. Internally, every CS task is implemented as a simple loop, see Listing 2.

5.2 API

The kernel provides a number of primitives for defining task groups, EDF tasks, CS tasks, etc. The code of a CS ta

Listing 1 Pseudocode of 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 (new segment should begin) {
    Read inputs (if any);
    Release segment job (signal semaphore);
  }
}
Determine next wakeup time;
Set up timer;
```

Listing 2 Internal implementation of CS task.

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

written according to a special format illustrated with a PID controller in Listing 3. The kernel primitives `ReadInput` and `WriteOutput` are used to access the inputs and outputs associated with the segment.

6. Control Experiments

Some control experiments were performed on the ball and beam process, see Figure 9. The objective is to move the ball to a given position on the beam. The input to the process is the beam motor voltage, and the outputs are voltages representing the beam angle and the ball position. The process is regulated with a cascaded PID controller, implemented as a single task (in order to keep the example simple). The controller is designed with the sampling interval $T_1 = 40$ ms and has the execution time $C_1 = 20$ ms, thus consuming $U_1 = 0.5$ of the CPU. (To generate a high CPU load, busy cycles were inserted in the task code). The code is divided into two segments: Calculate Output (5 ms) and Update State (15 ms).

Listing 3 PID controller written in Modula-2.

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

Figure 9 The ball and beam process.

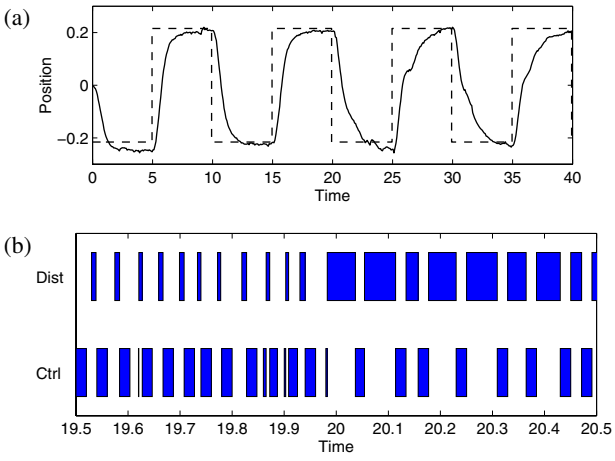

Figure 10 Control experiment under EDF scheduling: (a) control performance, and (b) close-up of execution trace at $t = 20$.

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

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

The result of a control experiment under EDF scheduling is shown in Figure 10. The performance is satisfactory up to $t = 20$, when the sporadic task starts to consume a large part of the CPU time, which disturbs the control task.

In a second experiment, running the tasks under CS scheduling, both tasks were assigned a CPU share of 50%. The experimental results are shown in Figure 11. The controller execution is no longer disturbed by the misbehaving sporadic task, and (not visible in the trace) there is no longer any I/O jitter. The control performance is identical both before and after time $t = 20$.

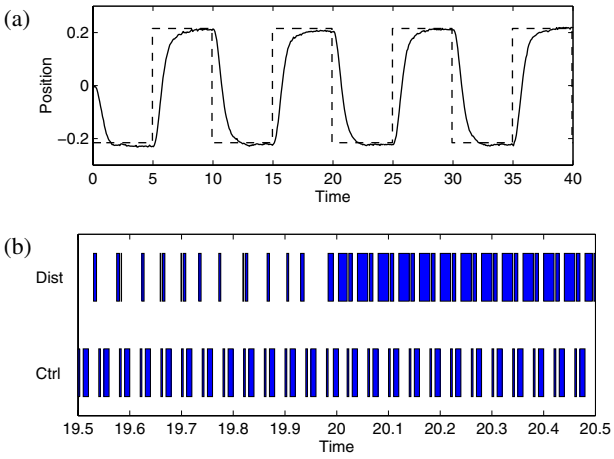

Figure 11 Control experiment under CS scheduling: (a) control performance, and (b) close-up of execution trace at $t = 20$.

7. Conclusion

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

The present work may be extended in several directions. The CBS servers used could be modified to use a slack stealing algorithm such as CASH [6] or GRUB [16]. This could improve the performance further when the system is under-utilized.

We do not account for the interrupt time (including the I/O operation) in the scheduling analysis. Possibilities for more detailed analysis are found in [17] ("mixed scheduling") and in [13].

Another topic that needs further investigation is the overrun handling. How should a controller be designed in order to cope with postponed outputs? Should segments sometimes be aborted?

Also, we would like to exploit the codesign properties of the model in feedback scheduling applications where the goal is to dynamically distribute the available computing resources such that the overall control performance is optimized. In our previous work [10] we did not account for the latency and the jitter in the on-line optimization.

References

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

[2] P. Albertos, A. Crespo, I. Ripoll, M. Vallés, and P. Balbastre. "RT control scheduling to reduce control performance degrading." In *Proc. 39th IEEE Conference on Decision and Control*, Sydney, Australia, 2000.

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

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

[5] 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.

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

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

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

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

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

[11] 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.

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

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

[14] 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.

[15] 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, December 2002.

[16] 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.

[17] C. L. Liu and J. W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment." *Journal of the ACM*, **20:1**, pp. 40–61, 1973.

[18] J. Liu and E. Lee. "Timed multitasking for real-time embedded software." *IEEE Control Systems Magazine*, **23:1**, February 2003.

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

[20] 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*, **1:3**, pp. 344–357, 1993.

[21] 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.

[22] 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*, pp. 13–21, Washington, DC, 1996.

Establishing Timing Requirements and Control Attributes for Control Loops in Real-Time Systems

Iain Bate[1], Peter Nightingale[1], Anton Cervin[2]

1 Department of Computer Science,
University of York, York, YO10 5DD, UK
{ijb, pwn101}@cs.york.ac.uk

2 Department of Automatic Control
Lund Institute of Technology, Lund, Sweden.
anton@control.lth.se

Abstract

Advances in scheduling theory have given designers of control systems greater flexibility over their choice of timing requirements. This could lead to systems becoming more responsive, more flexible and more maintainable. However, experience has shown that engineers find it difficult to exploit these advantages due to the difficulty in determining the "real" timing requirements of systems and therefore the techniques have delivered less benefit than expected. Part of the reason for this is that the models used by engineers when developing systems do not allow for emergent properties such as timing. This paper presents an approach and framework for addressing the problem of identifying an appropriate and valid set of timing requirements and their corresponding control parameters based on a combination of static analysis and simulation.

1 Introduction

This paper addresses the perennial problem of how to identify an appropriate and valid set of timing requirements for a hard real-time system. Over the years, research on real-time systems has evolved techniques which provide greater flexibility in scheduling whilst still providing a means for guaranteeing that timing requirements are met [1, 6]. The increased flexibility was expected to give many benefits, including more efficient use of resources and simpler maintenance of schedules when changes to the control software are made. In addition, maintaining schedules is often a costly and error prone manual process, so these techniques have the potential to offer significant economic as well as engineering benefit.

However, experience has shown that engineers find it difficult to exploit this increased flexibility, and the techniques have delivered less benefit than expected. Based on our own experience and that of others in industry [2, 6, 10], a key reason is an absence of information about the *true* timing requirements which are needed to make best use of the approaches. In many cases current systems are developed with simple timing requirements, such as a timing margin to be achieved. (A timing margin is the amount of usable spare capacity available.) In other cases the timing requirements are largely historic, and are simply expressed in terms of iteration rates which have been proven effective in previous designs. Despite the changing contexts between systems, this strategy is normally successful because the requirements are over conservative, e.g. update rates specified are much faster than needed. Even where more modern control law design environments are used (e.g. Matlab/Simulink [3]), the control models are often produced assuming a particular computational model. For example a 50 ms cycle/20Hz bandwidth is chosen because there is a regular clock tick in the system with a period of 25 ms (i.e. 40 Hz) and therefore it is easier to release tasks at a harmonic of this frequency.

Other techniques such as Shannon's sampling theroem [5] place an upper bound on the sampling period. When the sampling theorem is used, an actual sampling period still needs to be selected as well as other timing attributes such as the deadline of the sampling task, period and deadline of the actuator task, and the maximum separation time between data capture and sensor actuation.

A major contributor to the situation that has arisen is because both the research and practical use of control theory and scheduling theory have largely been carried out in isolation [4]. Thus for example, work on how advanced control regimes, such as H∞ [5], might ease the integration issues between control and software, have received little attention. Other pressures include the move towards model-based development that places greater onus on capturing evidence within the actual models and including low-level implementation details within the models, i.e. emergent properties such as timing.

This paper presents an approach and framework for addressing the problem of identifying an appropriate and valid set of timing requirements in order that the best use can be made of the advances in scheduling theory. The paper is an extension to previous work [15] that adds greater traceability back to the system's objectives using an argumentation technique to target the evaluation used in the framework, and for evaluation purposes using Jitterbug to perform static analysis [11] and the use of scenario-based assessment to determine the extent to which the system copes with other situations - e.g. changes to the system, errors in models and measurements, and random failures.

The approach taken is to first establish the objectives of importance (based on argumentation techniques used in the critical systems domain) and then use component-based models that allow for emergent properties of systems (in this case timing) so that the models are more representative of how an actual system would actually behave. Then, a genetic algorithm is used to explore the design space in-order to identify timing requirements and corresponding control parameters which enable objectives such as control stability to be achieved, thus deriving and validating the requirements against more realistic properties of the control system. When valid combinations of parameters are found, the framework

Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)
0-7695-1936-9/03 $17.00 © 2003 IEEE

produces evidence that the solution is appropriate in a traceable manner via static analysis and test.

The advantage of using genetic algorithms instead of traditional model-based design approaches, such as frequency domain loop shaping [5], include it allows many properties and effects to be considered at the same time and their demands on the system to be traded-off against one another [9].

The work presented here is intended for use in a range of control problems, but is illustrated with the PID (Proportional Integral Differential) control approach [5].

The rest of the paper is structured as follows. Section 2 gives further background on the control techniques to be used in the context of this work. It also provides a technical motivation (as opposed to the "economic" motivation outlined above) for seeking a systematic approach to deriving timing requirements. Section 3 gives an overview of an argument that assesses the desirable properties of a control system scheduled on a computer and evolves an experimental method to show the properties are met. Section 4 presents the framework, and the costs of evaluating the requirements. Section 5 contains a case study which have been used to evaluate the approach, as well as presenting a discussion of how the resulting timing requirements may be used. Finally, section 6 gives a summary and suggests possible future developments for the work.

2 Background and Motivation

All scheduling approaches require a minimum set of information about timing requirements so that an appropriate scheduler can be produced. For most scheduling approaches the minimum set of information is the deadline and period of tasks [6, 7]. This section explains why these requirements are important in the context of PID loops and how they can be generated by considering basic control properties.

2.1 PID Loop

The main purpose of a PID loop is to ensure the response to inputs is sufficiently fast whilst maintaining the stability, accuracy and limits on data. Figure 1 depicts a typical PID loop used to control the operation of a plant as part of a control system. The Figure shows the key aspects and components of the controller – e.g. there is only one input and one output.

In its simplest form, a continuous ideal domain representation, the output of the PID loop is the plant input. The control system input is the difference between the input demand (denoted by *I*), which is the desired plant state, and the plant's actual output (denoted by *O*) and it is referred to as the error, (denoted by *E*). The continuous and discrete forms of the PID loop are given in Equation 2 and Equation 4 (current sample denoted by k) respectively.

$$E(t) = O(t) - I(t) \quad \textbf{Equation 1}$$

$$O(t) = K_P E(t) + K_I \int E(t)dt + K_D \frac{dE(t)}{dt} \quad \textbf{Equation 2}$$

$$E(k) = O(k) - I(k) \quad \textbf{Equation 3}$$

$$O(k) = K_P E(k) + K_I \sum_{j=1}^{k} E(j) + K_D \left[E(k) - E(k-1)\right] \quad \textbf{Equation 4}$$

In the computer-based approach, the *Input Demand* (e.g. pilot stick position) and the *Actual Plant Output* (e.g. aircraft's flap position) are usually analogue signals. The computer performs the rest of the processing in the digital domain. Converters are used to sample the analogue signals, e.g. to produce the *Error* input, and then converted back to analogue values at the output. Converting back to an analogue signal is often referred to as digital to analogue conversion, de-sampling or actuation. In order to give better control over jitter, the functionality that needs to be performed in software is normally split into three separate tasks – sampling, calculation and actuation [4, 6, 7].

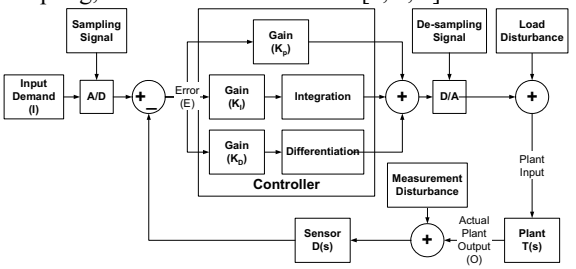

Figure 1 – Typical PID Loop

In industrial practice it is common for a controller to be developed as a continuous system based on the system's response in the frequency domain. Often modelling packages or special purpose plant simulations are used to validate the requirements. If a computer-based implementation is to be used, then once the requirements have been established in the continuous domain they are converted to the discrete domain. Typically the conversion involves calculating the PID loop gains (K_P, K_I, K_D) based on the assumption that a constant sampling period is used. This means the conversion is performed based on an idealised model of the computer system. The conversions for the PID loop gains are give in Table 1. In other words the conversion uses unrealistic assumptions, e.g. infinite processing bandwidth and zero jitter in sampling the inputs (jitter is the variation in time when an action occurs between one cycle of the controller and the next). In addition, "real" systems have errors through effects such as measurement disturbance, load disturbance and plant error. These are also represented in Figure 1.

Parameter in Continuous Domain	Discretisation Formula for a Sampling Period of T
K_P	K_P
K_I	$T.K_I$
K_D	K_D / T

Table 1 – Conversion from Continuous to Discrete

The approach presented in this paper addresses this shortcoming by taking into account the constraints of real computer systems, and thus enables valid and realistic requirements to be produced. To explain how this is done the rest of this section explains in more detail the relationship between computational properties such as jitter and control properties such as stability.

2.2 Scheduling Properties

It is, of course, essential that the sampling, core functions and de-sampling tasks are executed in that