A Computational Model for Real-Time Control Tasks

(extended abstract)

Anton Cervin and Johan Eker

Department of Automatic Control
Lund Institute of Technology
Sweden

Summary

We propose a computational model for real-time control tasks which combines ideas from the synchronized I/O model of Giotto [Henzinger *et al.*, 2001] with the CPU resource reservation model of the Constant Bandwidth Server (CBS) [Abeni and Buttazzo, 1998]. The goal of the model is to facilitate co-design of flexible real-time control systems. In particular, the model should provide

- a simple interface between control design and real-time design
- minimal sampling jitter and output jitter
- short input-output latency
- isolation between unrelated tasks (including non-control tasks)
- predictable control and real-time behavior during overruns
- a possibility to combine several tasks (components) into a new task (component) with predictable control and real-time behavior
- a possibility to adapt to changing CPU load to provide optimal control performance

Background

Traditional scheduling models give poor support for the design of multi-threaded control systems. One difficulty lies in the nonlinearity in dynamic scheduling mechanisms such as rate-monotonic or earliest-deadline-first scheduling: a small change in a task parameter, e.g. period, execution time, deadline, priority, etc., 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 input-output 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. On the other hand, static scheduling may simplify the control design problem (because of the predictable delays) but cannot handle overruns or adaptation to changing CPU load.

Model Overview

Combining the best of two worlds, we propose the use of static (time-triggered) scheduling for I/O (communication with the environment and other tasks), and dynamic scheduling (based on CBS) for all computations in between. We use CBS servers to make each task appear as if it were running at a given fraction of the CPU speed. This means that a task that is given x% of the CPU will appear as if it were running as a lone thread on a machine with only x% of the original capacity. The task is only allowed to communicate with the environment or other tasks at specified *interaction points*. A key property of the model is that, for each task, time only has to progress linearly with respect to its interaction points. Between points, the CBS scheduling algorithm (which is based on EDF) is used to efficiently schedule the pending computations. All I/O is handled by the kernel (which executes at the highest priority) and is hence not prone to jitter.

Task Model

Each task τ_i has a period T_i and a CPU share U_i. To facilitate short latencies (contrary to e.g. Giotto which imposes a one-sample delay) the task is divided into $l_i \geq 1$ segments $S_i^1, \ldots, S_i^{l_i}$. The lengths of the segments are expressed as fractions of the task period, $s_i^1, \ldots, s_i^{l_i}$, $\sum_{j=1}^{l_i} s_i^j = 1$. Within a segment S_i^j, the task is guaranteed (by the CBS scheduling algorithm) to be able to execute for a time $s_i^j U_i T_i$. The beginning of a segment may be declared as a *read point*, where shared variables or physical inputs are read. The end of a segment may be declared as a *write point*, where shared variables or physical outputs are written.

An example of a task with two segments in shown in Figure 1. In the first segment, an input is read, some computations are performed, and an output is written. In the second segment, some more computations are performed. This is a typical model of a controller whose computations can often be split into two parts, Calculate Output and Update State, in order to minimize the input-output latency. In this example, the latency will be equal to $s^1 T$. If the worst-case execution time of the two segments are bounded by $s^1 UT$ and $s^2 UT$ respectively, the controller will have a perfectly predictable run-time behavior, regardless of the execution of the other tasks in the system.

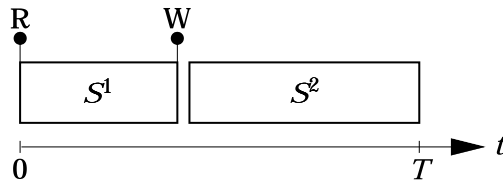

Figure 1 An example of a task divided into two segments. The first segment has two interaction points: a read point (R) in the beginning and a write point (W) at the end.

Task are allowed to communicate with each other through shared variables. To facilitate minimal end-to-end latencies, a task τ_i can be given an optional release offset O_i (which defaults to zero). If the write point of one task should occur at the same time as the read point of another task, the write action is guaranteed to take place first.

An example of two communicating tasks are shown in Figure 2. This could model for instance a cascade controller, where the inner controller (τ_2) executes twice as frequently as the outer controller (τ_1). After executing segment S_1^1, the outer controller writes a control signal to a shared variable which will be read and used as a reference value in the inner controller. To minimize the end-to-end delay, task τ_2 is given an offset $O_2 = s_1^1 T_1$.

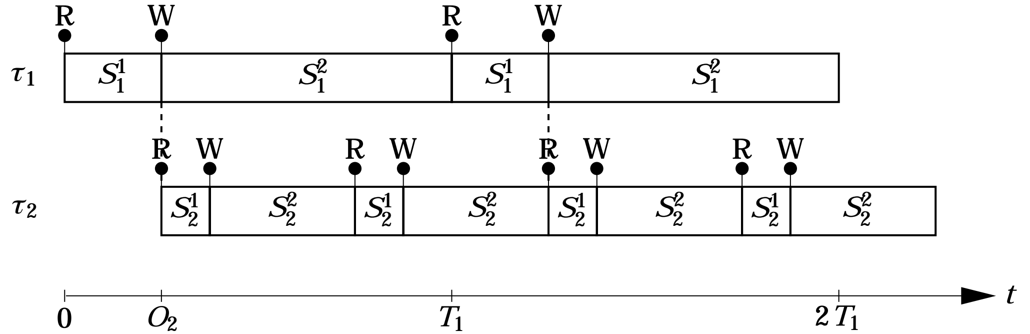

Figure 2 Communicating controller tasks. Task τ_2 is given an offset such that the control signal written by S_1^1 is immediately read by S_2^1.

Notice that, taken together, τ_1 and τ_2 can be viewed as a new task (or component) with the CPU share $U_1 + U_2$.

Control and Scheduling Co-Design

The great advantage of the proposed computational model is that we get a simple control and scheduling co-design problem, which still takes the implementation into account. If we ignore the overhead associated with the I/O operations and the CBS servers, schedulability of the task set is simply a function of the total CPU demand $U = \sum_i U_i$. Just as for plain EDF, the system is schedulable iff $U \leq 1$ [Abeni and Buttazzo, 1998].

The optimal performance of each controller is also just a function of its CPU share U_i[1]. Given a share U_i, it is possible to select a task period T_i and segment sizes $s_i^1 \ldots s_i^{l_i}$ such that the control performance is optimized. In the case of constant execution times $C_i^1 \ldots C_i^{l_i}$ of the segments, the choices that minimize the period and the latency without missing any deadlines are given by $T_i = \frac{1}{U_i} \sum_{j=1}^{l_i} C_i^j$ and $s_i^j = \frac{1}{U_i} C_i^j$. The latency of the controller will be known and can be compensated for in the control design.

For linear controllers, it is possible to compute a quadratic performance index for the controller as a function of the sampling period, the input-output latency, and the jitter [Lincoln and Cervin, 2000]. In our model, however, the jitter is eliminated and the optimal period and latency is determined by the CPU share U.

To summarize, in our model, both schedulability and control performance are functions of the CPU shares U_i only. Each controller can be designed independently of the others, compensating for a known latency.

An Example

As an example, we will look closer at the cascaded controller mentioned above. The principle of the controller is shown in Figure 3. The cascaded

[1]Assuming that shorter period implies better performance and also that shorter latency implies better performance.

controller consists of two components: the outer controller (Ctrl 1) and the inner controller (Ctrl 2). The inner controller handles the fast part of the process dynamics (G_ϕ), and is assumed to have twice the sampling frequency of the outer controller which handles the slower dynamics (G_x).

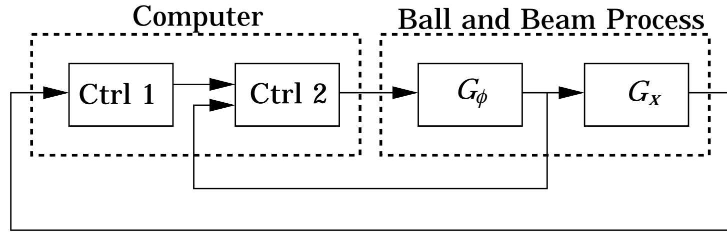

Figure 3 A cascaded controller structure. The inner controller has twice the sampling frequency of the outer controller.

Both controllers execute the same PID control algorithm. Each period, the controller should

1. Read the measurement signal and the reference value
2. Calculate control signal (execution time: 0.4 ms)
3. Write the control signal
4. Update the controller state, log data, etc. (execution time: 1.2 ms)

In the our model, each controller is modeled as a task τ_i with two segments, Calculate Output (S_i^1) and Update State (S_i^2). To use the computing resources optimally, the lengths of the segments are proportional to the execution times: $s_i^1 = 1/4$ and $s_i^2 = 3/4$. There is a read point at the beginning of Calculate Output, where the reference value and the measurement value are read, and there is a write point at the end of Calculate Output where the control signal is written. The control signal of Ctrl 1 acts as the reference signal to Ctrl 2, so this is a shared variable.

To minimize the control latencies, Ctrl 2 should execute its first Calculate Output right after Ctrl 1 has finished its Calculate Output. This is achieved by assigning the offsets $O_1 = 0$ and $O_2 = s_1^1 T_1$.

Now, suppose that 60 % of the total CPU resources are available for the cascade controller, while the rest is needed for other tasks. We would then assign the shares $U_1 = 0.2$ and $U_2 = 0.4$ to the controller tasks. To use the resources fully, we assign the periods $T_1 = 8$ ms and $T_2 = 4$ ms. Note that, should we choose to assign more or less resources to the cascade controller, these numbers are simply rescaled by a factor.

The example has been implemented in the TRUETIME simulator [Henriksson *et al.*, 2002]. A schedule simulation with the two control tasks and a kernel task handling the I/O (now more realistically assumed to take 0.1 ms to execute) is shown in Figure 4. Also shown is the linear CPU time as perceived by the tasks, together with the actual CPU time, and the interaction points.

Note that the remaining 40 % of the CPU can be used by other tasks, without affecting the behavior of the controllers. The actual CBS schedule will change, but the interaction points will remain the same and the segments will finish their computations in time.

Again, note that τ_1 and τ_2 can be viewed as *one component* that consumes 60 % of the CPU. Using JITTERBUG [Lincoln and Cervin, 2000] it

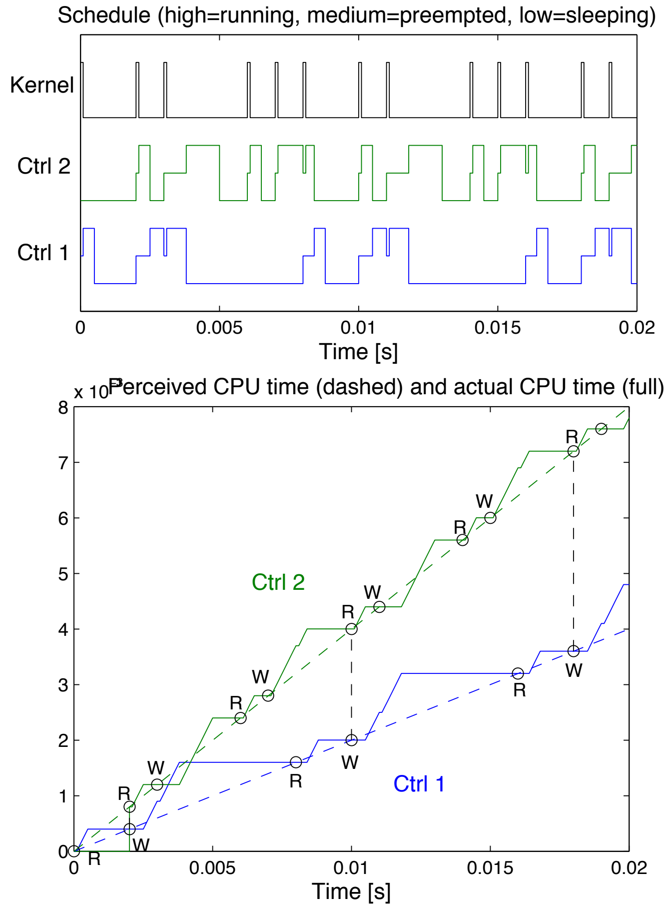

Figure 4 *Above:* The task schedule generated by CBS servers. *Below:* Perceived linear CPU time, actual CPU time, and interaction points for the controller tasks. Notice that the interaction points are always located on the linear timelines.

is possible to compute a quadratic control performance index also for this type of multi-rate controller. As an example, in Figure 5, a cost criterion on the form

$$J(U) = \lim_{T\to\infty} \frac{1}{T} \mathop{\mathbf{E}}_{v,w} \left\{ \int_0^T \Big(x^T(t)\, Q_1\, x(t) + u^T(t)\, Q_2\, u(t) \Big)\, dt \right\}$$

has been computed for different values of the total CPU share $U = U_1 + U_2$.

Future Work

Much work remains before the model will be useful. Some things are outlined below.

Overrun Handling If a control task does not finish its computations in time, the CBS server will postpone its deadline to prevent other tasks from suffering. This can mean the controller will have to continue its computations in the next period. There is then the choice between finishing the computation that was started in the last period, or to read a new input and restart the computation. Calculations have shown that the best choice will be different from controller to controller. In each case, the situation must be handled differently in the controller code.

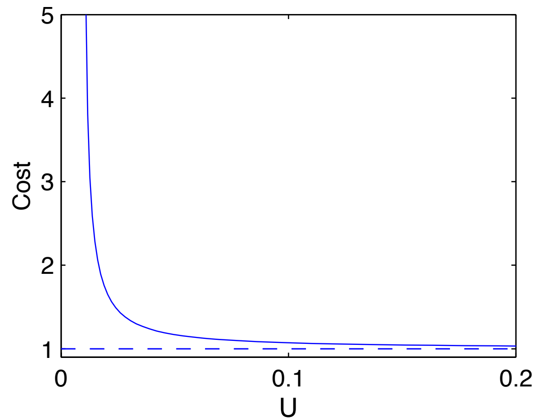

Figure 5 Cost vs total CPU share for the cascaded controller. The dashed line indicates the performance of a continuous-time controller (infinite CPU resources).

Adaptive Resource Distribution When the CPU demand of the tasks in the system change (e.g. when new tasks enter the system or when control tasks switch modes) the CPU resources should be redistributed so that the overall control performance is optimized. Changing the CPU shares of the CBS servers at run-time can lead to transients, however. Perhaps the transients can be tolerated, but this must be investigated further.

Implementation Several issues retaining to the implementation also have to be resolved. As a first step, we will implement the full model in the TRUETIME simulator [Henriksson *et al.*, 2002].

Conclusions

We have outlined a computational model for real-time control tasks, with the primary goal of simplifying the control and scheduling co-design problem. Using a model where time appears to progress linearly for all components (when viewed in the interaction points only) the low-level scheduling details can be eliminated from the design process.

References

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

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

Henzinger, T. A., B. Horowitz, and C. M. Kirsch (2001): "Giotto: A time-triggered language for embedded programming." In *Proceedings of EMSOFT 2001*.

Lincoln, B. and A. Cervin (2000): "Jitterbug: A tool for analysis of real-time control performance." Submitted to the 2002 Conference on Decision and Control.