Proceedings of the 2006 IEEE
Conference on Computer Aided Control Systems Design
Munich, Germany, October 4-6, 2006

Control Loop Timing Analysis Using TrueTime and Jitterbug

A. Cervin, K.-E. Årzén, D. Henriksson
Department of Automatic Control, LTH
Lund University, Sweden

M. Lluesma, P. Balbastre, I. Ripoll, A. Crespo
Department of Computer Engineering
Technical University of Valencia, Spain

Abstract— **A modern control system is typically implemented as a multitasking software application executing in a real-time operating system. If the computer load is high, the controller will experience delays and jitter, which in turn degrade the control performance. Arguing for an integrated design approach, the paper describes two computer tools for implementation-aware control analysis: TrueTime and Jitterbug. An example is given where the tools are used together to evaluate the performance of various control task implementations.**

I. Introduction

The design of computer-based control systems is traditionally based on the principle of *separation of concerns*. By providing a suitable implementation framework, the concerns of interest to the control engineers can be separated from the concerns related to the computing and communication platform on which the controller is implemented. The assumptions underlying the separation are that the implementation platform is able to provide deterministic (often periodic) sampling, negligible or constant input-output latencies, and floating point arithmetics. Separation of concerns has several advantages. It allows the control engineers to focus on the pure control design without having to worry about how the control system eventually is implemented. At the same time, it has allowed the real-time computing community to focus on development of scheduling theory and computational models that makes it possible to fulfil the assumptions, without any need to understand what impact the scheduling has on the stability and performance of the plant under control.

However, in practice it is not always so easy to obtain this separation of concerns. In embedded applications, computing and communication resources are often severely limited, and it is therefore desirable to maximize their utilization. The priority-based scheduling used by most real-time operating systems introduces temporal nondeterminism. The schedulability theory that is available is mostly concerned with worst-case scenarios. Providing worst-case guarantees often implies over-provisioning of resources and low average-case utilization. Time-driven static scheduling is more deterministic but less efficient from an utilization perspective. A consequence of these problems is that the separation is often incomplete and therefore the control issues and the computing issues interact, causing temporal nondeterminism in the form of jitter in sampling and latencies. More specifically, scheduling can cause jitter in both the sampling operation and in the actuation operation, as illustrated in Fig. 1.

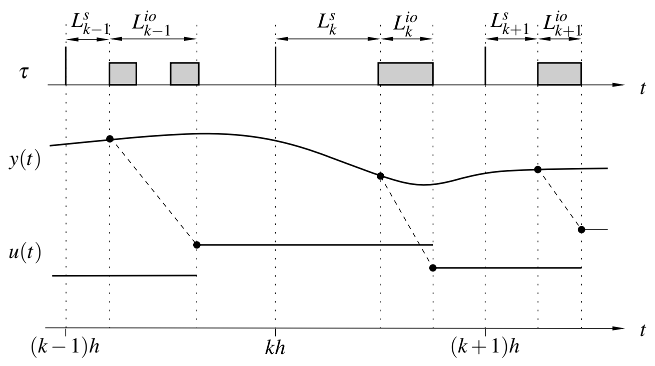


Fig. 1. Controller timing with scheduling-induced sampling latency L^s and input-output latency L^{io}. Since the latencies vary from sample to sample, there will be *jitter* in the input and output operations.

The main drawback with separations of concerns is that it often gives rise to worse control performance than what can be achieved if the design of the control and real-time computing and communication parts are integrated. Better performance can be achieved if a co-design approach is adopted, where the control system is designed taking the resource constraints into account, and where the real-time computing and scheduling is designed with the control performance in mind. The resulting *implementation-aware control systems* are better suited to meet the requirements of embedded and networked applications.

A drawback with integration-based design is the increased complexity. Therefore tool support is particularly important. This paper describes two such tools: Jitterbug and TrueTime. TrueTime (Section II) can be used to simulate how the temporal aspects of real-time kernels and network communication influence the timing of a control loop. Given the timing information of a control loop expressed in terms of latency distributions, Jitterbug (Section III) can be used to calculate the control performance, expressed in terms of a quadratic cost function. The paper also presents an example (Section IV) where the two tools have been used together to evaluate various controller task models (including a new task model) with respect to the obtained control performance. The combined usage of the tools is illustrated in Fig. 2. The input

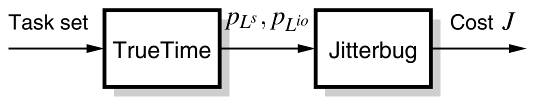
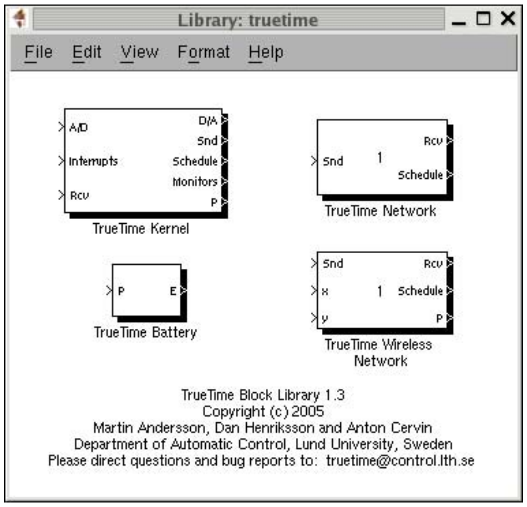


Fig. 2. Possible combined usage of the tools.



TrueTime Block Library 1.3
Copyright (c) 2005
Martin Andersson, Dan Henriksson and Anton Cervin
Department of Automatic Control, Lund University, Sweden
Please direct questions and bug reports to: truetime@control.lth.se

Fig. 3. The TrueTime block library. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

to TrueTime is a task set with specified periods, execution time distributions and a task model/scheduling policy. Using the logging functionality of TrueTime, estimates of the latency probability distributions p_{L^s} and $p_{L^{io}}$ are obtained. These are then used together with a linear controller/plant model in Jitterbug to estimate the resulting performance index J.

An extensive survey of related work within the area of co-design tools for embedded control systems is given in the companion paper [12].

II. TrueTime

TrueTime [4], [8], [9] is a MATLAB/Simulink-based tool that facilitates simulation of the temporal behavior of a multitasking real-time kernel executing controller tasks. The tasks are controlling plants that are modeled as ordinary continuous-time Simulink blocks. TrueTime also makes it possible to simulate simple models of communication networks and their influence on networked control loops.

TrueTime provides a number of Simulink blocks, which are shown in Fig. 3. The kernel block is event-driven and executes code that models, e.g., I/O tasks, control algorithms, and network interfaces. The scheduling policy of the individual kernel blocks is arbitrary and can be decided by the user. Likewise, in the network, messages are sent and received according to a chosen network model.

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

TrueTime can be used as an experimental platform for research on dynamic real-time control systems. For instance, it is possible to study compensation schemes that adjust the control algorithm based on measurements of actual timing variations (i.e., to treat the temporal uncertainty as a disturbance and manage it with feedforward or gain scheduling). It is also easy to experiment with more flexible approaches to real-time scheduling of controllers, such as feedback scheduling, see [3]. There, the available CPU or network resources are dynamically distributed according to the current situation (CPU load, the performance of the different loops, etc.) in the system.

A. The Kernel Block

The kernel block is a Simulink S-function that simulates a computer with a simple but flexible real-time kernel, A/D and D/A converters, a network interface, and external interrupt channels. The kernel executes user-defined tasks and interrupt handlers. Internally, the kernel maintains several data structures that are commonly found in a real-time kernel: a ready queue, a time queue, and records for tasks, interrupt handlers, monitors and timers that have been created for the simulation.

An arbitrary number of tasks can be created to run in the TrueTime kernel. Tasks may also be created dynamically as the simulation progresses. Tasks are used to simulate both periodic activities, such as controller and I/O tasks, and aperiodic activities, such as communication tasks and event-driven controllers. Aperiodic tasks are executed by the creation of task instances (jobs). Each task is characterized by a number of static (e.g., relative deadline, period, and priority) and dynamic (e.g., absolute deadline and release time) attributes.

Interrupts may be generated in two ways: externally (associated with the external interrupt channel of the kernel block) or internally (triggered by user-defined timers). When an external or internal interrupt occurs, a user-defined interrupt handler is scheduled to serve the interrupt.

The execution of tasks and interrupt handlers is defined by user-written code functions. These functions can be written either in C++ (for speed) or as MATLAB m-files (for ease of use). Control algorithms may also be defined graphically using ordinary discrete Simulink block diagrams. Simulated execution occurs at three distinct priority levels: the interrupt level (highest priority), the kernel level, and the task level (lowest priority). The execution may be preemptive or non-preemptive; this can be specified individually for each task and interrupt handler.

B. The Network Blocks

TrueTime has two types of network blocks: for wired networks and for wireless networks. The network blocks are event-driven and executes when messages enter or leave the network. When a node tries to transmit a message, a triggering signal is sent to the network block on the corresponding input channel. When the simulated transmission of the message is finished, the network block sends a new triggering signal on the output channel corresponding to the

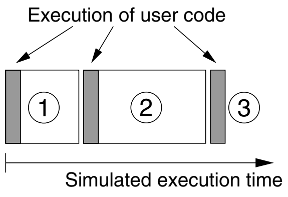


Fig. 4. The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

receiving node. The transmitted message is put in a buffer at the receiving computer node.

A message contains information about the sending and the receiving computer node, arbitrary user data (typically measurement signals or control signals), the length of the message, and optional real-time attributes such as a priority or a deadline.

The network block simulates medium access and packet transmission in a local area network. Six simple models of wired networks are currently supported: CSMA/CD (e.g., Ethernet), CSMA/AMP (e.g., CAN), Round Robin (e.g., Token Bus), FDMA, TDMA (e.g., TTP), and Switched Ethernet. TrueTime also supports two wireless protocols: IEEE 802.11 b/g (WLAN) and IEEE 802.15.4 (the MAC protocol used in Zigbee). The propagation delay is ignored, since it is typically very small in a local area network. Higher network layer protocols such as TCP can be implemented as user applications in the kernel blocks.

Configuring the network blocks involve specifying a number of general parameters, such as transmission rate, network model, and probability for packet loss. Protocol-specific parameters that need to be supplied include the time slot and cyclic schedule in the case of TDMA.

C. Execution model

The execution of tasks and interrupt handlers is defined by code functions. A code function is further divided into code segments according to the execution model in Fig. 4. The code can interact with other tasks and with the environment at the beginning of each code segment. This execution model makes it possible to model input-output latencies, blocking when accessing shared resources, etc. The number of segments can be chosen to simulate an arbitrary time granularity of the code execution. Technically it would, e.g., be possible to simulate very fine-grained details occurring at the machine instruction level, such as race conditions. However, that would require a large number of code segments.

The simulated execution time of each segment is returned by the code function, and can be modeled as constant, random, or data-dependent. The kernel keeps track of the current segment and calls the code functions with the proper argument during the simulation. Execution resumes in the next segment when the task has been running for the time associated with the previous segment. This means that preemption by higher-priority activities and interrupts may cause

```
function [exectime,data] = ctrl_code(segment,data)
switch segment,
  case 1,
    data.y = ttAnalogIn(1);
    data.u = calculate_output(data.x,data.y);
    exectime = 0.002;
  case 2,
    ttAnalogOut(1,data.u);
    data.x = update_state(data.x,data.y);
    exectime = 0.006;
  case 3,
    exectime = -1; % finished
end
```

Fig. 5. Example of a standard controller code function written in MATLAB code. The local memory of the control task is represented by the data structure `data`. This stores the input, the controller state, and the output between invocations of the code segments.

the actual delay between execution of segments to be longer than the execution time.

The listing in Fig. 5 shows an example of a code function corresponding to the time line in Fig. 4. The function implements a standard regulator in state-space form. In the first segment, the plant is sampled and the control signal is computed (`calculate_output`). In the second segment, the control signal is actuated and the internal state is updated (`update_state`). The third segment indicates the end of execution by returning a negative execution time.

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

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

III. Jitterbug

Jitterbug [4], [10], [5] is a MATLAB-based toolbox that calculates a quadratic performance criterion for a linear control system under various timing conditions. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. The main contribution of the toolbox, which is built on well-known theory (LQG theory and jump linear systems), is to make it easy to apply this type of stochastic analysis to a wide range of problems.

Jitterbug offers a collection of MATLAB routines that allow the user to build and analyze simple timing models of computer-controlled systems. A control system is built by connecting a number of continuous- and discrete-time systems. For each subsystem, optional noise and cost specifications may be given. In the simplest case, the discrete-time systems are assumed to be updated in order during the control period. For each discrete system, a random

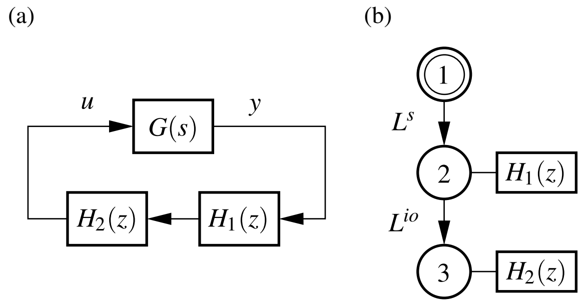


Fig. 6. A simple Jitterbug model of a computer-controlled system: (a) signal model and (b) timing model. The process is described by the continuous-time system $G(s)$ and the controller is described by the discrete-time systems $H_1(z)$ and $H_2(z)$, representing the sampler and the controller. The discrete systems are executed according to the periodic timing model.

delay, described by a discrete probability density function, is specified that must elapse before the next system is updated. The total cost of the system (summed over all subsystems) is computed algebraically if the timing model system is periodic or iteratively if the timing model is aperiodic.

To make the performance analysis feasible, Jitterbug can only handle a certain class of system. The control system is built from linear systems driven by Gaussian white noise, and the performance criterion to be evaluated is specified as a quadratic, stationary cost function. The timing delays in one period are assumed to be independent from the delays in the previous period. Also, the delay probability density functions are discretized using a time-grain that is common to the whole model.

Even though a quadratic cost function can hardly capture all aspects of a control loop, it can still be useful when one wants to quickly judge several possible controller implementations against each other. A higher value of the cost function typically indicates that the closed-loop system is less stable (i.e., more oscillatory), and an infinite cost means that the control loop is unstable. The cost function can easily be evaluated for a large set of design parameters and can be used as a basis for the control and real-time design.

A. *Example*

In Jitterbug, a control system is described by two parallel models: a signal model and a timing model. The signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period. An typical example is shown in Fig. 6, where a computer-controlled system is modeled by three blocks. The plant is described by the continuous-time system G, and the controller is described by the two discrete-time systems H_1 and H_2. The system H_1 could represent a periodic sampler, while H_2 represents the computation of the control signal and the actuator. The associated timing model says that, at beginning of each period, there is a random delay L^s before H_1 is executed. Then there is another random delay L^{io} before H_2 is executed. The delay could model computational delays, scheduling delays, or network transmission delays. Note that this model corresponds to the task timing diagram in Fig. 1. The latency probability distributions could be obtained by statistical scheduling analysis, by simulation of the scheduling algorithm, or from measurements.

As examples of extensions, the same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. It is also possible to make the update equations depend on the time since the first node became active. This can be used to model jitter-compensating controllers.

IV. Task Model Evaluation

A. *Control Subtask Models*

There are several ways to implement a periodic controller in a priority-based real-time kernel. The simplest approach is to use a single task to implement the controller, and to use, e.g., rate-monotonic or deadline-monotonic priority assignment to decide the priority of the task. This approach is denoted the *standard task model* (STM) in the sequel. However, in a multi-tasking environment a task can always be suspended by higher-priority tasks. Likewise, a task can be blocked by lower-priority tasks if the tasks share other resources. The result of this is temporal nondeterminism in the form of sampling jitter and input-output jitter.

One way to reduce the jitter is to use a subtask model, in which the controller is modelled as a sequence of subtasks and where different priorities are used in the different subtasks. Here, three such subtask models will be evaluated with respect to the control performance that they typically give rise to.

In the *Calculate Output-Update State* (CO_US) model [2] each control task is split into two sequential subtasks: Calculate Output and Update State. The sampling is performed at the beginning of the CO part and the actuation is performed at the end of the CO task. In the update state part the internal states of the controller are updated. Since it is the execution time of CO that decides the input-output latency it is assigned a shorter deadline than the US task. By using deadline-monotonic priority assignment the deadlines are reflected in the priorities.

In the *Initial, Mandatory and Final* (IMF) subtask model [6], [1] each control task is instead split into three subtasks: the initial task in which the sampling is performed, the mandatory task in which the control calculations, including the state update, are performed, and the final task where the actuation is performed. The model assigns high priority to the final task, medium priority to the initial task, and low priority to the mandatory task.

Finally, we propose a new model called *Initial, Calculate Output, Final, Update* (ICOFU) that combines the properties of the two above models. The main idea is to split the mandatory subtask of the IMF model into a Calculate Output subtask and an Update State subtask. The initial and final tasks remain the same as in the original IMF model. This

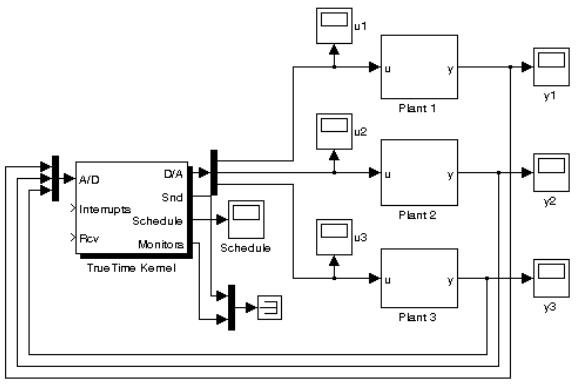


Fig. 7. TrueTime model to control multiple plants

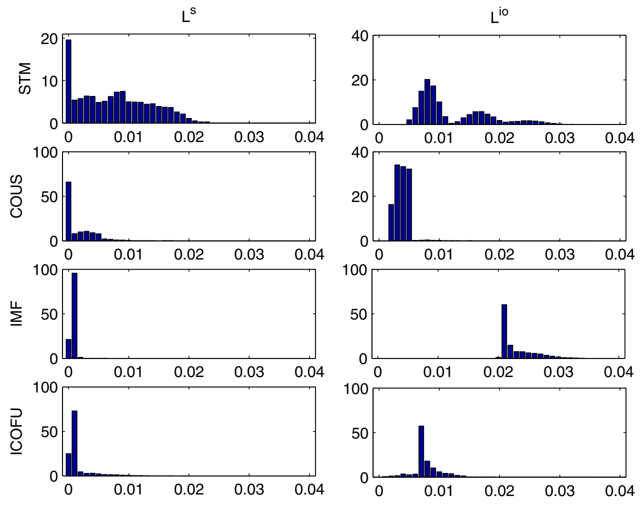


Fig. 8. L^s and L^{io} distributions for STM, CO_US, IMF and ICOFU models

gives additional flexibility since now it is possible for the update state subtask to complete after the final task, and even after the initial task of the next iteration. The schedulability of the ICOFU model is checked through offset-based, or asynchronous, schedulability analysis, see [11].

B. TrueTime and Jitterbug Models

The evaluation of the different subtask models is performed through a scenario in which a computer with limited computing resources is used to control three independent plants with different initial parameters. An LQG controller is designed for each plant. The controller is designed to compensate for the minimum possible delay for each subtask model. Fig. 7 shows the TrueTime block diagram for the system.

In order to evaluate the performance of the subtask models, a plant test batch is used. The batch is inspired by [7], where a test batch for process control and PID tuning was presented. Compared to that batch, plants with excessive dead-times have not been considered, and some more difficult-to-control resonant and unstable plants have been added. There batch contains 27 plants in total, including stable, marginally stable, and unstable plants.

The performance of the controller is measured using a quadratic cost function

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

where ρ is a weight, u is the control signal and y is the plant response. The performance index is valid under the assumptions that the reference value is zero, the closed-loop system is asymptotically stable, and the system is disturbed by zero-mean white noise.

The cost function could in theory be evaluated numerically using very long simulations with TrueTime. A better alternative is to use Jitterbug, where the cost function can be computed analytically. The necessary delay distributions are obtained from TrueTime using the possibility to log scheduling events to the Matlab workspace and thereby calculate the true distributions for the sampling latency and input-output latency given the task set. The computation times for the different subtasks in TrueTime were specified by uniform distributions between a minimum and maximum value.

C. Results

Fig. 8 reports the sampling latency and input-output latency distributions obtained by TrueTime simulations of the various task models. The STM model gives the highest sampling jitter and input-output jitter. In order to reduce this variability, IMF introduces a constant input-output latency. Even though sampling and actuation are done periodically (at 0.5 and 21.5 ms respectively) the fixed delay is very large (60% of period). The input-output latency is reduced to 7.5 ms when using the ICOFU model and the jitter is kept very low, achieving the best results. On the other hand, the CO_US model sends the control action as soon as possible (control action will never be sent later than 6ms) but with a higher input-output jitter.

Fig. 9 shows the values of the cost function obtained from Jitterbug when the overall system consists of three unstable plants (three inverted pendulums with different lengths). For long sampling periods, h, i.e. when the product $\omega_b h$ approaches 1, where ω_b is the bandwidth of the closed loop system, performance becomes degraded for all four task models (STM, CO_US, IMF, ICOFU). This is the expected behaviour, since, in general, slow sampling causes worse performance. The ICOFU model gives the best overall performance, for most sampling periods. When $\omega_b h \approx 0.4$ and $\omega_b h \approx 0.8$ for STM and IMF respectively, the cost goes to ∞. This corresponds to an unstable closed-loop system.

Fig.s 10 and 11 show the cost function for stable and marginally stable plants respectively, for the longest-period task. This task will suffer the most from interference from other tasks and therefore it has the most pessimistic distributions of the sampling latency and input-output latency. For stable plants no instability problems are obtained and the difference between the models is minor. For marginally

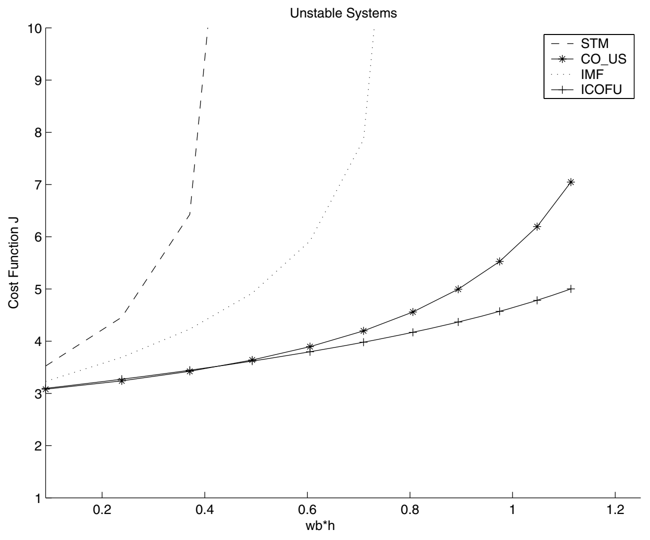


Fig. 9. Cost evaluation for the different task models on an open-loop unstable plant.

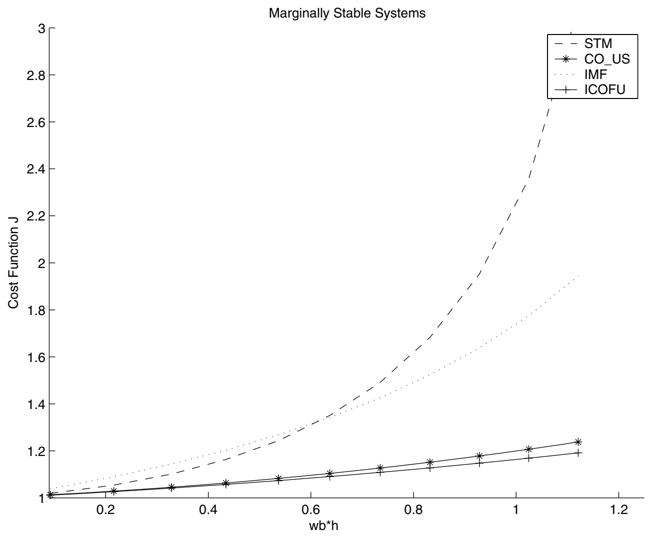


Fig. 11. Cost evaluation. Marginally Stable Systems

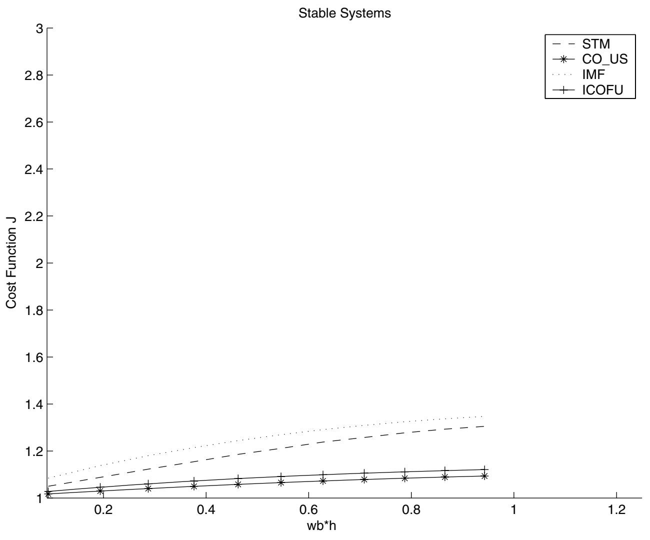


Fig. 10. Cost evaluation. Stable Systems

unstable plants ICOFU and CO_US give a nearly linear behaviour whereas the performance using the other two models deteriorate substantially as the sampling period is increased.

V. Conclusions

In this paper the two co-design tools TrueTime and Jitterbug have been presented. As an application example, we showed how the two tools can be used together to evaluate the performance of different controller implementations. First, the probability distributions of the sampling and input-output latencies are obtained using simulations in TrueTime. Then, the expected performance of the control loop as measured by a quadratic cost function is evaluated algebraically using Jitterbug. We also proposed a new task model, which was able to reduce the latency and jitter further. Repeating the cost computations for a large plant batch, some general conclusions regarding the various controller implementations could be drawn.

References

[1] P. Balbastre, I. Ripoll, and A. Crespo. Control task delay reduction under static and dynamic scheduling policies. In *7th International Conference on Real-Time Computer Systems and Applications RTCSA'00*, 2000.

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

[3] Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1–2):25–53, July 2002.

[4] Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and Karl-Erik Årzén. How does control timing affect performance? *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.

[5] Anton Cervin and Bo Lincoln. Jitterbug 1.1—Reference manual. Technical Report ISRN LUTFD2/TFRT--7604--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, January 2003.

[6] A. Crespo, I. Ripoll, and P. Albertos. Reducing delays in rt control: the control action interval. *IFAC World Congress Beijing*, 1999.

[7] T Hagglund and K. J. Astrom. Revisiting the Ziegler-Nichols step response method for PID control. *Journal of Process Control*, pages 635–650, 2004.

[8] Dan Henriksson and Anton Cervin. TrueTime 1.1—Reference manual. Technical Report ISRN LUTFD2/TFRT--7605--SE, Department of Automatic Control, Lund Institute of Technology, October 2003.

[9] Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. TrueTime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, July 2002.

[10] Bo Lincoln and Anton 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.

[11] I. Ripoll, A. Crespo, and A. Mok. Improvement in feasibility testing for real-time tasks. *Journal of Real-Time Systems*, 11:19–40, 1996.

[12] Martin Törngren, Dan Henriksson, Karl-Erik Årzén, Anton Cervin, and Zdenek Hanzalek. Tools supporting the co-design of control systems and their real-time implementation; current status and future directions. In *Proceedings of the 2006 IEEE Computer Aided Control Systems Design Symposium*, October 2006.