

Minimising Sampling Jitter Degradation in Real-Time Control Systems

Pau Marti and Josep M^a Fuertes
Automatic Control Department
Technical University of Catalonia (Spain)
{pmarti,pepf}@esaii.upc.es

Gerhard Fohler
Department of Computer Engineering
Mälardalen University (Sweden)
gfr@mdh.se

Abstract

Control applications mandate the application of real-time systems to meet stringent timing constraints. The commonly used priority-based preemptive scheduling methods introduce jitter variability on task instance executions. This implies that the separation between consecutive starting times of a given controller task is not constant while sampling periods were assumed to be constant for the control analysis. This variability on the sampling period of the computer implementation can degrade the controlled-system performance and even lead to instability in the system.

In this paper we present a method of minimising the control performance degradation due to sampling jitter of priority scheduled controller tasks. The main idea is to include actual controller task periods, i.e., the separation of its previous instances starting times, to calculate control parameters in order to obtain accurate control actions. Thus, our method compensates for variations of sampling jitter by appropriate variations of controller parameters. Analysis shows that even for standard rate monotonic scheduling, good control quality can be achieved.

1. Introduction

Usually, real-time computer-controlled systems are built in two separate stages: control design and its computer implementation. In this sense, Törngren [14] pointed out that there is a gap between control engineering methods and computer engineering methods when dealing with real-time computer-controlled systems.

On the one hand, it is known that priority-based preemptive scheduling algorithms introduce jitter variability on tasks instances executions. In real-time computer-controlled systems, one of the consequences is that the separation between consecutive task instance starting times, called sampling interval, is not constant. This variability in the sampling interval is called sampling jitter [6]. On the other hand, in the computer implementation of a control design, control theory

assumes a constant sampling period where control outputs are calculated according to this constant period.

This contradictory situation, sampling interval variability due to sampling jitter vs. constant sampling period assumed by control theory, can result in a degradation of the controlled system performance, and even lead to instability in the system. This situation is shown in Figure 1, where task τ_2 is a controller task designed with a sampling period 15 time units (t.u.) but due to sampling jitter, its actual sampling period (sampling interval) is 19 t.u.

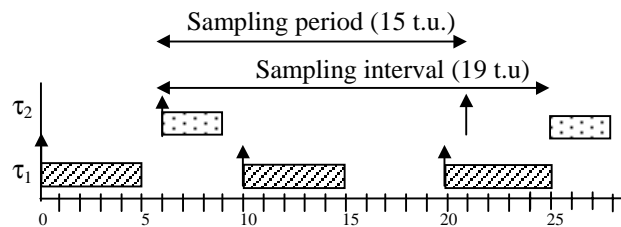


Figure 1. Sampling period vs. sampling interval

In this paper, we present a method of minimising control performance degradation that real-time computer-controlled systems can suffer due to sampling jitter when controller tasks are scheduled by priority-based preemptive scheduling algorithms. We do so by adjusting controller parameters according to actual instance separation, i.e., with jitter. Therefore, at each controller task instance execution and according to the separation of its consecutive starting times, we adjust the assumed controller task periods, which are parameters of the controller tasks used to calculate the control. In this way, more accurate control actions are performed, which implies a minimisation of the introduced sampling jitter degradation.

A similar timing compensation for the variations from sample to sample is suggested in [7] and [1] where controller parameters are recalculated at run-time assuming that time measurements are available. What we suggest is to use off-line knowledge of the exact tasks instances starting times in order to allow the recalculation of all controller parameters before runtime. Once all the new calculated parameters for each known sampling

interval are tabled, during run-time, controller tasks will only have to access the table instead of performing on-line re-computations. By doing that, controller tasks computation times will be significantly reduced, allowing a better control performance [6].

We achieve off-line knowledge of the exact tasks instances starting times that will appear during run time by using Dobrin et al. results [12]. In this approach, off-line scheduled tasks are transformed into fixed priority tasks which, if scheduled by Fixed Priority Scheduling, will reconstruct at run-time the original off-line schedule as well as will preserve the off-line scheduled starting times.

We have developed a set of procedures to enable simple calculation of sampling intervals for runtime period adjustment. We have focused on standard rate monotonic [9], RM – deadlines equal to periods - to show the effectiveness of our approach. Although RM is not well suited for control applications, our method compensates the jitter problems well. Applying our methods to the Earliest Deadline First or other more appropriate scheduling algorithms will further the improvement.

The jitter problem has been a focus of research. Eker et al. [6] show that, by scheduling two parts of a control algorithm as separate tasks, the input/output computational delay can be reduced significantly and, as a consequence, the system performance is improved. In [3], Baruah proposed a formal quantitative model for output. However, in the above works, the common feature is that they enhance the control performance or they minimise the jitter by modifying the scheduling algorithms. What we propose is to accept the jitter that the scheduling algorithm introduces and include it in the controller at runtime in order to minimise the jitter degradation.

Seto has also treated optimization of control systems performance in [13], where tasks frequencies are optimised in order to make all tasks schedulable and to enhance control system performance. However, the presented analysis is done before runtime while what we propose is done at runtime. Caccamo [5] and Buttazzo [4] have presented runtime control performance optimization approaches. In [5], the main idea is that tasks computation times are allowed to range from average to worst case computation times and periods are adjusted at runtime to optimise control performance and enhance schedulability by using server approaches. In [4], an elastic task model is presented, in which task periods are treated as springs, with given elastic coefficients. Using this new task model, periodic tasks can intentionally change their execution rate at runtime to provide different quality of service, and the other tasks can automatically adapt their periods to keep the system underloaded. In both approaches, the major goal has been to adapt at runtime properties of the resulting schedule by modifying the scheduling algorithm in order to improve schedulability and enhance control

performance responsiveness. However, no jitter degradation due to sampling jitter is taken into account, even though it can appear in the proposed scheduling algorithms and degrade the system performance.

Related to scheduling and control systems, Marti [10] outlines requirements of a real-time scheduling algorithm aimed to be applied in real-time computer-controlled systems.

This paper is structured as follows: in section 2, we introduce the problem description in terms of jitter variability. Section 3 gives an overview on different control theory design approaches. In section 4, we present the controller runtime period readjustment along with the set of procedures used to calculate the task instance starting times. Section 5 presents the obtained results through an example and finally, in section 6, conclusions and future work are outlined.

2. Problem description

Real-time systems usually assume task periods as fixed timing constraints. A real time task τ_i can be characterised by a fixed period T_i , a deadline D_i and a worst-case computation time C_i . Since the early work on real time scheduling, real time tasks are mainly scheduled using priority-based preemptive algorithms. One common feature of almost all these scheduling algorithms is that they introduce jitter variability on the tasks instances executions. The introduced jitter can be divided into two major sources of variability that lead to three different jitter types.

The first source of variability occurs between consecutive starting times of task instances (sampling interval), variability, called sampling jitter. It appears while higher priority tasks are executing when the target task is released (interference). This sampling jitter implies that the separation between consecutive starting times of task instances is not constant at runtime.

The second source of variability affects the task computation time. It occurs when higher priority tasks are released while the target task is executing, or when the target task tries to access a shared resource that it is being used by another task (blocking time). This variability, called computational jitter implies that the execution time of tasks instances is not constant at runtime. As a consequence, the separation between consecutive completion times of tasks instances (called actuation interval) is not constant either. We call this later variability actuation jitter. In [3] it is referred to as an output jitter.

As an example, let's assume a control system that includes two controller tasks, each performing the sampling at the beginning of its execution time and performing the output at the end of its execution time.

Table 1. Task set parameters

	T_i	C_i	Description
τ_2	12	3	Controller task2
τ_1	7	4	Controller task1

Task τ_2 has been designed with a sampling period of 12 time units (t.u.) and a computation time of 3 t.u. The other task has been designed with a sampling of 7 t.u. and a computation time of 4 t.u. Under RM, task τ_1 will have a higher priority than τ_2 . In this case, task τ_2 will suffer all types of jitter caused by task τ_1 . The set of tasks passes the RM schedulability test:

$$U = \sum_{i=1}^2 \frac{C_i}{T_i} = \frac{4}{12} + \frac{3}{7} \approx 0.77 < 2(2^{1/2} - 1) \approx 0.83$$

In the resulting schedule partially shown in Figure 2, we can see that some of the task τ_2 instances suffer sampling interval (SI) variability due to sampling jitter (compare SI_1 with 9 t.u. vs. SI_2 with 12 t.u.). They also suffer computation time (CT) variability due to computational jitter (compare CT_1 with 4 t.u. vs. CT_2 with 7 t.u.). Therefore the actuation interval (AI), due to actuation jitter (compare AI_1 with 12 t.u. vs. AI_2 with 9 t.u.) will not be constant either.

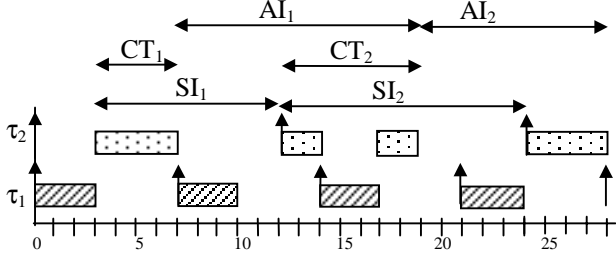


Figure 2. Partial schedule with jitter presence

In real-time computer-based controlled-systems, all three types of jitter variability can degrade the system responsiveness.

In this paper, our main interest is to minimise the degradation that the system can suffer due to sampling jitter. Therefore, in order to be able to focus on sampling jitter, we eliminate computational jitter and consequently actuation jitter by setting the computation time of the controller tasks to zero. As a result, sampling, control computation, and actuation are considered to be instantaneous. For practical purposes, the execution time of the controller tasks will be greater than zero, introducing computational and actuation jitter as well, while our method still compensates for the sampling jitter.

3. Computer-control: discrete vs continuous approach

Control theory [2] assumes a highly deterministic timing of an implementation. At the design stage, controllers are designed according to system dynamics, specified requirements and control methodologies. After the design stage, when the controller is computer implemented, constant sampling periods are assumed. That means that at each controller task execution, the control output signal is calculated according to the chosen sampling period, that is considered a constant value throughout the execution time.

Computer-based control systems can be designed in two different ways: discrete-time design or discretization of a continuous-time design.

Discrete-time control theory considers the system through the values of the system inputs and outputs at the sampling instances. To do this, a sampled version of the continuous system model, that will be sampling period dependent, is derived. By doing this, well-known discrete-time system descriptions are obtained. Using these descriptions, a wide range of discrete-time controller design methods can be applied in order to obtain the discrete controller, which in the end, is an algorithm that will be executed at every sampling period. It has to be pointed out that the resulting controller is characterised by several design parameters that are highly dependent of the sampling period used earlier, at the beginning of the design stage.

Discretization of continuous-time design is to design the controller in the continuous time domain, and then, approximate this design by an implementation (computer-based controller) through fast sampling. Knowing that the faster sampling, the better discretisation, some criteria exist in order to determine which has to be the appropriate sampling period. These criteria are the so called rules-of-thumbs, that can be summarised saying that the appropriate sampling period has to be within a certain range, obtained according to a given system parameter or mathematical expression of it. For example, it is said [2] that reasonable sampling rates are 10 to 30 times the bandwidth, or 4 to 10 per rise time. Using these criteria, smaller sampling intervals than for the discrete-time design are obtained. Therefore, in the discretisation stage, the sampling period can be chosen within a given range.

To exemplify the runtime period readjustment, we have chosen the discretisation of a continuous-time design approach for the sake of simplicity. However, the work could be extended to discrete-time design by taking into account that at each controller instance execution, period readjustment will imply also to redesign all the controller parameters.

We have also chosen the discretisation of continuous-time design approach because it has as advantage that, in general, the resulting controller is less sensitive to variations in the sampling interval due to the controller design method used. That means that a discrete continuous-time designed controller, once designed, can be executed at different frequencies, within the acceptable range and without degrading the system performance. However, a discrete-time designed controller will degrade the system performance and even lead the system to instability if it is executed at a different frequency than the one chosen at the design stage. Therefore, in discrete-time design, if the execution frequency of the controller is different from the design frequency, the controller has to be redesigned using the new execution frequency before executing it.

Besides, between all continuous-time controller design methods that can be used to design continuous-time controllers, we will focus on PID (Proportional, Integral and Derivative) controllers, which are one of the most popular controllers and widely used in many application areas. PID controllers are also chosen because once the discretisation of the continuous-time design is done, it can be easily seen that the resulting controller, which is a small algorithm, is basically a function of the sampling period, facilitating then the development of our method.

4. Controller runtime period readjustment

In this section we introduce the controller runtime period adjustment used to minimise the degradation that real-time computer-controlled systems can suffer due to sampling jitter. To perform the period readjustment we need to measure the sampling interval which we defined as a separation between consecutive controller task instance starting times. To facilitate this calculation that will be performed off-line, we have implemented a set of procedures to calculate the starting time of the k-instance of the i-task, given a set of schedulable tasks under RM algorithm.

The computation performed using these procedures is based on tasks periods (deadlines equal to periods) and worst-case computation times. These assumptions could also be relaxed. Recall that for simulation purposes, controller tasks will have a computation time set to zero.

The basic idea is, in RM, given the theoretical start time in isolation (TST) of the k-instance of the i-task, $k \cdot T_i$ do:

- Calculate the initial time instant (ITI) from which the theoretical starting time of the task instance can suffer delay.
- From the initial time instant, calculate the delay period (DP), which is the interference period (IP)

plus the idle time (IDT) that are actually delaying the real starting time of the task instance.

- The interference period is the execution time of all higher priority tasks than i-task within a given period if and only if, their execution can delay the starting time of the task instance.
- The idle time is the remaining time that is not used for execution of tasks but it can also delay the starting time of the tasks instance
- The real start time (RST) will be the initial time instant plus the delay period

An example is given using the set of tasks of Table 2.

Table 2. Example task set parameters

	T_i (t.u.)	C_i (t.u.)
τ_1	5	2
τ_2	14	4
τ_3	18	2

We will show that, due to all sources of jitter, the real starting time of some instance of some task is not always the theoretical starting time in isolation of the instance.

Figure 3 shows a partial schedule for the given set of schedulable tasks under RM scheduling algorithm. For example the real starting time of the second instance of the task τ_3 is not 18 t.u. ($1 \cdot 18$), but 22 t.u.:

$$RST = ITI + DP = ITI + (IP + IDT) = 10 + (10 + 2) = 22 \text{ t.u.}$$

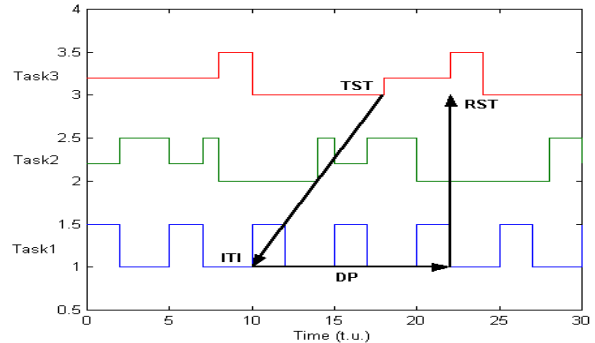


Figure 3. Starting time timing analysis

In order to perform the above computations, a specification of the task set in terms of tasks periods and worst case computations times has to be given as well as k-instance of the target i-task that we wish to calculate. The tick-size, understood as a base time unit from which all the computations are performed, has to be provided as well. The set of procedures are executed by the following command: $rst = RST(k, i, Tasks, ticksize)$.

These procedures are used off-line because using [12] we can assure that the on-line task instances starting times

will be the same of the off-line scheduled task instances starting times. Therefore, controller parameters for all possible sampling intervals can be recalculated and tuned off-line, minimising thus the controller computation time. The obtained values over the hyper-period are tabled and fast on-line access can be achieved. Detailed information can be found in [11].

In the following we introduce the runtime period readjustment, which uses the presented procedures, in the case of a continuous-time designed PID controller as we justify in section 4.

A continuous-time designed PID controller is designed according to certain specifications in order to improve both the transient response and the steady-state response for a given input. Figure 4 shows the block diagram of the closed-loop system under study.

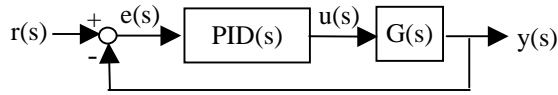


Figure 4. Block diagram of a closed-loop system

The physical system to be controlled is specified by the transfer function of the physical plant, $G(s)$. The output of the plant, $y(s)$ is compared to the reference signal, $r(s)$, which is the desired response, obtaining an error signal, $e(s)$. The PID controller, specified by its transfer function, $PID(s)$, then processes the error signal. The controller then generates the control output signal $u(s)$, to drive the error signal towards to zero, that is, to approach the actual response $y(s)$ to the desired response $r(s)$.

A PID controller can be generally described by equation (1), where $u(t)$ is the controller output signal, $e(t)$ is the input error signal (difference between real response $y(s)$ and desired response $r(s)$) to be processed, and K_p , K_i and K_d are constant gains to be determined in the design stage.

$$u(t) = K_p e(t) + K_i \int_0^t e(t) dt + K_d \frac{de(t)}{dt} \quad (1)$$

Its transfer function is described by the equation (2)

$$PID(s) = K_p + \frac{K_i}{s} + K_d s \quad (2)$$

The PID gains are determined according to the plant dynamics, following the specified requirements and using one of the continuous-time PID design methods.

Once the PID controller has been designed in the continuous-time domain, the discretization of the controller is done using the rule-of-thumb that indicates that a reasonable sampling rate is 10 to 30 times the bandwidth of the system and choosing one of the discrete approximations of a continuous-time PID controller. The chosen PID discretization approximation can be described by the following set of equations (3):

$$\begin{aligned} p(t) &= K_p e(t) \\ i(t) &= i(t-T_s) + \frac{K_i T_s}{2} (e(t) + e(t-T_s)) \\ d(t) &= \frac{K_d}{T_s} (e(t) - e(t-T_s)) \\ u(t) &= p(t) + i(t) + d(t) \end{aligned} \quad (3)$$

Recall that constant gains K_p , K_i and K_d have been calculated previously at the continuous-time design stage and the sampling period T_s has been chosen in the discretization stage. At this point, the PID controller task can be implemented and the algorithm that has to be executed at each k task instance, called PID algorithm, is shown if Figure 5.

In the PID algorithm execution, at each PID controller task instance, T_s is constant because the separation between task instances is supposed to be constant. However, if this PID controller task is scheduled using priority-based preemptive scheduling algorithms, sampling jitter can appear. The consequence of this jitter will imply that, although the sampling interval may not be constant, the output u_k of the algorithm will be calculated according to the constant T_s , because the sampling period is a constant value fixed in the discretization stage. This contradiction will lead to a degradation of the system response because the control action is calculated taking into account that the period between instances will be constant when it really may not be constant.

Algorithm PID

```

{
  Read_input (yk);
  Read_input (rk);
  ek = rk - yk;
  pk = Kp ek;
  ik = ik-1 +  $\frac{K_i T_s}{2} (e_k + e_{k-1})$ ;
  dk =  $\frac{K_d}{T_s} (e_k - e_{k-1})$ ;
  uk = pk + ik + dk;
  write_output (uk);
}

```

Figure 5. PID algorithm

However, this problem can be solved by readjusting the period T_s in the controller algorithm by a new period T_A calculated according to the separation between the actual instance of the PID controller task and the previous one. In this way, the computation of the control action will take into account the exact separation between instances,

which can be T_s but also can be different and suffer a small variation due to sampling jitter. As a result, the degradation of the system response can be eliminated.

The new PID algorithm, called ReadjustedPID algorithm, which readjust the period T_s to T_A using the set of developed procedures and will minimise the sampling jitter degradation, is shown in Figure 6.

```

Algorithm ReadjustedPID
{
  Read_input ( $y_k$ );
  Read_input ( $r_k$ );
   $e_k = r_k - y_k$ ;
   $rst_{k-1} = RST(k-1, i, Tasks, ticksize)$ ;
   $rst_k = RST(k, i, Tasks, ticksize)$ ;
   $T_A = rst_k - rst_{k-1}$ ;
   $p_k = K_p e_k$ ;
   $i_k = i_{k-1} + \frac{K_i T_A}{2} (e_k + e_{k-1})$ ;
   $d_k = \frac{K_d}{T_A} (e_k - e_{k-1})$ ;
   $u_k = p_k + i_k + d_k$ ;
  Write_output ( $u_k$ );
}

```

Figure 6. ReadjustedPID algorithm

It has to be pointed out that this period readjustment to the new calculated one is correctly done if the variations between T_A for each PID controller task instance don't exceed the range of sampling periods specified by the rule-of-thumb used in the discretisation stage. Besides, the system response for the longest T_A that will appear at runtime due to sampling jitter, which can be known offline, has to be previously evaluated.

5. Results

We present the results of the method through an example. We consider a servo problem, which is one of the most common control problems, where the major issue is the command signal following. All the simulations have been done using the simulator presented in [8].

Consider a DC servo described by the following continuous-time transfer function:

$$G(s) = \frac{1000}{s(0.5 \cdot s + 1)}$$

Following the specified requirements, as to have a percentage overshoot less that 15%, the PID parameters

are tuned at the following values $K_p = 1.8$, $K_i = 0.1$ and $K_d = 0.09$. The quadratic pulse (reference signal) response of the DC servo with the PID controller in the continuous-time domain, that fulfils the specified requirements, can be seen in Figure 7.

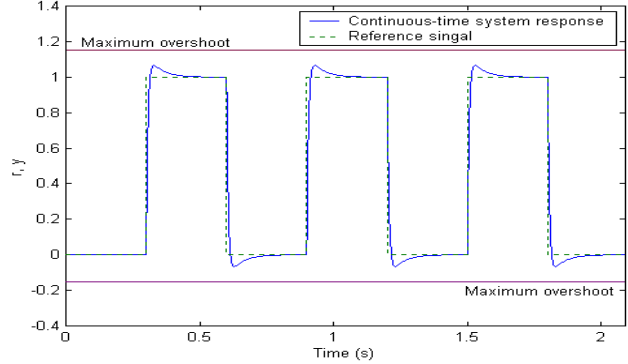


Figure 7. Continuous-time system response

Once the PID controller has been designed in the continuous-time domain, following the stated rise time rule-of-thumb, the discretisation of the controller is done. The chosen sampling period (T_s) is 2ms.

The quadratic pulse response of the DC servo with the PID controller approximated for the previous discrete-time equations (3) with sampling period 2ms and implemented using the PID algorithm in a PID controller task executing on a CPU can be seen in Figure 8.

Comparing Figure 7 and Figure 8 it can be seen that the quadratic pulse response of the system in both cases is almost the same, fulfilling the specified requirements, which means that the discretization process has been successfully done.

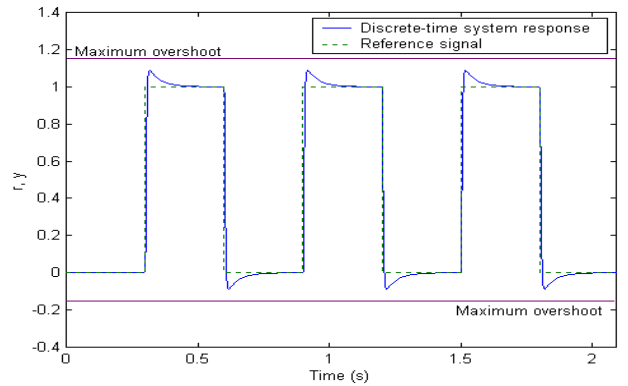


Figure 8. Discretization system response

In this case, the PID control task is executing alone on one CPU. Its execution is at each 2ms, which means that each task instance executes 2ms after the previous one, with a constant separation between consecutive instances.

Now we are going to see what the effects on the control performance are when the PID controller task has to share the CPU with other tasks. For the sake of simplicity, we

schedule the control task by RM (see Table 3) along with a higher priority task.

Table 3. Task set

	C_i (ms)	T_i (ms)
PID controller task	0	2.0
Dummy task	0.5	1.3

The given set of tasks passes the schedulability test:

$$U = \sum_{i=1}^2 \frac{C_i}{T_i} = \frac{0}{0.0020} + \frac{0.0005}{0.0013} \approx 0.38 < 2(2^{1/2} - 1) \approx 0.83$$

The resulting partial schedule where the PID controller task suffers sampling jitter is shown in Figure 9.

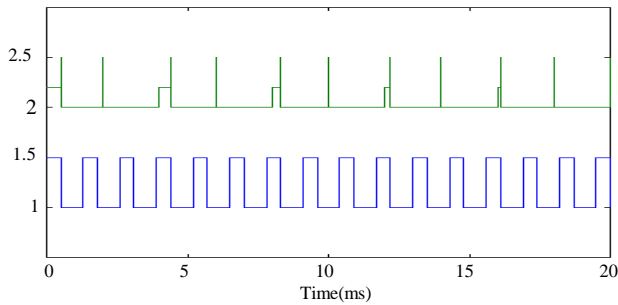


Figure 9. Partial schedule with jitter presence

PID controller task suffers sampling jitter at many instances (in the figure 9 sampling jitter corresponds to interference periods, with a value equal to 2.2. Value 2 means that the task is not executing and value 2.5 means that the task is executing). For example, the first instance in the partial schedule starts at time $t+0$. Therefore, the second instance should start at $t+2.0$, but due to sampling jitter, it starts at $t+0.25$, 0.5ms later that it was suppose to start. In this case, instead of having the PID task executing at every 2.0ms, its activation sequence follow and activation pattern where the time variability between sampling intervals is within ± 0.5 ms.

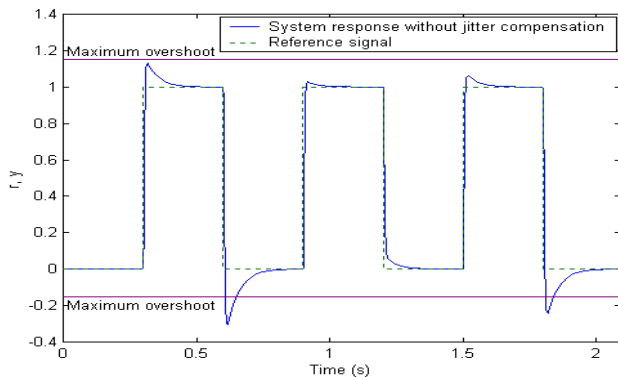


Figure 10. System response with sampling jitter degradation

Figure 10 shows the effect of this sampling variability on the system performance. The system response, which was to track the quadratic pulse input, suffers important degradation, which drives the system response out of the specified requirements. This degradation is due to the fact that control actions are calculated using the PID algorithm, where the parameter T_s is constant, while the sampling interval is not constant.

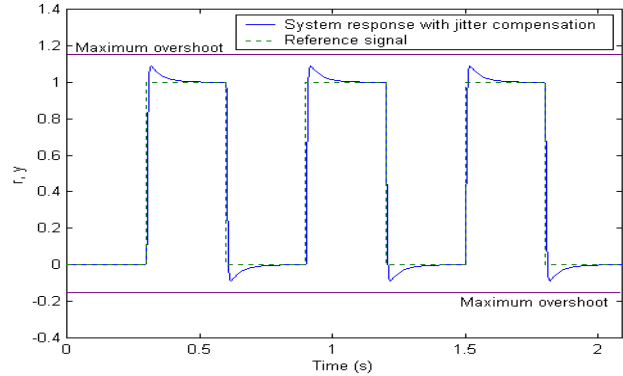


Figure 11. System response with jitter compensation

However, Figure 11 shows the quadratic pulse response of the DC servo with the PID controller approximated for the previous discrete-time equations (3) with sampling period 2ms and implemented using the ReadjustedPID algorithm scheduled along with the dummy task.

It is clear that in the system response performance the previous degradation due to sampling jitter is eliminated. This elimination is due to the fact that in this case, control actions are calculated using the ReadjustedPID algorithm, where the T_A parameter is adjusted at runtime to the separation between consecutive PID tasks instances.

Figure 12 shows partially the response of the system without jitter compensation and with jitter compensation in order to stress how big the sampling jitter degradation can be and how good the compensation achieved by our method is.

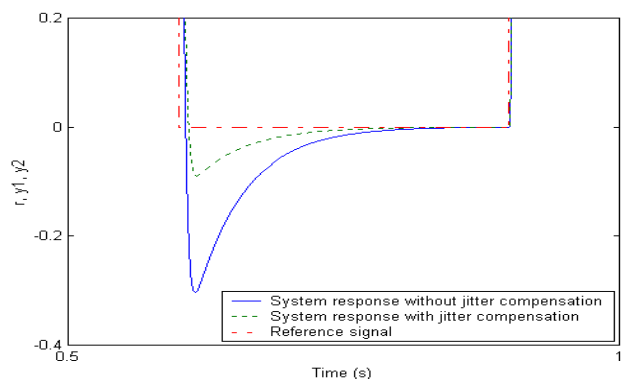


Figure 12. Jitter degradation vs. jitter compensation

Finally, Figure 13 shows the same partially simulation as the above, where the response of the system with jitter compensation is compared to the system response of Figure 5, where the PID controller task was scheduled alone. We can see that the jitter compensation response is as good as the one obtained by a PID controller tasks when it doesn't suffer any sampling jitter.

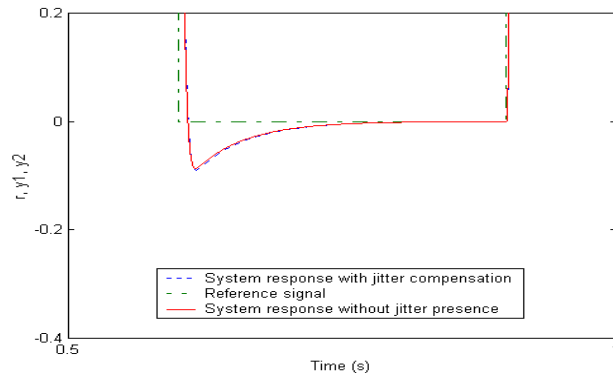


Figure 13. Jitter compensation vs. no jitter presence

6. Conclusions

In this paper, we presented a method of minimising control performance degradation that real-time computer-controlled systems can suffer due to sampling jitter when controller tasks are scheduled by priority-based scheduling algorithms.

Taking advantage of control systems properties, we have identified the possibility of readjusting parameters of a controller implementation by taking into account actual separation, i.e., including jitter. In this way, we obtain more accurate control actions. As a consequence, the introduced jitter and the implied degradation can be eliminated.

We have focused on standard rate monotonic scheduling, which is not well suited for control applications. Despite this, our method compensates the sampling jitter well and should improve further if applied to other scheduling algorithms.

Simulation results show the effectiveness of our approach.

References

[1] K.-E. Årzen, A. Cervin, J. Eker and L. Sha. "An Introduction to Control and Scheduling Co-Design", *39th IEEE Conference on Decision and Control, Sydney, Australia*, December 12-15, 2000

[2] K.J. Åström and B. Wittenmark. *Computer-Controlled Systems. Third edition*. Prentice Hall. 1997

[3] S. Baruah, G. Buttazzo, S. Gorinsky and G. Lipari, "Scheduling Periodic Tasks Systems to Minimize Output Jitter" *Proc. International Conference on Real-Time Computing Systems and Applications*, IEEE Computer Society Press. pp 62-69, Hong Kong, December 1999

[4] G. Buttazzo, G. Lipari and L. Abeni, "Elastic Task Model for Adaptive Rate Control" *IEEE Real-Time Systems Symposium, Madrid, Spain*, December 1998

[5] M. Caccamo, G. Buttazzo and L. Sha, "Elastic Feedback Control", *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, pp. 121-128, June 2000

[6] A. Cervin "Improved Scheduling of Control Tasks" in *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, England, June 1999

[7] A. Cervin *Towards the Integration of Control and Real-Time Scheduling Design*. Licentiate Thesis. ISRN LUTFD2/TFRT—3226—SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, May 2000

[8] J.Eker and A. Cervin "A Matlab Toolbox for Real-Time and Control Systems Co-Design", in *Proc. of the 6th Int. Conference on Real-Time Computing Systems and Applications*, Hong Kong, China, December 1999

[9] C. Liu and J. Layland "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *J.ACM*, 20, 46-61. 1973

[10] P. Marti, R. Villa, J.M. Fuertes and G. Fohler, "Real Time Methods Requirements in Distributed Control Systems". *Proc. 25th IFAC Workshop on Real-Time Programming*, Palma, Spain, pp 101-108, 2000

[11] P. Marti, "Rate Monotonic Timing Analysis" Research report ESAII-RT-00-07. Automatic Control Department, Technical University of Catalonia. July 2000

[12] R. Dobrin and G. Fohler, "Attribute Assignment for the Integration of Off-line and Fixed Priority Scheduling". In *WIP Proc. of the 21st IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, November 2000

[13] D. Seto, J.P. Lehoczky, L. Sha and D.G. Shin, "On task Schedulability in Real-Time Control Systems". *RT Systems Symposium, 17th IEEE*. p 13-21, 1996

[14] M. Törngren, "Fundamentals of Implementing Real-time Control Applications in Distributed Computer Systems." *J. of Real-Time Systems*, 14, 219-260, Kluwer Academic Publishers. 1998