Adaptive Scheduling for Real-Time Control

Sanjoy Baruah
Washington University
United States
baruah@wustl.edu

Mehdi Hosseinzadeh
Washington State University
USA
mehdi.hosseinzadeh@wsu.edu

Ilya Kolmanovsky
The University of Michigan
USA
ilya@umich.edu

Bruno Sinopoli
Washington University
USA
bsinopoli@wustl.edu

Abstract

Controllers can be designed to adapt to dynamic changes in the computational capacity that is available for their execution by adjusting their control computations. The concurrent development of such controllers, and the algorithms for run-time scheduling of these controllers, is investigated. It is shown how a mitigative controller, that can compensate for small errors that are made in computing the control signal during one iteration of the control loop by taking corrective action during the subsequent iteration, can be scheduled by a server-based real-time scheduling algorithm to provide both efficient resource-usage and acceptable control performance. This illustrates that concurrent and reciprocal consideration of mutual adaptivity can yield more resource-efficient implementations as well as better controller performance, than would be possible if scheduling and control were each considered separately.

Keywords

Server-based scheduling; preemptive uniprocessors; scheduling-adaptive mitigative control; recurrent real-time task systems.

ACM Reference Format:
Sanjoy Baruah, Mehdi Hosseinzadeh, Ilya Kolmanovsky, and Bruno Sinopoli. 2024. Adaptive Scheduling for Real-Time Control. In *The 32nd International Conference on Real-Time Networks and Systems (RTNS 2024), November 06–08, 2024, Porto, Portugal*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3696355.3696357

1 Introduction

As safety-critical cyber-physical systems are increasingly implemented using resource-constrained embedded platforms, there is widespread recognition of the need to co-design control and scheduling for such systems: control strategies should be designed in a manner that facilitates resource-efficient implementation, and resource-allocation strategies (and corresponding schedulability analyses) should be devised that enable effective control. Although efforts at such co-design have been made over the years by the real-time scheduling theory community (e.g., [3–5, 8, 10, 13, 14, 17, 19, 29, 31]), these have primarily tended to emphasize the scheduling aspects while making some significant simplifying assumptions regarding control aspects. In a similar vein, papers in the control literature that address scheduling and schedulability tend to make many simplifying assumptions regarding implementation issues — see, e.g., [2, 11, 12, 20, 24, 25, 30, 34]. However some recent papers have appeared in the real-time scheduling literature (see, e.g., [26–28, 32]) that, while primarily focused upon issues of scheduling and schedulability analysis, are more sophisticated in regards to their assumptions about the capabilities, needs, and characteristics of control algorithms. This manuscript represents a further contribution to this recent trend: the authors comprise a team of control researchers who have had little prior exposure to real-time scheduling research collaborating with scheduling-theory researchers that have limited knowledge of control, to jointly obtain a comprehensive understanding of both the control and the scheduling aspects of this co-design problem. Our collaborative efforts thus far have primarily focused on *mutual adaptivity*: how control strategies can be devised that can adapt to the time-varying availability of computing capacity, and how to design scheduling strategies that are best able to exploit such adaptivity.

Contributions and Organization. In Section 2 we (i) provide a basic introduction to some essential concepts in control theory in a manner intended to be comprehensible to scheduling-theory researchers with no prior exposure to control theory; (ii) discuss some forms of control adaptivity with a particular focus on *mitigative* control strategies; and (iii) briefly survey[1] some closely-related prior work on integrating control and schedulability considerations. In Section 3 we apply principles from real-time scheduling theory to develop a server-based framework for the resource-efficient implementation of a given collection of mitigative controllers upon a shared preemptive processor with limited computing capacity (of the kind that may, e.g., be found on an embedded resource-constrained CPS). In Section 4 we discuss some of the design choices that arise in developing a mitigative controller, the resolution of which have considerable impact upon schedulability, and suggest an heuristic approach to making these choices in a manner that balances considerations of both schedulability and control performance. We evaluate, via simulation experiments, our proposed framework in Section 5; we conclude in Section 6 by placing this work within a larger context of the co-design of controllers, and the

[1]This survey is integrated into the text in this section rather than being collected into a distinct sub-section, with particular related works being cited at the point where they are most relevant to the discussion.

RTNS 2024, November 06–08, 2024, Porto, Portugal

ACM ISBN 979-8-4007-1724-6/24/11
https://doi.org/10.1145/3696355.3696357

scheduling framework needed to ensure effective execution of these controllers upon resource-constrained implementation platforms.

2 Background & Context

The central theme that we explore in this paper is one of mutual adaptivity: designing controllers to be able to adapt to implementation constraints (in particular, limited computing capabilities), and developing scheduling strategies for implementing multiple such controllers upon a shared platform that both exploits such adaptivity, and does dynamic resource allocation that adapts as computational requirements change. We start out in Section 2.1 describing some basics of control theory, framed in terms that should be comprehensible to real-time systems researchers with no prior exposure to control theory. In Section 2.2 we discuss some simple ways in which control schemes can be designed to be adaptive in order to facilitate efficient implementation, and the corresponding scheduling problems that arise. In Section 2.3 we focus upon a particular form of such adaptivity based upon *mitigation.*

2.1 Some Basics of Control Theory

A *controller* controls a *plant* by repeatedly (i) sensing the output of the plant; (ii) computing an appropriate control signal; and (iii) applying this control signal to the plant. The following time-varying *signals* are relevant to a basic discussion on implementing controllers:

- $y(t)$: plant output as observed by sensors
- $r(t)$: the expected ("reference") value of $y(t)$
- $e(t) \stackrel{\text{def}}{=} (y(t) - r(t))$ (the "*error*")
- $z(t)$: controller state (its internal variables)
- $u(t)$: the control signal that the controller applies to the plant

(Note that $r(t), y(t), e(t), z(t)$ and $u(t)$ may each be a vector comprising multiple individual signals.) For historical reasons, many controllers are designed by control engineers assuming they operate in the continuous-time domain. E.g., Linear Time-Invariant (LTI) systems may be described in the continuous time domain by the following equations which characterize both the control signal that is applied to the plant, and the manner in which the plant's internal state changes:

$$\begin{aligned} \dot{z}(t) &= M_1 z(t) + M_2 e(t) \\ u(t) &= M_3 z(t) + M_4 e(t) \end{aligned} \tag{1}$$

Here, $\dot{z}(t)$ denotes the time-derivative –the rate of change– of the state variables $z(t)$; M_1, M_2, M_3 and M_4 are (constant) matrices of the appropriate dimensions.[2]

Although designed in the continuous-time domain, computer implementation of such controllers usually happens in the discrete-time domain: the controller task is invoked ("releases jobs") at discrete time-instants $a_1, a_2, \ldots, a_k, \ldots$, ($k \in \mathbb{N}$). Under the *Logical Execution Time* (LET) paradigm [21] that is widely adopted in CPS,'s the control signal that is computed based on the sensing that happens at time-instant a_k is communicated to the plant at

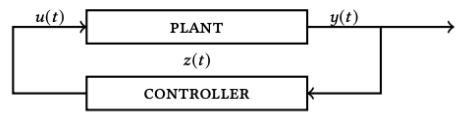
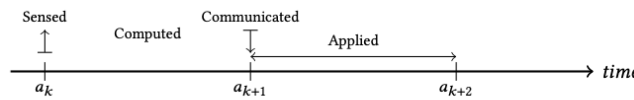


Figure 1: A generic control loop

time-instant a_{k+1} and forms the basis of the control signal that is applied to the plant over the duration $[a_{k+1}, a_{k+2}]$:[3]



In other words, it is the responsibility of the job released at time-instant a_k to compute $u(a_{k+1})$, the control signal that is to be applied to the plant at time-instant a_{k+1}, and to update the controller state to $z(a_{k+1})$. This is represented using the following notation:

$$\begin{aligned} z[k+1] &= Az[k] + Be[k] \\ u[k+1] &= Cz[k] + De[k] \end{aligned} \tag{2}$$

Note that although the system (plant + controller) state evolves continuously over time, the control signal $u[k]$ will be applied, and controller state $z[k]$ updated, only at discrete time instants. As a consequence, the *values* of the individual entries in the matrices $A, B, C,$ and D above depend upon the duration of the interval $[a_k, a_{k+1}]$; informally speaking, based upon having sensed the plant state at time-instant a_k the appropriate control signal to apply at time-instant a_{k+1} and the controller state at that instant, are obviously different for different values of a_{k+1}. We will henceforth use the notation $A(h_k), B(h_k), C(h_k),$ and $D(h_k)$ to indicate this dependence of the matrices on the duration $h_k \stackrel{\text{def}}{=} (a_{k+1} - a_k)$ between the release of the current and next jobs.

It is a common practice to associate a *period* parameter T with a controller, which specifies the duration between every successive pair of invocations (i.e., $h_k \equiv T$ for all $k \in \mathbb{N}$). When a controller is designed in the continuous-time domain to optimize for a certain control performance index and then discretized as in Expression 2, the value assigned to T has an impact upon controller stability and performance. As T decreases, the performance index with the discrete-time controller tends to that of the designed continuous-time controller while as T increases, the difference between the performance indices obtained by continuous-time and discrete-time controllers will increase, and eventually the system with the discrete-time controller may become unstable. Determining the precise value to assign to the period parameter in order to balance control performance with schedulability concerns is a challenging problem that has been widely studied, and various algorithms proposed (see, e.g., [6], and the references cited there) for assigning period values.

[2] Although system dynamics may require the use of time-varying matrices upon "mode change" – e.g., if a car were moving from a wet to a dry surface – in this paper we restrict consideration to constant matrices only.

[3] *Zero-order hold* policies apply the computed value throughout, while *first-order hold* policies (in either basic form or the *delayed* or *predictive* variants) apply some reconstructed piecewise linear approximation.

2.2 Adapting Control to Facilitate Scheduling

When multiple controllers are implemented upon a shared computing platform, the period parameters –frequencies of invocation– of the different controllers must be selected to ensure that all the controllers together are *schedulable* upon the platform. As a first step to control-scheduler co-design, one needs to be cognizant of the tradeoff inherent in the choice of the controller period parameters between controller performance and the computational load it imposes upon the shared platform. This tradeoff was considered in the *elastic task model* [8, 9, 15, 16] by associating an *elastic parameter* with a controller that characterizes how resilient controller performance is to increases in its period parameter. The elastic task model assumes a linear relationship between the frequency at which a controller is invoked and its performance. Roy et al. [28] observed that this is an over-simplification for many controllers, and developed a search-based method of exponential time-complexity that characterizes each controller's performance as a function of its period via extensive simulation, and uses these characterizations to search for values for the period parameters of the controllers to optimize overall performance while ensuring schedulability.

Scheduling-adaptive control. The elastic scheduling approach, in both its original [8] and modified [28] forms, is inherently static in the sense that the periods of individual controllers, once selected, are fixed so long as the mix of controller tasks sharing the computing platform is unchanged. *Scheduling-adaptive* control represents a more nuanced approach to the design of controllers and to their run-time scheduling. The idea is that rather than associating a constant period T with a controller so that $a_{k+1} = a_k + T$ for all $k \in \mathbb{N}$, the value to be assigned to a_{k+1} is determined at time-instant a_k based upon the current scheduling load on the shared computing platform, in a manner that ensures that the platform does not become overloaded.

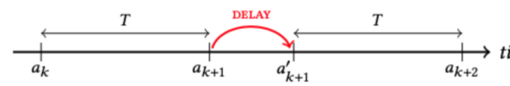
The advantage of such an adaptive approach is clear: it can provide superior control performance upon the same platform. However, it is in general *computationally too expensive* to compute the $A(h_k), B(h_k), C(h_k)$, and $D(h_k)$ matrices during run-time, after deciding upon a value for a_{k+1} (and thus the value of $h_k \equiv a_{k+1} - a_k$). So the practice is to pre-compute these matrices for a few selected values of h_k, and during run-time choose a_{k+1} such that $(a_{k+1} - a_k)$ is one of these values for which the matrices have been pre-computed. Once this value of h_k is chosen, the k'th job then simply performs the matrix computations

$$\begin{aligned} z[k+1] = &\quad A(h_k)z[k] + B(h_k)e[k] \\ u[k+1] = &\quad C(h_k)z[k] + D(h_k)e[k] \end{aligned} \tag{3}$$

using the appropriate precomputed (and stored) matrices. In prior work, Cervin et al. [13] have explored the issue of how to change period parameters dynamically during run-time, and [14, 19] present algorithms for computing multiple period parameters per task and cycling through these pre-computed periods in a pattern that optimizes control performance without compromising schedulability.

2.3 Mitigative Control

Pazzaglia et al. [26] propose an alternative form of scheduling-adaptivity in control whereby the value of the period is set to a constant T (as with the approaches [8, 9, 15, 16, 28] based on the elastic tasks model), and the controller invocation at time a_k computes a control signal that is intended for application to the plant at time-instant $a_{k+1} = a_k + T$. However, *mitigative control* allows for the possibility that the control signal that was intended for use at time-instant a_{k+1} will not have been computed by then. In that case, the previously-assigned value of a_{k+1} (at which instant the control signal was intended to be applied to the plant) is delayed until after the completion of this computation. Let a'_{k+1} denote the actual time-instant at which the computation of this control signal finally completes and is applied to the plant. The next invocation of the control task occurs at time-instant a'_{k+1}: this invocation computes a control signal that is intended to be applied to the plant at time-instant $a_{k+2} \stackrel{\text{def}}{=} (a'_{k+1} + T)$:



This next control signal is computed so as to account for, and mitigate the effect of, the delay: *"the control strategy of each job is adjusted to compensate the amount of the [...] overrun experienced by the previous job"* [26].

This form of mitigative control allows us to be less conservative from a scheduling perspective: rather than needing to guarantee, as fixed-period or scheduling-adaptive control must, that the computation that commenced at time-instant a_k completes by time a_{k+1} under all processor load conditions including ones that are highly unlikely to occur in practice, under mitigative control we only need to deal with exceptionally poor processor-load conditions if they actually occur. (An alternative way of looking at this is that since the deadlines for individual jobs are no longer hard deadlines that can never be missed, we can consider scheduling for such mitigative control to be a *soft*-real-time scheduling problem rather than a hard-real-time one.)

Since under mitigative control the control signal computed by each invocation of the controller is designed to compensate for the overrun, if any, of the previous invocation of the controller, it depends upon the degree of such overrun — i.e., the duration of the delay. Allowing this duration to take on arbitrary values would require that the controller doing so be designed at run-time rather than beforehand; as previously, the workaround is to only allow for a few permitted values for this delay, and at run-time to simply round up the actual delay to the next-higher permitted one.

3 Scheduling for Mitigative Control

In this section we present an algorithm for implementing multiple mitigative controllers upon a shared platform. We assume that a given collection $\tau = \{\tau_i, \tau_2, \ldots, \tau_n\}$ of mitigative controller tasks are to be implemented upon a single preemptive processor. Each task τ_i is characterized by a positive integer worst-case execution time C_i ; each invocation of the controller takes an execution duration that is guaranteed to not exceed C_i.

The WCET Problem. It is widely known that the execution durations of pieces of code (such as the code implementing our controller tasks) tend to exhibit a good deal of variation and unpredictability, particularly upon modern processors – see Figure 2. The *WCET problem* [33], the problem of determining a safe upper bound

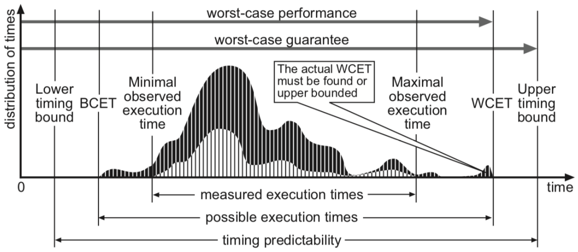


Figure 2: This figure, from [33], illustrates how the execution time of a single piece of code may vary when executed repeatedly upon a modern processor.

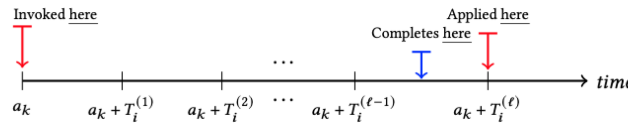


Figure 3: The control computation initiated at time-instant a_k completes at some instant between $a_k+T_i^{(\ell-1)}$ and $a_k+T_i^{(\ell)}$, and so is applied to the plant at time-instant $a_k + T_i^{(\ell)}$.

upon the worst-case execution time (WCET) of pieces of code, is a widely-studied problem in real-time computing. It is known that safe upper bounds on WCET tend to be extremely conservative in the sense that pieces of code very rarely, if ever, actually execute for a duration as large as their WCET values as determined by a WCET-analysis tool that is certified for use in validating highly safety-critical systems.[4] Hence provisioning computing capacity to allow each controller task invocation to execute for a duration up to its WCET C_i, is likely to result in considerable computing resource under-utilization during runtime. So rather than doing so, we instead leverage the mitigative capabilities of our controllers in order to be more aggressive in provisioning computing resources by making more optimistic assumptions about the actual execution duration of the task invocations. We are able to do this because under mitigative control the consequences of being incorrect about the execution duration (and thereby being unable to apply the control signal at the expected instant) can be mitigated by the next control signal that will be applied. This allows for more aggressive scheduling decision-making since negative consequences of being incorrect can be remedied.

3.1 The Task Model

As stated above, we assume that we are given a collection $\tau = \{\tau_i, \tau_2, \ldots, \tau_n\}$ of *mitigative controller tasks* that are to be implemented upon a single shared preemptive processor. Each task τ_i is characterized by a single[5] positive integer worst-case execution time (WCET) C_i and multiple positive integer period parameters $\overrightarrow{T_i} = [T_i^{(1)}, T_i^{(2)}, \ldots, T_i^{(n_i)}]$; we assume without loss of generality that these are indexed in increasing order: $T_i^{(j)} < T_i^{(j+1)}$ for all j. (Sec. 4 discusses how these parameter values are assigned.) The interpretation of these parameters is as follows:

- The controller is first invoked at time-instant zero. Successive invocations happen at the instant that a computed control signal is applied to the plant.
- Each invocation of the controller takes an execution duration that is guaranteed to not exceed C_i. It is expected that most invocations of τ_i will have an execution duration far smaller than C_i.
- Suppose that the controller is invoked at some instant a_k. The control signal that is computed by this invocation is *intended* to be applied at time-instant $(a_k + T_i^{(1)})$. However in the event that the job has not completed execution by then, controller implementations are provided that can *compensate for* the effects of being tardy by amounts $\left(T_i^{(\ell)} - T_i^{(1)}\right)$ for each $\ell, 1 < \ell \leq n_i$, in the next control signal that will be computed –see Figure 3. Hence the control signal may be applied at time-instant $a_k + T_i^{(\ell)}$ for any $\ell, 1 < \ell \leq n_i$ – the sooner it is applied, the better the performance (since control is then applied more frequently: as stated above, the next invocation of the controller occurs the instant the current control signal is applied to the plant).
- The invocation must complete its execution by time-instant $a_k+T_i^{(n_i)}$: not doing so represents a failure since controller implementations are not provided that can compensate for tardiness exceeding $\left(T_i^{(n_i)} - T_i^{(1)}\right)$.

Observe that the control designer has some freedom in choosing both the number n_i and the values $T_i^{(1)}, T_i^{(2)}, \ldots, T_i^{(n_i)}$ of the period parameters; we will address the issue of making these choices in Section 4 below, where we will see that this is again a co-design problem where considerations of control are scheduling should both be taken into account.

3.2 The Scheduling Algorithm

We first define a *utilization parameter* U_i for each mitigative controller task τ_i, as follows:

$$U_i \stackrel{\text{def}}{=} \left(C_i / T_i^{(n_i)}\right) \tag{4}$$

Since each invocation of τ_i may require up to C_i time units to complete execution and successive invocations must occur no further than $T_i^{(n_i)}$ time apart, U_i denotes a lower bound on the fraction of the processor computing capacity that must be guaranteed for executing task τ_i in the worst case (i.e., in the unlikely circumstance that every one of its invocations actually takes C_i time units to complete execution).

[4]In Figure 2, the value that would be determined by such a tool is the one labeled "worst-case guarantee". It represents the best upper bound that can be authoritatively established on the maximum duration the code will take to execute over all circumstances under which the system is required to behave correctly. (Please see [33] for additional details.)

[5]For simplicity, we assume here that the computational cost of the mitigative actions are substantially smaller than that of computing the control signal, and hence C_i is the same regardless of which version of the controller is executed. This assumption is easily removed at some slight increase in the complexity of our proposed algorithm.

Given a collection $\{\tau_1, \tau_2, \ldots, \tau_n\}$ of mitigative controller tasks that satisfy the *schedulability condition* (or *admission control* property)

$$\sum_{i=1}^{n} U_i \leq 1 \tag{5}$$

that are to be implemented upon a shared preemptive processor, we associate a *server*[6] S_i with each task τ_i. Each server S_i is characterized by a *scheduling deadline* D_i and an *execution budget* B_i. During run-time, some of the servers are designated as being *active* (the events that cause a server to be so designated are described below); at each instant in time, the active server with the earliest scheduling deadline is selected for execution upon the processor. (In other words, these servers are prioritized for execution according to *preemptive earliest deadline first* (EDF) [18, 23].) Let us explain the workings of these servers by stepping through a simple example scenario. At each instant in time each server is in one of the three states depicted in Figure 4 (with INACTIVE being the initial state).

- Suppose that the controller τ_i is invoked at some time-instant a_k while server S_i is in the INACTIVE state. This causes server S_i to be designated as being active, and to transition to the CONTEND state. Values are associated to scheduling deadline D_i and budget B_i as follows:

  $$D_i \leftarrow (a_k + T_i^{(1)}),\ B_i \leftarrow (T_i^{(1)} \times U_i).$$

  In so doing, we are making the optimistic assumption that the actual execution time of this invocation of the server will not exceed $(T_i^{(1)} \times U_i)$, in which case (as we will see below) our server can guarantee to complete its execution by time-instant $(a_k + T_i^{(1)})$.
- As stated above, the active server with the earliest scheduling deadline is selected for execution, where executing server S_i corresponds to executing the invocation of controller τ_i. Therefore at each instant in time while there is some controller task needing execution exactly one of the active servers – the one with the smallest value of its scheduling deadline parameter – is in the EXECUTE state, while the other active servers are in their respective CONTEND states. While server S_i is executing (i.e., while it is in its EXECUTE state), its budget B_i gets depleted at a unit rate. The transition between the CONTEND and EXECUTE states is determined entirely by the scheduler.
- It follows from Lemma 1 (below) that one of the following two events is guaranteed to occur prior to D_i: either (i) the controller invocation completes execution; or (ii) the budget B_i is depleted to zero.

(1) If the controller invocation of τ_i completes execution before the budget has been entirely depleted (i.e., before B_i has become equal to zero), then (i) server S_i transitions from the EXECUTE state to the INACTIVE state; (ii) the control signal that was computed by the just-completed controller invocation will be applied to the plant at time-instant $(a_k + T_i^{(1)})$; and (iii) the next invocation of τ_i is scheduled for time-instant $(a_k + T_i^{(1)})$ — at that instant, it will once again transition from the INACTIVE to the CONTEND state.

(2) If however the budget B_i becomes equal to zero prior to the controller invocation completing its execution then its scheduling deadline D_i is increased to $(a_k + T_i^{(2)})$, and its execution budget replenished to $(T_i^{(2)} - T_i^{(1)}) \times U_i$.
The rationale for this is as follows. The budget B_i becoming equal to zero indicates that our optimism that the execution duration of the task invocation would not exceed $(T_i^{(1)} \times U_i)$ was unwarranted – it executed for this duration without completing. We therefore make another (still optimistic) assumption that the actual execution time of this invocation of the server will not exceed $(T_i^{(2)} \times U_i)$ – this is achieved by setting B_i to $(T_i^{(2)} - T_i^{(1)}) \times U_i$ — in which case our server can guarantee to complete its execution by time-instant $(a_k + T_i^{(2)})$.
Increasing the value of D_i in this manner may cause S_i to no longer be the earliest-deadline server, in which case the scheduler would cause it to transition to its CONTEND state and some other active server (the one with the current earliest scheduling deadline) would enter its own EXECUTE state.

- Suppose the latter of the two possibilities above had occurred: the budget became equal to zero prior to the controller invocation completing execution. It again follows from Lemma 1 that one of the following two events is guaranteed to occur prior to D_i: either the controller invocation completes execution, or the budget B_i is depleted to zero.
  If the controller invocation completes execution before the budget has been depleted then server S_i transitions to its INACTIVE state and the next invocation of τ_i is scheduled for time-instant $(a_k + T_i^{(2)})$, while if the budget B_i becomes equal to zero before completion then D_i is increased to $(a_k + T_i^{(3)})$ and budget B_i replenished to $(T_i^{(3)} - T_i^{(2)}) \times U_i$
- In general, assume that server S_i's scheduling deadline D_i was last set to be equal to $(a_k + T_i^{(\ell)})$. It follows from Lemma 1 that one of the following two events is guaranteed to occur prior to D_i: either the controller invocation completes execution, or the budget B_i is depleted to zero.
  (1) If the controller invocation completes execution first, then (i) server S_i transitions from its EXECUTE state to its INACTIVE state, and (ii) the next invocation of τ_i is scheduled for time-instant $(a_k + T_i^{(\ell)})$.
  (2) If however the budget becomes equal to zero prior to the controller invocation completing, then the scheduling deadline D_i is increased to $(a_k + T_i^{(\ell+1)})$, and the budget replenished to $(T_i^{(\ell+1)} - T_i^{(\ell)}) \times U_i$.

  And what if the budget becomes equal to zero but D_i is already equal to $(a_k + T_i^{(n_i)})$? Lemma 2 below will show that in this case the invocation of τ_i has executed for a duration greater than C_i without completing — i.e., the WCET estimate for τ_i was incorrect. We assume that this corresponds to an *error condition* that must be handled outside the framework of this run-time scheduling algorithm: hence the scheduler flags an error and places the server S_i in its INACTIVE state.

[6] See, e.g., [7, Ch 5–6] for a textbook introduction to servers. We are not the first to consider the use of servers to service control tasks – see, e.g., [4] and some of the references cited therein. However, to our knowledge we are the first to propose their use for servicing *mitigative* control tasks.

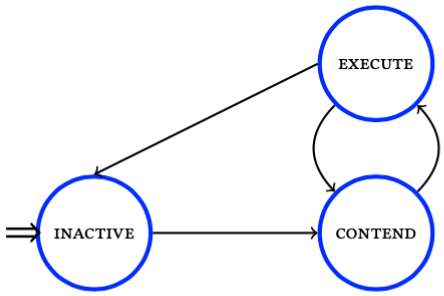


Figure 4: Server States: an inactive server in the states designated " INACTIVE", WHILE AN ACTIVE SERVER IS IN ONE OF THE TWO STATES DESIGNATED " CONTEND" AND " EXECUTE".

```
invokeController(τ_i, a_k)
   // Controller task τ_i is invoked at time-instant a_k
1  Transition from INACTIVE state to CONTEND state
2  ℓ = 1; done = FALSE
3  replenishServer(τ_i, a_k, ℓ)
4  repeat
5     if (the invocation completes execution)
6        Transition from EXECUTE state to INACTIVE state
7        Schedule τ_i's next invocation for time a_k + T_i^(ℓ)
8        done = true
9     else
10       if (B_i becomes equal to zero)
11          ℓ = ℓ + 1
12          replenishServer(τ_i, a_k, ℓ)
13 until done

replenishServer(τ_i, a_k, ℓ)
1  if (ℓ > n_i) // Error condition: WCET was incorrect
2     Transition from EXECUTE state to INACTIVE state
3     error "WCET bound C_i exceeded"
4  D_i = a_k + T_i^(ℓ)
5  B_i = (T_i^(ℓ+1) − T_i^(ℓ)) × U_i
6  if (ℓ == 0)
7     B_i = T_i^(1) × U_i
8  else
9     B_i = (T_i^(ℓ+1) − T_i^(ℓ)) × U_i
```

Figure 5: Pseudo-code representation of the server S_i run-time algorithm

The algorithm discussed above is represented in pseudo-code form in Figure 5. All the controllers are invoked at time-instant zero (i.e., INVOKECONTROLLER$(\tau_i, 0)$ is called for each $i, 1 \le i \le n$), and each τ_i is subsequently invoked at the instants specified in Line 7 of the pseudocode of procedure INVOKECONTROLLER(τ_i, a_k). Upon each such invocation, initializing the budget and the server deadline is done by the call to REPLENISHSERVER$(\tau_i, a_k, 1)$ in Line 3; subsequent budget replenishments and the corresponding deadline postponements are done in subsequent calls to REPLENISHSERVER(τ_i, a_k, ℓ) in Line 12.

Proof of correctness

We will now show that the algorithm described above is correct in the following sense (see Theorem 1): if $\left(\sum_{i=1}^{n} U_i \le 1\right)$ and the WCET estimates are correct in the sense that no invocation of any task τ_i needs more than C_i time units to complete execution, then each invocation of each τ_i completes within a duration $T_i^{(n_i)}$ of its invocation. In the remainder of this section let us assume that $\left(\sum_{i=1}^{n} U_i \le 1\right)$, i.e., Condition 5 is satisfied.

Lemma 1. *Suppose that the server S_i transits out of its INACTIVE state at time-instant a_k, and has not since returned to INACTIVE at some time-instant $t_{\text{cur}} > a_k$. It must be the case that t_{cur} is no larger than the current value of D_i at time t_{cur}.*

Proof. Let the value of D_i at time-instant t_{cur} equal $(a_k + T_i^{(\ell)})$. Since the budget B_i is initially assigned the value $U_i \times T_i^{(1)}$ and replenished by an amount $U_i \times (T_i^{(j)} - T_i^{(j-1)})$ when D_i is increased from $a_k + T_i^{(j-1)}$ to $a_k + T_i^{(j)}$ for each j, the cumulative budget that has been assigned to S_i over the interval $[a_k, a_k + T_i^{(\ell)})$ is given by

$$U_i \times T_i^{(1)} + U_i \times \left(T_i^{(2)} - T_i^{(1)}\right) + \cdots + U_i \times \left(T_i^{(\ell)} - T_i^{(\ell-1)}\right)$$
$$= U_i \times T_i^{(\ell)}$$

Hence, server S_i's budget request could be satisfied in its entirety by its deadline, if a fraction U_i of the processor were to be reserved for S_i's use.

Repeating the above argument for all the servers, it follows from Condition 5 that there always exists a *processor-sharing schedule* in which each server S_j is assigned a dedicated fraction U_j of the processor capacity, such that each budget request of each server is satisfied in its entirety by its deadline.

It therefore follows from the optimality property of preemptive uniprocessor EDF [18, 23] that if Condition 5 is satisfied then each budget request of each server is satisfied in its entirety by its deadline if the servers are prioritized according to their deadlines (as indeed they are in our framework). □

Lemma 2. *Each invocation of τ_i is guaranteed to have either completed or received at least C_i units of execution within an interval of duration $T_i^{(n_i)}$ since its invocation.*

Proof. As we saw in the proof of Lemma 1, the cumulative budget that has been assigned to server S_i when the value of D_i is equal to $a_k + T_i^{(n_i)}$ equals $U_i \times T_i^{(n_i)}$; by definition of U_i (Expression 4), this equals C_i. It therefore follows that the invocation of τ_i that triggered the transition of S_i out of its INACTIVE state at some time-instant a_k will definitely have completed execution or received at least C_i units of execution when the value of D_i is equal to $a_k + T_i^{(n_i)}$. □

Theorem 1 immediately follows.

Theorem 1. *If the WCET parameters are correct — i.e., each invocation of each task τ_i completes upon receiving no more than*

C_i units of execution — and

$$\sum_{i=1}^{n} U_i \leq 1 \tag{6}$$

then each invocation of τ_i is guaranteed to complete within an interval of duration $\leq T^{(n_i)}$ since its invocation. □

It is worth pointing out another consequence of Lemma 2: the effect of a mis-parametrized control task —some task for which the C_i parameter value turns out to in fact not be an upper bound on its WCET— is limited to that task only. That is, while such a mis-parametrized task may report an error (line 3 of the REPLENISHSERVER(τ_i, a_k, ℓ) procedure in Figure 5), this will not impact the correct execution of servers for which the parameter values are correct. In other words, our scheduling algorithm is ***robust to WCET errors*** in the sense that underestimating the WCET of some control tasks does not compromise the correct execution of the remaining (correctly characterized) tasks. It is hence perfectly acceptable to be somewhat more optimistic in characterizing the WCETs of less safety-critical tasks: while they may fail to perform correctly if the optimism turns out to be unwarranted, correctness of the safety-critical tasks is not compromised.

As a pragmatic improvement to the design of our scheduling algorithm, we may also wish to incorporate the idea behind the GRUB (for *"Greedy Reclamation of Unused Bandwidth"*) server [1, 22] that is available as part of the Linux OS.[7] Under GRUB, excess processor capacity (e.g., an amount $(1 - \sum_i U_i)$, if this exceeds zero, plus the budget that is left over if some invocation completes without exhausting its budget) is allocated to the currently-executing server. It has been shown that such a greedy reclamation strategy tends to hasten completion time on average.

4 Choosing n_i and the $T_i^{(\ell)}$ values

In Section 3 above, we characterized each mitigative control task by its WCET and multiple period parameters. We observed that the value to be assigned to the WCET parameter is determined using a WCET-analysis tool, but what determines the values assigned to the period parameters? This of course depends upon the particulars of the controller that is being modeled. For instance, in the situation considered by Pazzaglia et al. [26] the plant output (the "$y(t)$" is available only at particular periodic time-instants and hence the $T_i^{(\ell)}$ values must all be synchronized to coincide with these time-instants. In a more general setting, though, one can envision controllers that are able to sense the plant output at any time, in which case the choice of n_i is likely to be determined by the design effort (how many different controllers, each mitigating for a different degree of error, do we want to design?) and, when implemented upon memory-limited embedded platforms, perhaps upon considerations of how much storage we wish to devote to storing multiple different controller implementations. We now propose an heuristic approach for assigning values to the $T_i^{(\ell)}$ parameters for such controllers and for a chosen value of n_i.

[7] See, e.g., https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html; accessed October 29, 2021.

(1) We set the longest period, $T_i^{(n_i)}$, to the largest value that guarantees stability and the minimum acceptable level of performance.[8] The utilization parameter U_i of τ_i is then set equal to $C_i/T_i^{(n_i)}$, where C_i denotes the WCET as determined by some high-integrity WCET-analysis tool. Hence each invocation of τ_i is guaranteed to complete within an interval of duration $T_i^{(n_i)}$ of its invocation (provided it executes for no more than the WCET).

(2) We conduct extensive simulation experiments by executing the controller under a wide range of conditions (i.e., by having it execute concurrently with different mixes of other workloads) and measuring its execution duration in order to obtain a profile, similar to the one depicted in Figure 2, of its actual execution-time.

Based on this observed profile, let us define a function $P : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{[0,1]}$ to denote the cumulative distribution function (CDF) of the observed execution duration:

$$P(t) \stackrel{\text{def}}{=} \text{Probability}\Big[\text{Execution duration} \leq U_i \times t\Big] \tag{7}$$

Here the scaling down by U_i reflects the fact that server S_i is only guaranteed a fraction U_i of the processor capacity. In essence, $P(t)$ seeks to represent the probability that a random invocation of τ_i will complete within an interval of duration t upon a dedicated speed-U_i processor.

(3) We determine, by extensive simulation, a quantitative measure of the controller's performance, including its mitigation actions, when it is executed at different frequencies (i.e., with different inter-invocation periods) —this process is illustrated on an example in Section 5 below. Let $I_E(t)$ denote the performance obtained when the selected period is t; without loss of generality let us assume smaller values of $I_E(t)$ correspond to better performance.

(4) Under the scheduling algorithm described in Section 3 above, a job of the task that completes execution t time units after its invocation where $T_i^{(\ell-1)} < t \leq T_i^{(\ell)}$, will be next invoked $T_i^{(\ell)}$ time units after the previous one and thereby achieve a performance of $I_E(T_i^{(\ell)})$. Defining $T_i^{(0)} \stackrel{\text{def}}{=} 0$ for notational convenience, we should therefore choose values for $T_i^{(1)}, \ldots, T_i^{(n_i-1)}$ to **minimize** the following objective:

$$\sum_{\ell=1}^{n_i} \left(I_E(T_i^{(\ell)}) \times \left(P(T_i^{(\ell)}) - P(T_i^{(\ell-1)}) \right) \right) \tag{8}$$

since for each $\ell, 1 \leq \ell \leq n_i$, a performance of $I_E(T_i^{(\ell)})$ is obtained with probability $P(T_i^{(\ell)}) - P(T_i^{(\ell-1)})$.

The precise manner in which this optimization problem is to be solved depends upon various factors, primarily the properties of the functions $P(\cdot)$ and $I_E(\cdot)$. Specific solution techniques can be developed for various specific forms of the $P(\cdot)$ and $I_E(\cdot)$ functions. For instance if the function $I_E(t)$ is only defined for a (relatively small) discrete set of values of t, then the optimization problem of finding values for the $T_i^{(\ell)}$'s that minimizes Expression 8 can be solved by exhaustive search – this is illustrated in Section 5 below.

[8] Note that we are assuming that each task represents an *independent* control task. We leave consideration of stability and performance in systems in which the different control tasks interact with each other to future work..

h_k	10	15	20	25	30	35	40	45	50
I_E	5.847	5.913	6.307	6.496	7.162	7.761	9.174	11.57	19.51

Table 1: Performance index I_E (smaller is better) as a function of period h_k

5 Experimental Evaluation

In Section 3 above, we showed that our scheduling framework essentially guarantees each individual mitigative controller task τ_i a fraction U_i of the processor capacity – it guarantees to provide τ_i at least as much execution as τ_i would receive if it were executing upon a dedicated slower processor of speed U_i. Hence to understand the controller performance versus schedulability tradeoff that is inherent in our proposed co-design approach, it suffices to consider the performance of a single controller task when scheduled according to our framework. Hence we experimentally evaluated the algorithm and the period-selection heuristic described in Sections 3 and 4 above upon an example controller that had previously been introduced by Roy et al. [28] earlier this year. This is a second-order DC motor speed-control system with the following continuous-time dynamics:

$$\dot{x}(t) = \begin{bmatrix} -10 & 1 \\ -0.02 & -2 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 2 \end{bmatrix} u(t)$$

and output

$$y(t) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} x(t)$$

We model this system as a discrete-time system with sampling time $10ms$ (millisecond) and a one-sample delay:

$$\psi[k+1] = \Phi\psi[k] + \Gamma u[k],\ \psi[0] = \begin{bmatrix}1, 0, 0\end{bmatrix}^T$$

where $\psi[k] = \begin{bmatrix} x[k] \\ u[k-1] \end{bmatrix}$ and $u[k] = -K\psi[k]$ with

$$K = [157.9898, 17.07916, 0.0850342]$$

(please see [28] for additional details), to place the closed-loop poles for the system at 0.6, and use as performance index I_E the quadratic cost over a finite horizon of 100 iterations:

$$I_E \stackrel{\text{def}}{=} \sum_{k=1}^{100} \psi[k]^T Q \psi[k],\ \text{where } Q = \text{diag}\{1, 10^{-2}, 10^{-5}\}$$

Methodology. We now describe our experimental methodology. We first simulated the behaviour of the controller at various different (fixed) periods in multiples of $5ms$ in order to determine the largest period at which it could be invoked periodically and retain stability – see Figure 6. These experiments reveal that our controller exhibits stable behavior for periods up to $50ms$, but not beyond that.

We then determined, again via simulation experiments, the performance index values when the controller is invoked periodically at several different periods[9] in the range $[10ms, 50ms]$. These performance index values are listed in Table 1.

We modeled the actual distribution of execution times as following a Weibull distribution with parameters *shape* = 2.0, *location* = 4.0, and *scale* = 15; the probability density function (pdf) and cumulative distribution function (cdf) ($P(t)$ as defined in Expression 7) of the consequent time to completion are depicted by the red lines in Figure 8.

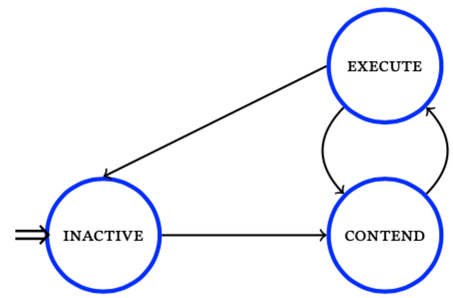


Figure 6: Observations from simulations for determining the range of frequencies over which the controller is stable. The curves plot system state $y(t)$ as a function of time t for different choices of controller invocation periods: the curve for invocation period $55ms$ oscillates without converging.

Choosing the $T_i^{(\ell)}$ values. We used the simulated performance index values of Table 1, and the *Weibull*(2.0, 4.0, 15) execution-time distribution model, to assign values to the $T_i^{(\ell)}$ parameters. Note that we have nine values of h_k listed in Table 1 – these are all potential $T_i^{(\ell)}$ values. Suppose that we are restricted to five distinct periods ($n_i = 5$). As stated in Section 4, $T^{(n_i)}$ should be set equal to the largest value for which stability and a minimum level of performance is guaranteed; in our example, therefore, $T_i^{(5)} \leftarrow 50$. It is reasonable to set $T_i^{(1)}$ to 10 (since the controller was designed with an intended period of 10); this leaves us to choose an additional three periods from the seven remaining values $\{15, 20, 25, 30, 35, 40, 45\}$. By exhaustive enumeration — i.e., computing the objective function of Expression 8 for all $\binom{7}{3} = \mathbf{35}$ choices of the three intermediate periods — we determined that the expected value of the performance index takes of its minimum value of 6.584 when the selected periods are 15, 25, and 35:

$$\overrightarrow{T_i} = [10, 15, 25, 35, 50] \tag{9}$$

Experiments and Observations. We performed two sets of experiments, one assuming controller execution times are indeed drawn from the *Weibull*(2, 4, 15) distribution as was assumed in choosing the periods, and another when they are not.

§1. For the first set of experiments, we considered three different choices of controller periods $\overrightarrow{T_i}$: one (Expression 9) determined as described in Section 4, a second ($\overrightarrow{T_i} = [10, 20, 30, 40, 50]$) obtained

[9] In this experiment we assume for simplicity that the control signal to the plant $u[k+1]$ is unchanged by such mitigation actions, which only modify the controller state $z[k+1]$ – see Expression. 3.

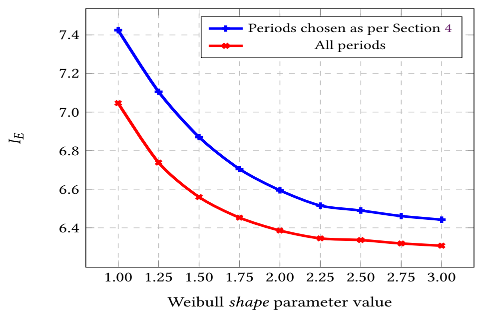


Figure 7: Robustness to error in modeling execution times: the impact of the shape parameter in Weibull distribution on performance index.

by choosing the intermediate periods uniformly, and a third that includes all the stable periods: $\overrightarrow{T_i} = [10, 15, 20, 25, 30, 35, 40, 45, 50]$ — this third choice, which is not a viable design option for implementation since it violates our constraint that we only have five distinct periods ($n_i = 5$), serves as a proxy for optimality. We measured the performance of the controller over simulated runs of one hundred invocations. Averaged over five thousand runs, the performance indices obtained for the three choices of periods are as follows:

Periods:	As in Expression 9	[10, 20, 30, 40, 50]	All
I_E	6.5942	6.6199	6.3859

This indicates that choosing periods as in Section 4, which yields the periods in Expression 9 for our running example, does not suffer too large a performance penalty in comparison to using all the available periods. This performance penalty may be a reasonable one to pay, particularly when noting that choosing all the periods almost doubles the value of n_i in our example (from five to nine). We also note that there is a slight performance benefit (about 10%) to choosing the periods according to the heuristic of Section 4 rather than just having them spaced uniformly apart.

§2. Our second set of experiments evaluate the robustness of our period-selection approach in the event that our modeling of the distribution of execution times (step 2 of the heuristic in Section 4) turns out to be inaccurate. To do so we determined controller performance when its actual execution duration is drawn from distributions that differ from the *Weibull*(2, 4, 15) model that was assumed when determining the choice of periods. An an illustrative example, Figure 8 depicts the modeled *Weibull*(2, 4, 15) probability density function (pdf) and cumulative distribution function (cdf) in red; the blue lines depict pdf and cdf for the *Weibull*(**3**, 4, 15) distribution from which the assumed execution durations were chosen. We obtained the following performance indices upon performing the same simulations as above but with execution durations drawn from the *Weibull*(3.0, 4, 15) distribution:

Periods:	As in Expression 9	[10, 20, 30, 40, 50]	All
I_E	6.4422	6.5017	6.3073

We repeated this experiment for the *Weibull*(x, 4, 15) distributions for various values of x; some of the results are depicted in Figure 7. These observations offer some evidence that our period-determination method may be quite robust to inaccuracies in the modeling of controller execution duration.

6 Context and Conclusions

Control systems have long been one of the most important motivating use-cases for the development of new models and algorithms in real-time scheduling theory (for instance, the seminal work of Liu & Layland [23] begins with the words "The use of computers for control and monitoring of industrial processes has expanded greatly in recent years. . . "). It is coming to be increasingly widely recognized (see, e.g., [26–28, 32]) that the real-time scheduling theory community needs to collaborate closely with control engineers to jointly develop models and algorithms that are more faithful to control considerations. The research reported in this manuscript was developed in this spirit. We looked at models of control that (i) need not be invoked in a rigidly periodic manner – rather, they are capable of dynamically adapting their frequency of invocation in response to constraints upon the availability of computational resources; and (ii) may make optimistic assumptions regarding execution duration since they are able to compensate for errors that are made in computing the control signal during one iteration of the control loop by taking corrective action during subsequent iterations. We applied principles from real-time scheduling theory to design and show correct a server-based scheduling framework for implementing such mitigative controllers in a manner that balances the need for effective control with the need for efficient and effective stewardship of computing resources, and experimentally demonstrated the effectiveness of this framework upon a simple control application.

Acknowledgments

This research was supported in part by the US National Science Foundation (Grants CPS-1932530, CNS-2141256, and CPS-2229290).

References

[1] L. Abeni, J. Lelli, C. Scordino, and L. Palopolii. 2014. Greedy CPU reclaiming for SCHED DEADLINE. In *Proceedings of the Real-Time Linux Workshop.*
[2] Mohammad Al Khatib, Antoine Girard, and Thao Dang. 2017. Scheduling of Embedded Controllers Under Timing Contracts. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control (HSCC '17)*. Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/3049797.3049816
[3] Amir Aminifar and Enrico Bini. 2017. Anomalies in scheduling control applications and design complexity. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, David Atienza and Giorgio Di Natale (Eds.). IEEE, 1607–1610. https://doi.org/10.23919/DATE.2017.7927247
[4] Amir Aminifar, Enrico Bini, Petru Eles, and Zebo Peng. 2016. Analysis and Design of Real-Time Servers for Control Applications. *IEEE Trans. Computers* 65, 3 (2016), 834–846. https://doi.org/10.1109/TC.2015.2435789
[5] Enrico Bini and Anton Cervin. 2008. Delay-Aware Period Assignment in Control Systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*. IEEE Computer Society, 291–300. https://doi.org/10.1109/RTSS.2008.45
[6] Enrico Bini and Anton Cervin. 2008. Delay-Aware Period Assignment in Control Systems. In *2008 Real-Time Systems Symposium*. 291–300. https://doi.org/10.1109/RTSS.2008.45
[7] Giorgio C. Buttazzo. 2005. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* (second ed.).
[8] Giorgio C. Buttazzo, Giuseppe Lipari, and Luca Abeni. 1998. Elastic Task Model for Adaptive Rate Control. In *IEEE Real-Time Systems Symposium.*

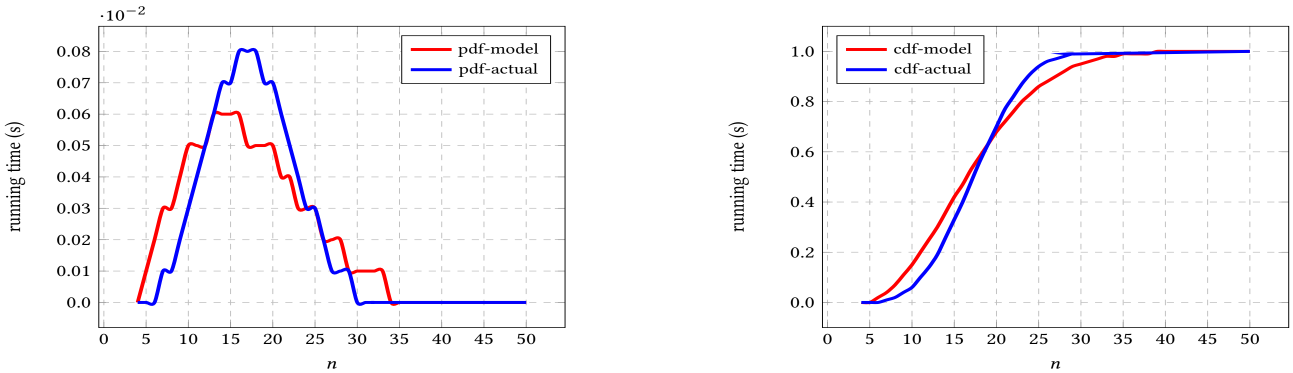


Figure 8: Modeled (in red) and actual (in blue) probability density function (pdf) and cumulative distribution function (cdf) of the duration a controller invocation takes to complete upon a speed-U_i processor in our experiments.

[9] Giorgio C. Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. 2002. Elastic Scheduling for Flexible Workload Management. *IEEE Trans. Comput.* 51, 3 (March 2002), 289–302. https://doi.org/10.1109/12.990127

[10] M. Caccamo, G. Buttazzo, and Lui Sha. 2002. Handling execution overruns in hard real-time control systems. *IEEE Trans. Comput.* 51, 7 (2002), 835–849. https://doi.org/10.1109/TC.2002.1017703

[11] Anton Cervin. 2005. ANALYSIS OF OVERRUN STRATEGIES IN PERIODIC CONTROL TASKS. *IFAC Proceedings Volumes* 38, 1 (2005), 219–224. https://doi.org/10.3182/20050703-6-CZ-1902.01076 16th IFAC World Congress.

[12] Anton Cervin and Johan Eker. 2005. Control-Scheduling Codesign of Real-Time Systems: The Control Server Approach. *J. Embedded Comput.* 1, 2 (April 2005), 209–224.

[13] Anton Cervin, Manel Velasco, Pau Marti, and Antonio Camacho. 2011. Optimal Online Sampling Period Assignment: Theory and Experiments. *IEEE Transactions on Control Systems Technology* 19, 4 (2011), 902–910. https://doi.org/10.1109/TCST.2010.2053205

[14] Wanli Chang, Dip Goswami, Samarjit Chakraborty, and Arne Hamann. 2018. OS-Aware Automotive Controller Design Using Non-Uniform Sampling. *ACM Trans. Cyber-Phys. Syst.* 2, 4, Article 26 (July 2018), 22 pages. https://doi.org/10.1145/3121427

[15] T. Chantem, X. S. Hu, and M. D. Lemmon. 2006. Generalized Elastic Scheduling. In *IEEE International Real-Time Systems Symposium.*

[16] T. Chantem, X. S. Hu, and M. D. Lemmon. 2009. Generalized Elastic Scheduling for Real-Time Tasks. *IEEE Trans. Comput.* 58, 4 (April 2009), 480–495. https://doi.org/10.1109/TC.2008.175

[17] Xiaotian Dai and Alan Burns. 2020. Period adaptation of real-time control tasks with fixed-priority scheduling in cyber-physical systems. *Journal of Systems Architecture* 103 (2020), 101691. https://doi.org/10.1016/j.sysarc.2019.101691

[18] Michael Dertouzos. 1974. Control Robotics : the Procedural Control of Physical Processors. In *Proceedings of the IFIP Congress.* 807–813.

[19] Sumana Ghosh, Souradeep Dutta, Soumyajit Dey, and Pallab Dasgupta. 2017. A Structured Methodology for Pattern Based Adaptive Scheduling in Embedded Control. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 189 (Sept. 2017), 22 pages. https://doi.org/10.1145/3126514

[20] Dip Goswami, Reinhard Schneider, and Samarjit Chakraborty. 2011. Co-design of cyber-physical systems via controllers with flexible delay constraints. In *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011).* 225–230. https://doi.org/10.1109/ASPDAC.2011.5722188

[21] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. 2001. Giotto: A Time-Triggered Language for Embedded Programming. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, Thomas A. Henzinger and Christoph M. Kirsch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 166–184.

[22] Giuseppe Lipari and Sanjoy Baruah. 2000. Greedy reclaimation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the EuroMicro Conference on Real-Time Systems.* IEEE Computer Society Press, Stockholm, Sweden, 193–200.

[23] C. Liu and J. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. ACM* 20, 1 (1973), 46–61.

[24] Alejandro Masrur, Sebastian Drossler, Thomas Pfeuffer, and Samarjit Chakraborty. 2010. VM-Based Real-Time Services for Automotive Control Applications. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications.* 218–223. https://doi.org/10.1109/RTCSA.2010.38

[25] Truong Nghiem, George J. Pappas, Rajeev Alur, and Antoine Girard. 2012. Time-Triggered Implementations of Dynamic Controllers. *ACM Trans. Embed. Comput. Syst.* 11, S2, Article 58 (Aug. 2012), 24 pages. https://doi.org/10.1145/2331147.2331168

[26] Paolo Pazzaglia, Arne Hamann, Dirk Ziegenbein, and Martina Maggio. 2021. Adaptive Design of Real-Time Control Systems Subject to Sporadic Overruns. In *Proceedings of DATE: Design, Automation and Test in Europe.*

[27] Paolo Pazzaglia, Claudio Mandrioli, Martina Maggio, and Anton Cervin. 2019. DMAC: Deadline-Miss-Aware Control. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Sophie Quinton (Ed.), Vol. 133. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:24. https://doi.org/10.4230/LIPIcs.ECRTS.2019.1

[28] Debayan Roy, Clara Hobbs, James H. Anderson, Marco Caccamo, and Samarjit Chakraborty. 2021. Timing Debugging for Cyber-Physical Systems. In *Proceedings of DATE: Design, Automation and Test in Europe.*

[29] Danbing Seto, John P. Lehoczky, Lui Sha, and Kang G. Shin. 1996. On task schedulability in real-time control systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA.* IEEE Computer Society, 13–21. https://doi.org/10.1109/REAL.1996.563693

[30] Damoon Soudbakhsh, Linh Thi Xuan Phan, Anuradha M. Annaswamy, and Oleg Sokolsky. 2018. Co-Design of Arbitrated Network Control Systems With Overrun Strategies. *IEEE Transactions on Control of Network Systems* 5, 1 (2018), 128–141. https://doi.org/10.1109/TCNS.2016.2583064

[31] Manel Velasco, Pau Martí, and Enrico Bini. 2008. Control-Driven Tasks: Modeling and Analysis. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008.* IEEE Computer Society, 280–290. https://doi.org/10.1109/RTSS.2008.29

[32] Nils Vreman, Anton Cervin, and Martina Maggio. 2021. Stability and Performance Analysis of Control Systems Subject to Bursts of Deadline Misses. In *2021 33rd Euromicro Conference on Real-Time Systems.* Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[33] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 7, 3, Article 36 (May 2008), 36:1–36:53 pages.

[34] Yang Xu, Anton Cervin, and Karl-Erik Årzén. 2018. Jitter-Robust LQG Control and Real-Time Scheduling Co-Design. In *2018 Annual American Control Conference (ACC).* 3189–3196. https://doi.org/10.23919/ACC.2018.8430953