Safety-Aware Implementation of Control Tasks via Scheduling with Period Boosting and Compressing

Shengjie Xu  
Department of Computer Science  
UNC Chapel Hill  
Chapel Hill, USA  
0000-0003-1784-1386

Bineet Ghosh  
Department of Computer Science  
UNC Chapel Hill  
Chapel Hill, USA  
0000-0002-1371-2803

Clara Hobbs  
Department of Computer Science  
UNC Chapel Hill  
Chapel Hill, USA  
0000-0001-6046-9511

P.S. Thiagarajan  
Department of Computer Science  
UNC Chapel Hill  
Chapel Hill, USA

Prachi Joshi  
Electrical and Controls Systems Laboratory  
General Motors  
Detroit, USA  
0000-0001-8299-1888

Samarjit Chakraborty  
Department of Computer Science  
UNC Chapel Hill  
Chapel Hill, USA  
0000-0002-0503-6235

Abstract—A crucial requirement for control tasks in safety-critical systems like automotive is that all deadlines be met. This is becoming increasingly difficult when several tasks share common resources. One main reason for this lies in obtaining tight WCET estimations, especially as software and processor architectures continue to become more complex. Using safe but not necessarily tight WCET estimates and meeting all deadlines come at the expense of very pessimistic and inefficient implementations. In this paper, we show that by focusing on “higher-level” properties like control safety, instead of trying to meet all deadlines, it is possible to achieve more efficient implementations of control tasks on shared resources. This has considerable benefits in cost-sensitive domains like automotive. The core of our technique follows the AUTOSAR paradigm where groups of control computations with the same period constitute units of scheduling. Towards this, we suitably increase (boost) or decrease (compress) the sampling periods of control tasks and schedule them in a manner that is cognizant of their high-level safety constraints, but does not necessarily meet all deadlines. Our results for several standard controllers from the automotive domain illustrate the benefits of our approach.

I. INTRODUCTION

a) Proposed scheme:: We propose a new technique for maximizing the number of control tasks that may be “packed” on a processor or resource. Our main novelty is a scheduling technique that satisfies “system-level” properties like control safety, described later in the paper, instead of aiming to meet all task deadlines. This shift in focus buys considerable implementation efficiency, which has not been explored in the past in the manner that we do. Figure 1 provides an overview of our approach. Given  1  a set of control tasks  \(T_1, T_2, \ldots\) with distinct sampling periods  \(P_1, P_2, \ldots\), our first step is to determine a common sampling period \(P^C\) by suitably increasing (boosting) or reducing (compressing) each period  \(P_1, P_2, \ldots\). This results in a new task set  2  that is then scheduled in a time-triggered manner on a resource where time is partitioned into slots of the chosen period size \(P^C\). Such a schedule  3  only allows a subset of tasks from the set  \(T_1, T_2, \ldots\) to be executed in each slot. The ones not scheduled miss their deadlines. In the example shown in Figure 1, the schedule for the task  \(T_1\) is 110110 . . . , that of  \(T_2\) is 010101 . . . ,  \(T_3\) is 101101 . . . , and finally, that of  \(T_4\) is 001001 . . . , where a 1 denotes the deadline being met and a 0 a deadline miss. Here, each slot is only large enough to execute at most two of the four tasks. Although there are several deadline misses, the schedule is derived in a manner that system-level properties of relevance, like the safety of the physical system being controlled, is not violated. If all the deadlines were to be met, then the full task set  \(T\) would not have been implementable on this single resource and more processors would be necessary. Given pre-designed controllers  1  and a safety property associated with each task  \(T_i\), how to obtain  3  while guaranteeing all the safety properties, is the main technical contribution of this paper.

b) Background and motivation:: Efficient implementation of software is a key to success in many cost-sensitive domains like automotive. Today, a modern car has several hundred million lines of software code implemented on different electronic control units (ECUs). The core functionality implemented by such code consists of various feedback control loops, e.g., engine control, brake control, cruise control, motor control, and suspension and vibration control. Here, the traditional implementation workflow consists of the control strategy being designed first, followed by implementing it as a software task, that is scheduled to meet the deadline determined during the controller design phase. This ensures a separation of concerns that enables control theorists and embedded systems engineers to only communicate via the deadlines that needed to be met.

However, meeting all deadlines—which is assumed in the above workflow—is turning out to be increasingly challenging. With growing software and processor architecture
complexity, estimating safe and tight worst case execution time (WCET) estimates of software tasks is becoming a losing proposition [1]. For WCET estimates to be safe, they are increasingly overestimated. Meeting all task deadlines with such overestimated WCET values leads to pessimistic or infeasible implementations. Further, automotive in-vehicle architectures are rapidly moving away from “one function per ECU” or federated, to multiple functions sharing resources, *viz.*, “integrated” architectures [2]. The clear trend is that future architectures will be less “static” than before, as indicated by developments like AUTOSAR Adaptive [3] and service-oriented paradigms [4]. Such trends necessitate decoupling the software from the underlying hardware architecture, to attain flexibility and ease of task migration across architectures. However, “architecture-independent” WCET estimates exacerbate the pessimism even further. As a result, all downstream scheduling techniques that rely on safe WCET estimates of tasks are becoming too pessimistic to remain useful in practice.

c) **Deadlines are only a means to an end:** This paper, therefore, asks the question—Can implementation pessimism be reduced by not having to meet all deadlines? In other words, can the focus be shifted to satisfying properties of consequence, instead of deadlines? In the context of our problem, they are “safety” properties defined on the physical system being controlled by the software tasks to be implemented and scheduled. We define them as follows: Given a plant and a suitable controller, let $\tau_{nom}$ be a trajectory in the state space of the closed-loop system (plant + controller) when the control task meets all deadlines. This is referred to as the ideal or nominal behavior. Any other trajectory $\tau$ is referred to as “safe” if it is at most a specified $d_{safe}$ distance away from $\tau_{nom}$, under a suitably defined distance metric. This is mathematically defined later in the paper. The intuition here is that occasional deadline misses will result in a different but acceptable state space trajectory, as long as its deviation is not much from the ideal one. As outlined at the start of this section, how to schedule multiple control tasks with different sampling periods and safety properties, in order to reduce implementation pessimism, is the key technical contribution of this paper.

For simplicity of exposition, we consider implementations on a single processor only. But the general scheme derived here can be extended to cases where control tasks are partitioned and implemented on multiple computation and communication resources. Our proposed scheme is also compliant with OSEK and AUTOSAR, where *runnables* or tasks with the same period are grouped together for scheduling [5], [6].

d) **Related work:** There is a considerable volume of literature on checking where control safety properties (including stability) are satisfied under a given deadline hit/miss pattern. Some of the representative recent literature on this include [7]–[11]. But there has been much less work on how to synthesize task schedules to meet control safety properties, and especially properties that are more general than stability, as we do in this paper. The work in [12] investigated scheduling with safety constraints, but the applicability of their methods is limited to controllers with the same period only. Our work is also closely related to scheduling using weakly-hard constraints (that specify deadline hit/miss patterns) [13] and its applications [9], [14], [15], as we discuss in the next section.

e) **Organization:** The rest of this paper is organized as follows: in Section II we introduce the necessary mathematical preliminaries and the models used in this work. We formally state the problem in Section III followed by our proposed approach in Section IV. Numerical results showing the benefits of this approach are in Section V. Finally, Section VII concludes the paper followed by an outline of possible extensions.

II. SYSTEM MODELLING

In this section, we introduce the necessary background on system modeling that will be utilized throughout the remainder of the work. We first wish to lay out the fundamentals of modeling control systems, then move on to the discussion about the characterization of deadline hit/miss patterns and how they affect system behavior such as control safety.

A. The State-Space Model

Control systems are dynamic in nature and are often described using differential equations. One common representation of control systems is the state-space model, where the *state of the system* is represented by a state vector $x(t) = [x_1(t) \ x_2(t) \ \ldots \ x_n(t)]^T$ and the *input to the system* by $u(t) = [u_1(t) \ u_2(t) \ \ldots \ u_p(t)]^T$. Using these notations, the state-space model of a continuous linear time-invariant system is given by

$$\dot{x}(t) = Ax(t) + Bu(t),$$

(1)

where $A \in \mathbb{R}^{n \times n}$, and $B \in \mathbb{R}^{n \times p}$. Eq. (1) shows that the rate of change of the system state ($\dot{x}(t)$) depends both on the current state ($x(t)$) and the control input ($u(t)$). When implemented on a processor, the state-space model is discretized, with a discretization period of $P$, and assumes the form of

$$x[t + 1] = A_d x[t] + B_d u[t].$$

(2)
In this work, we assume that the system model is closed-loop, where the measurement of the current state is used to determine the control input of the next actuation. In practice, the control input \( u \) is computed by a periodic real-time task running on a processor and is assumed to be of the form

\[
    u[t] = K x[t - 1],
\]

where \( K \in \mathbb{R}^{p \times n} \) is the feedback gain. We follow the logical execution time (LET) paradigm, where the deadline equals the sampling period. A new control input is always applied at the deadline of the control job, i.e., the system state is sampled at time \( t - 1 \) and used to compute the control input for time \( t \), where the state and control input is computed according to Eqs. (2) and (3).

### B. Characterizing Deadline Hit/Miss Patterns

The feedback gain matrix \( K \) in Eq. (3) is designed in tandem with the discretization period \( P \) of the system, and the correct behavior of the control system relies on the timely completion of the computation of the control input \( u \) by the end of each period. If the periodic real-time task computing the control input misses its deadline, then control performance suffers, and the system may deviate from its ideal behavior and become unsafe. However, not all deadline misses have the same impact, and many control systems can tolerate a certain amount of deadline misses before the system is considered unsafe. Many works have studied the characterization of deadline misses and how they impact control performance. Notably, the work in [13] proposes a systematic method for characterizing deadline hit/miss patterns. These so-called weakly-hard constraints—especially the \( \left( \frac{m}{k} \right) \) model, which demands that in any \( k \) consecutive invocations of a task, there can be at most \( m \) deadline misses—have been studied in a number of settings, including schedulability analysis, formal verification, and runtime monitoring, with [9], [14], [15] being some recent examples. In this work, we focus on constraints of the type \( \left( \frac{m}{k} \right) \) which states that there are at least \( m \) deadline hits in any \( k \) consecutive invocations of the task. This is equivalent to the constraint \( \left( \frac{k - m}{k} \right) \). Finally, as outlined in Section 4 if we represent the hit/miss patterns using a bit string, where \( 0 \) represents a deadline miss and \( 1 \) a deadline hit, then all hit/miss patterns that comply with the constraint \( \left( \frac{m}{k} \right) \) will be a regular language over the alphabet \( \{0, 1\} \). We shall denote this language as \( \mathcal{L}(m,k) \) and exploit the properties of regular languages to synthesize task schedules.

### C. System Behavior under Deadline Misses

We now characterize the effects of deadline misses on system behavior. Suppose \( x[t] \) is the plant state and \( u[t] \) is the control input at time \( t \). When \( x[t - 1] \) is read, a software job corresponding to the control task is released to compute \( u[t] \), which is then applied to the physical plant at time \( t \) if the job completes within its deadline. If the job is not scheduled (see Figure 1) then no new control input is computed and the previous control input continues to hold.

---

**Fig. 2.** An example of deviations as a result of deadline misses.

We consider the behavior of the plant only over a finite time horizon \( H \). Thus the states of the plant will be recorded at time points \( 0, 1, \ldots, H \). For ease of exposition, we also assume that the initial state of the system is \( z[0] \in \mathbb{R}^n \). With the initial state \( z[0] \), we define the nominal trajectory of the plant as the trajectory resulting from no deadline misses, denoted as \( \tau_{nom} \). Formally, it is the sequence of states of length \( H + 1 \) of the form \( x[0], x[1], \ldots, x[H] \) with \( x[0] = z[0] \), where \( x[t+1] \) is computed with Eq. (2) and \( u[t] \) is computed with Eq. (3). As an example, the black-colored trajectory in Fig. 2 is the nominal trajectory. We next wish to define the set of trajectories that do not deviate from the nominal trajectory \( \tau_{nom} \) by more than a safety bound \( d_{safe} \), shown as the light blue envelope in Fig. 2. Let \( \mathcal{T} = \{\tau\} \) be the set of sequences of length \( H + 1 \) over \( \mathbb{R}^n \) where \( \tau = (\tau[0], \tau[1], \ldots, \tau[H]) \) with \( \tau[i] = x[i] \), \( x[0] = z[0] \) and \( i = 0, \ldots, H \). Intuitively, \( \mathcal{T} \) denotes the set of all possible trajectories of length \( H + 1 \) in the state space that starts from \( z[0] \), including ones where some task deadlines are missed. Clearly, the nominal trajectory—where all deadlines are met—is also a member of \( \mathcal{T} \).

To quantify deviations from the nominal trajectory, we first define the distance between two points in \( \mathbb{R}^n \) using a distance metric \( d(\cdot) \). This can be any metric such as the Euclidean distance. We then define the distance between a pair of trajectories \( (\tau, \tau') \), also denoted as \( d(\cdot, \cdot) \), given by:

\[
    d(\tau, \tau') = \max_{0 \leq t \leq H} d(\tau[t], \tau'[t]).
\]

We now fix a safety margin \( d_{safe} > 0 \). This leads to the set of safe trajectories \( \mathcal{T}_{safe} \subset \mathcal{T} \), defined as

\[
    \mathcal{T}_{safe} = \{\tau \mid d(\tau, \tau_{nom}) \leq d_{safe}\}.
\]

Intuitively, this is the set of trajectories that do not exceed the safety margin around the nominal trajectory, i.e., trajectories that do not deviate more than \( d_{safe} \) from the nominal trajectory. For example, the green trajectory in Fig. 2 is a member of \( \mathcal{T}_{safe} \), while the red one is not. Clearly, the nominal trajectory is also a member of \( \mathcal{T}_{safe} \).

Finally, suppose \( \gamma \in \{0, 1\}^H \) is a sequence of length \( H \) representing a pattern of deadline hits and misses. Then starting from \( z[0] \) we can compute the sequence of plant states with Eq. (2) and control inputs with Eq. (3) if \( \gamma[t] = 1 \), or
Controllers with same period

No

and compressing

Period boosting

Controllers with same period

Yes

No

Yes

No

Yes

Tried all periods?

Yes

found schedule?

No

found schedule?

Schedule

Schedule

Synthesis

Synthesis

Period boosting and compressing (no redesign)

Period boosting and compressing (redesigning)

Fig. 3. An overview of the proposed period compression/boosting and schedule synthesis scheme.

The overview of our approach is outlined in Fig. 3. While converting all control tasks to the same sampling periods makes the safety analysis and scheduling problem much simpler, modifying sampling periods (either by boosting or compressing) changes the dynamics of the closed-loop system. This is further complicated by the fact that the tasks are implemented on a resource-constrained platform, and cannot be scheduled to meet all deadlines. This leads to some unintuitive effects on the schedulability of the system. For example, while a shorter sampling period generally enhances—or at least does not deteriorate—the control performance of a single system, it also means that fewer tasks can fit in that shorter period and will miss more deadlines. Therefore, the control period must be carefully determined to trade off the control performance of the systems with the number of tasks that fit within the period (i.e., not causing too many deadline misses). Despite the complications, we argue that when done carefully, the combined effect of the two can be used to produce a safe and more efficient implementation of control tasks. We now present the details of period compression and boosting and schedule synthesis in the next two subsections.

A. Period Compression and Boosting

Choosing a common period for all control tasks is a delicate issue. We proceed by considering the WCETs of each control task. Assume we are given a set of controller tasks. For each task \( T_i \), its WCET, period, and safety margin are denoted by \( C_i \), \( P_i \), and \( d_i^{safe} \), respectively. We assume that the task set is not schedulable on the shared processor when no deadline misses are allowed (e.g., the utilization \( U = \sum_i C_i/P_i > 1 \)), since otherwise, a standard scheduling algorithm such as earliest deadline first (EDF) will suffice. We first sort the execution times from large to small, and define potential common periods or slot sizes for scheduling (see Figure 1) as follows:

\[
P^C_k = \sum_{i \leq k} C_i.
\]

We choose these values to ensure that when we use \( P^C_k \) as the common time slot size, any \( k \) controller tasks can be executed within it. This is because it is large enough to fit the sum of the \( k \) largest WCET values. As an example, for the set of five controller tasks in Table III, the possible \( P^C_k \) values would be 15 ms, 28 ms, 40 ms, 50 ms, and 60 ms. As discussed earlier, it is important to note that the chosen \( P^C_k \)
will determine how much a control task is over- or under-sampled compared to its original design, and this by itself might violate a task’s safety property. Missing deadlines on top of it, by not scheduling the task at every slot, might only further deteriorate the violation and not correct it. But depending on the underlying dynamics of the closed-loop system, certain patterns of deadline misses might also correct a violation caused by over- or under-sampling, causing the combination of the two steps to be important and interesting.

a) Recomputing controller gains:: Finally, we have an additional design dimension: whether or not the controller gain values are recomputed based on the chosen common sampling period [17]. This is because gain values designed for the originally-intended sampling period might not work well for a new period. However, it may not always be realistic to redesign a controller for each implementation architecture, especially when controller design and implementation phases are handled by different teams. We explore both possibilities depending on the underlying dynamics of the closed-loop system.

For completeness, this scheme is summarized below:

in Sections II-B and II-C. For completeness, this scheme is summarized below:

1) For each controller, we find a set of weakly-hard constraints \((m_k)\) whose resulting trajectories \(T_{m,k}\) are safe.
2) We use these sets of constraints and properties of regular languages to synthesize a schedule satisfying the constraints for all controllers.

1) Safe constraints:: Given a control task \(T_i\) and its safety margin \(d_i^{safe}\), we wish to determine the set of weakly-hard constraints under which the system is safe. Checking safety under a particular constraint \((m_k)\) amounts to checking if \(T_{m,k} \subseteq T_{safe}\), or whether \(d(m,k) \leq d_i^{safe}\). Here, \(d(m,k)\) is the maximum deviation of the trajectories in \(T_{m,k}\) from the nominal trajectory of \(T_i\). More precisely,

\[
d(m,k) = \max\{\text{dis}(\tau, \tau_{nom}) \mid \tau \in T_{m,k}\}. \tag{8}
\]

However, checking this directly is expensive, due to the exponential number of hit/miss patterns of length \(H\). To get around this, we compute an upper bound \(\bar{d}(m,k)\) on \(d(m,k)\) using the BoundedRuns algorithm proposed in [7]. It then suffices to check that \(\bar{d}(m,k) \leq d_i^{safe}\) to guarantee the safety of the system.

Equipped with the method to check a single constraint \((k)\), we move on to generate the set of safe constraints by iterating through all weakly-hard constraints \((m_k)\) up to a maximum window size \(k_{max} (\ll H)\), a parameter fixed by the user. For each constraint \((m_k)\), we compute \(\bar{d}(m,k)\) using the BoundedRuns algorithm and compare it with \(d_i^{safe}\). If \(\bar{d}(m,k) \leq d_i^{safe}\), we conclude that the system is safe under \((m_k)\) and add \((m_k)\) to the set of safe constraints. Otherwise, no conclusion can be drawn and we do not add it to the set.

We additionally apply an optimization scheme that reduces the required number of iterations to generate the list based on the following observations:

\[
m_1 \geq m_2 \implies L_{(m_1,k)} \subseteq L_{(m_2,k)} \tag{9}
\]

\[
k_1 \geq k_2 \implies L_{(m,k_1)} \supseteq L_{(m,k_2)} \tag{10}
\]

These observations imply that

1) \((m_k)\) is safe if \((m_2)\) is safe and \(m_1 \geq m_2\), and
2) \((m_k)\) has no safe guarantee if \((m_k)\) has no safe guarantee and \(k_1 \geq k_2\).

Therefore, we do not need to iterate over every \((m_k)\) constraint. Instead, we start with \(m = 1\) and \(k = 2\), increment \(k\) when \(\bar{d}(m,k) \leq d_i^{safe}\), and increment \(m\) when \(\bar{d}(m,k) > d_i^{safe}\). As a result, the total number of iterations is reduced from \(O(k_{max}^2)\) to \(O(k_{max})\).

2) Synthesizing safe schedules:: As introduced in Section II-B a weakly-hard constraint \((m_k)\) is a regular language \(L_{(m,k)}\) over \(\{0,1\}\), where a string represents a hit/miss pattern satisfying \((m_k)\). The set of safe constraints generated for controller \(T_i\) in the previous step can be represented as regular language as well, by taking the union of the regular languages representing each of the constraints. We use an automaton \(A_i = (\Sigma^*, T_i, \ell_i^0, \ell_i^1)\) to represent the weakly-hard constraints for the control task \(T_i\), where \(\Sigma^*\) is a set of locations (states), \(\Sigma = \{0,1\}\) is the input alphabet, \(T_i^*\) is the transition function, \(\ell_i^0\) is the set of accepting locations, and \(\ell_i^1\) is the initial location.

With this construction, an accepting run of \(A_i\) is a hit/miss pattern that satisfies at least one safe weakly-hard constraint for the corresponding controller task \(T_i\). Corresponding to a task set with \(N\) control tasks, we will use the set of controller automata \(\{A_i \mid i \in 1, \ldots, N\}\) to construct a scheduler automaton, and use it to synthesize a schedule (of the form shown in Figure 1) where \(N = 4\). We consider a time horizon of \(H\) time slots and assume that \(J (\ll N)\) controller jobs can fit into one time slot. Thus, we shall construct the scheduler automaton as a product of the \(N\) controller automata, where accepting runs of length \(H + 1\) of the scheduler automaton will constitute the set of safe schedules that we seek.

a) Example:: Consider the following example, where two controller automata \(\{A_1, A_2\}\) representing a set of weakly-hard constraints on control tasks \(T_1\) and \(T_2\) are given, for which we wish to synthesize a schedule such that only one task can be scheduled in each time slot. Assume that \(011011\) is an accepting run of \(A_1\), representing a pattern of deadline hit/miss that does not violate the safety constraint of \(T_1\), up to a time horizon of 6. Similarly, assume \(100100\) is an accepting run of \(A_2\). Clearly, given the two accepting runs, a possible schedule can be given as vectors \((1,0,0,1,0,0)\) and \((0,1,1,1,0,1)\), where each vector denotes the set of tasks that should be scheduled at each time. Control task \(T_1\) should be scheduled in the given time step if and only if there is a 1 in the first
position of the vector. Similarly, $T_2$ should be scheduled if and only if there is a 1 in the second position of the vector—for instance, $\binom{1}{0}$ denotes that controller task $T_2$ should be scheduled at that time step. We next show how to derive such schedules from a set $\{\alpha_i\}$.

**Definition 1:** A scheduler automaton $A^S$ for a set of $N$ control tasks whose constraints are represented by the automata of the form $A_i = \langle L_i, \Sigma_i, T^i, \ell^i_0 \rangle$, where at most $J$ controllers can be scheduled in each time slot, is defined as an automaton $\langle L^S, \Sigma^S, T^S, L^S_0, \ell^S_0 \rangle$:

$L^S$ set of locations, $L^S = \prod_i L_i$;
$\Sigma^S$ input alphabet, $\Sigma^S \subset \{0,1\}^N$. A sequence $\sigma \in \{0,1\}^N$ is in $\Sigma^S$ if and only if $\sum_i \sigma^i \leq J$;
$T^S$ transition function, $T^S(\ell, \sigma) = \prod_i T^i(\ell^i, \sigma^i)$;
$L^S_0$ accepting locations of the automaton, $L^S_0 = \prod_i L^i_0$;
$\ell^S_0$ initial location of the automaton, $\ell^S_0 = \prod_i \ell^i_0$.

The new set of locations $L^S$ of the scheduler automaton is obtained by taking a Cartesian product of all the controller automata locations $L_i$. Similarly, the initial location and the accepting locations are Cartesian products of the individual controller automata’s initial locations and accepting locations, respectively. The set of actions $\Sigma^S \subset \{0,1\}^N$ represents the legal actions to take at each time slot, i.e., an action $\sigma \in \Sigma^S$ is valid if and only if $\sum_i \sigma^i \leq J$. The transition function of the scheduler automaton is a modified Cartesian product. Intuitively, it is emulating all the (safe) weakly-hard constraints of all controllers together. Therefore, all the transitions that lead to an accepting state are in the set of valid schedules.

With this construction of the scheduler automaton, an accepting run of the automaton represents a safe schedule of the control tasks. The existence of safe schedules can be checked by running emptiness checking on the scheduler automaton, and schedules can be generated using breadth-first search (BFS).

For simplicity of exposition, we have presented the main steps of our construction explicitly. However, this will not be the most efficient implementation and there is significant scope for optimization using standard formal verification tools. [18]. For instance, one might just work with a set of weakly-hard constraints for each controller and begin to explore the state space of the scheduler automaton on the fly while inferring the state spaces of the controller automata (capturing the associated weakly-hard constraints) as needed. We will stop as soon as an accepting run of the scheduler automaton is found. Thus the full price of the construction will be incurred only when there is no feasible schedule.

**V. ILLUSTRATION OF THE PROPOSED APPROACH**

In this section, we illustrate the methods proposed in Section [IV] using a simple task set with two controller tasks (shown in Table I). Note that the utilization of this example test set is $U \approx 1.3 > 1$ and is not schedulable with conventional deadline-based methods. We will first find the suitable common periods for scheduling, and then verify them with schedule synthesis. We refer to Algorithm [I] as a structured guide for this section:

The first step is to find suitable common periods so that the methods of schedule synthesis can be applied. Sorting the tasks by their WCET, we obtain the common periods $P^C_1 = 15$ ms, and $P^C_2 = 15 + 10 = 25$ ms. These are the slot sizes that fit one and two tasks, respectively.

For each possible common period, we attempt to synthesize a safe schedule of the two systems. The first common period we use for schedule synthesis is $P^C = 15$ ms.

Assume that the dynamic of system 1 is described by the following state-space model:

$$x_1(t) = \begin{bmatrix} 5 & -2 \\ 0.7 & -1 \end{bmatrix} x(t) + \begin{bmatrix} 2 \\ 0.2 \end{bmatrix} u(t).$$

The system is initially discretized for its original period $P = 20$ ms. To apply the schedule synthesis methods, we first re-discretize it using the new common period $P^C = 15$ ms and obtain the following discrete state-space model:

$$x_1(t) = \begin{bmatrix} 1.0777 & -0.0309 \\ 0.0108 & 0.9850 \end{bmatrix} x(t) + \begin{bmatrix} 0.0311 \\ 0.0031 \end{bmatrix} u(t).$$

We apply the same process to Task 2 and obtain the discretized state-space model for it as well.

Assume the time horizon $H = 20$ and maximum window size $k_{max} = 4$. Running scheduleSynthesis returns the safe constraints shown in Table II where a “✓” in position $(m, k)$ means that the weakly-hard constraint $\binom{m}{k}$ is safe for that system, and an “×” indicates that the constraint is not known to be safe. Additionally, a safe schedule is synthesized, shown in Fig. 4. Intuitively, the scheduling strategy is to alternate between scheduling Task 1 and Task 2 in the given 15 ms slots, and it is easy to check that the deadline miss pattern of either task satisfies the $\binom{1}{1}$ constraint, a safe constraint for both systems.

<table>
<thead>
<tr>
<th>System</th>
<th>WCET ($C_i$)</th>
<th>Period ($P_i$)</th>
<th>Safety Margin ($d^{safe}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>10 ms</td>
<td>18 ms</td>
<td>2.5</td>
</tr>
<tr>
<td>Task 2</td>
<td>15 ms</td>
<td>20 ms</td>
<td>1.2</td>
</tr>
</tbody>
</table>

**TABLE I**

**TABLE II**

**SAFE CONSTRAINTS FOR THE EXAMPLE SYSTEMS IN TABLE I WITH COMMON PERIOD $P^C = 15$ ms.**
In this simple example, our algorithm is able to find a safe schedule without recomputing the gain values and exits here. If this was not the case, the algorithm will proceed to the second half and repeat the above process, but with recomputed gains for each potential $P^C_i$. The algorithm will return None if it reaches the end of the second loop without finding any safe schedule.

Algorithm 1: Illustration of the proposed approach.

```plaintext
input: A set of controller tasks $T$ with parameters 
$\{C_i, P_i, d_i^\text{safe}\}$ (WCET, period, safety margin)
output: A safe schedule with its period $P^C_i$; or None
1 // Try scheduling without recomputing controller gains
2 for $P^C_k \in \mathbb{P}$ do
3 $T_d \leftarrow \text{discretize}(T, P^C_k)$;
4 $s \leftarrow \text{scheduleSynthesis}(T_d)$;
5 if $s$ is not None then
6 return $s$
7 // Try scheduling with recomputing controller gains
8 for $P^C_k \in \mathbb{P}$ do
9 $T'_d \leftarrow \text{recomputeGains}(T, P^C_k)$;
10 $T'_d \leftarrow \text{discretize}(T'_d, P^C_k)$;
11 $s \leftarrow \text{scheduleSynthesis}(T'_d, P^C_k)$;
12 if $s$ is not None then
13 return $s$
15 return None
```

VI. EXPERIMENTAL RESULTS

We implemented our techniques using Julia and evaluated them on five standard controllers from the automotive domain. In Section VI-A we introduce these five controllers used in a case study along with their parameters (such as WCETs and periods), and the required safety margins that they must satisfy. In Section VI-B we give an overview of our approach and a summary of our findings. Section VI-C describes the details of the schedule synthesis process, and finally in Section VI-D we discuss the insights gained from our results.

A. Plant Models

Each of our five controllers was designed for a certain sampling period and WCET, and must be within the given safety margins—Table III provides these parameters.

1) RC Network (RC): Our first model is a resistor-capacitor network [19] with the following model:

$$\dot{x}(t) = \begin{bmatrix} -6.0 & 1.0 \\ 0.2 & -0.7 \end{bmatrix} x(t) + \begin{bmatrix} 5.0 \\ 0.5 \end{bmatrix} u(t).$$

2) FI10th Car (FI): Our second model is the linearized motion of an FI10th model car [20].

$$\dot{x}(t) = \begin{bmatrix} 0 & 6.5 \\ 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 19.685 \end{bmatrix} u(t).$$

Our next three plant models are selected from [21] and also represent subsystems from the automotive domain.

3) DC Motor (DC): Our third model is the speed control for DC motor adapted from [22].

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

4) Car Suspension (CS): Our fourth model is a suspension system adapted from [23].

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 & 0 \\ -8 & -4 & 8 \\ 0 & 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} 80 \\ 20 \\ -1120 \end{bmatrix} u(t).$$

5) Cruise Control (CC): Our final model is a cruise control system adapted from [24].

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 & 0 \\ -6.0476 & -5.2856 & -0.238 \\ 0 & 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 0 \\ 2.4767 \end{bmatrix} u(t).$$

B. Experiment Overview

As Table III shows, the control tasks have different periods and their utilization $U(\approx 2.51)$ is far greater than 1. That is, these tasks are not schedulable on a single processor if no deadline misses are allowed. In this section, we attempt to derive a common period for the controllers and demonstrate that with our proposed schedule synthesis technique they can be scheduled on a single processor while satisfying their safety margins (while missing certain deadlines).

With the WCET values shown in Table III we derive a number of suitable common periods and convert all the control tasks to one of the common periods so that our schedule synthesis technique may be applied. The common periods

<table>
<thead>
<tr>
<th>System</th>
<th>WCET</th>
<th>Period</th>
<th>Safety Margin</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>10 ms</td>
<td>23 ms</td>
<td>1.4</td>
</tr>
<tr>
<td>FI</td>
<td>13 ms</td>
<td>20 ms</td>
<td>1.2</td>
</tr>
<tr>
<td>DC</td>
<td>12 ms</td>
<td>23 ms</td>
<td>3.5</td>
</tr>
<tr>
<td>CS</td>
<td>10 ms</td>
<td>27 ms</td>
<td>9.4</td>
</tr>
<tr>
<td>CC</td>
<td>15 ms</td>
<td>28 ms</td>
<td>0.53</td>
</tr>
</tbody>
</table>

TABLE III

PARAMETERS FOR THE CONTROLLERS DEFINED IN SECTION VI-A
we explore are: $P^C_1 = 15\text{ ms}$ (where any one controller task can fit inside one scheduling slot), $P^C_2 = 15 + 13 = 28\text{ ms}$ (any two tasks fit), and $P^C_3 = 15 + 13 + 12 = 40\text{ ms}$ (any three tasks fit). We are able to derive a schedule for the five controllers, where each controller satisfies its corresponding safety constraints despite missing some deadlines and some of them running with a longer period than what they were designed for.

As shown in Fig. 5, one example of a valid schedule is as follows: From $t = 1$ to $t = 16$, the tasks to be scheduled in each slot (Step($t$)) are shown in the table. From $t = 17$ to $t = 100$, the schedule is repeated from $t = 5$ to $t = 16$, viz., the schedule at time $t \in [17, 100]$ can be given as $s[t] = (17 + (t - 17) \mod 12) + 5$. We note that this is the only valid schedule for this particular set of control tasks with the common period $P^C_2 = 28\text{ ms}$. The accepting runs of the scheduler automaton represent the set of all the valid schedules. Furthermore, there might be other common periods that can be used to schedule these tasks, but with $P^C_2 = 28\text{ ms}$, we are able to satisfy the safety requirements of all the control tasks.

D. Insights from the results obtained

To understand the effects of period compression and boosting and of deadline misses on system safety, we show the closed-loop dynamics of two specific controllers: the F1tenth Car and the Cruise Control system (see Fig. 6). The black lines denote their nominal trajectories, while the colored lines, except the red ones, represent their evolution under different safe common periods and schedules. There is no feasible schedule for $P^C_3 = 40\text{ ms}$ without recomputing the gain values of all the controllers. To illustrate this, we first obtain a feasible task schedule for $P^C_3 = 40\text{ ms}$ with recomputed gain values for all controllers. When this same schedule is applied to the controllers without recomputing their gain values, we obtain the red trajectories showing safety violations (viz., the trajectories of F1 and CC go outside their safety pipes).

<table>
<thead>
<tr>
<th>Period ($P^C_i$)</th>
<th>Original gain</th>
<th>Recomputed gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 ms (1 task)</td>
<td>Not schedulable</td>
<td>Not schedulable</td>
</tr>
<tr>
<td>28 ms (2 tasks)</td>
<td>Schedulable</td>
<td>Schedulable</td>
</tr>
<tr>
<td>40 ms (3 tasks)</td>
<td>Not schedulable</td>
<td>Schedulable</td>
</tr>
</tbody>
</table>

TABLE IV
Summary of Experimental Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Window Size ($k$)</th>
<th>Minimum Hits ($m$)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>2</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>F1</td>
<td>2</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DC</td>
<td>2</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CS</td>
<td>2</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CC</td>
<td>2</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

TABLE V
Synthesized constraints for $P^C_2 = 28\text{ ms}$, no redesign

C. Schedule Synthesis

For each common period, we first discretize all plant models using the selected common period. Using either the original or recomputed gain values, we apply our schedule synthesis technique to determine a safe schedule. We demonstrate this process with the common period $P^C_2 = 28\text{ ms}$. In this example, we use the original gain values and do not recompute them using this common period. The results of this schedule synthesis are presented below.

a) Safe constraints:: For each controller, we discretize the state-space model using the common period $P^C_2 = 28\text{ ms}$ and use our constraint synthesis technique outlined in Section [IV-B]. The time horizon $H$ is set to 100, and the maximum window size $k_{max}$ is set to 6. The results are shown in Table IV where a “✓” in position ($m, k$) means that the weakly-hard constraint ($m \leq k$) is safe for that system. Viz., a hit/miss pattern satisfying the weakly-hard constraint ($m \leq k$) would guarantee that the evolution of the plant stays within its safety margin. An “×” mark indicates that the constraint is not known to be safe.

b) Synthesize safe schedules:: After we select the common period and compute the list of safe weakly-hard constraints for each controller, we attempt to schedule all the controller tasks using the automata-based approach outlined in Section [IV-B]. With the common period $P^C_2 = 28\text{ ms}$, two controller tasks can fit within one period/slot. We are able to derive a schedule for the five controllers, where each controller satisfies their corresponding safety constraints despite missing some deadlines and some of them running with a longer period than what they were designed for.
Fig. 5. Synthesized schedule for the five controllers outlined in Section VI-A.

Fig. 6. Dynamics of the F1 tenth Car (left) and Cruise Control (right) models

As shown in the 1st subplot of Fig. 6, a system can diverge in some cases with a period change. For the F1 tenth Car, whose controller was originally designed for a period of 20 ms, when its period is changed to $P_{C_1} = 40$ ms and its gains are not recomputed, it diverges. This is clearly a safety violation. After recomputing the gain values for 40 ms, however, the system becomes safe under weakly-hard constraint ($\frac{5}{6}$). Another type of safety violation is highlighted in the 2nd subplot of Fig. 6. When CC’s sampling period is changed to 40 ms without recomputing its gains, it still remains convergent. However, its trajectory (in red) nevertheless goes outside its safety pipe and is thus deemed unsafe. For common period $P_{C_1} = 15$ ms, all systems have safe weakly-hard constraints that satisfy their safety property. But since only one out of the five tasks can fit in a time slot, the task set ends up not being schedulable, despite the fact that in theory, a shorter period enhances control performance. This highlights the non-intuitive nature of period compression and boosting: what is beneficial to systems on their own may not yield desirable results for the whole task set, and only a joint exploration of common periods and schedules provides the complete picture.

VII. CONCLUDING REMARKS

In this paper, we have studied the problem of packing multiple control tasks on a shared processor. The novelty of our approach lies in shifting the focus from ensuring all the deadlines are being met to satisfying system-level safety properties despite some deadline misses. This significantly enlarges the space of schedules that can be explored. To achieve this, we first address the issue of deriving a common sampling period for a set of control tasks that may be originally designed with distinct periods. This turns out to be a delicate problem involving counter-intuitive tradeoffs between the quality of control, the safety properties, and schedulability. We then use the language of weakly-hard constraints to capture the patterns of deadline misses that may be suffered by the control tasks and use them as the bridge between the system-level safety properties and the valid schedules. More precisely, given a safety property of a control system, we extract a set of weakly-hard constraints such that the system is guaranteed to be safe so long as these weakly-hard constraints are maintained. Then we construct a schedule for the task set such that deadline misses suffered by each control task satisfy its associated set of constraints.
Our experimental results show that considerably less pessimistic implementations can be obtained—and more schedules become feasible on a shared resource—when the focus is shifted from the usual goal of meeting all deadlines to meeting the safety properties of the systems under control. Furthermore, our approach is in line with the evolving paradigms in the automotive domain such as AUTOSAR Adaptive.

In our future work, we shall further explore the facets of deriving the common period, such as using periods that allow multiple levels of criticality to be specified, and the scheduling algorithm will take it into account when making scheduling decisions.

REFERENCES