DECNTR: Optimizing Safety and Schedulability with Multi-Mode Control and Resource Allocation Co-Design

Robert Gifford[†]   Felipe Galarza-Jimenez[‡]   Linh Thi Xuan Phan[†⋆]   Majid Zamani[‡]

[†]*University of Pennsylvania*   [‡]*University of Colorado, Boulder*   [⋆]*Roblox*

Abstract—As cyber-physical systems (CPS) become increasingly autonomous, there is a growing need for resource-efficient design techniques that can guarantee safety and timeliness during system reconfiguration or mode changes. In this paper, we present DECNTR, a co-design technique for jointly optimizing safety, schedulability and robustness for multi-mode CPS on multi-core platforms. By designing switching controllers that can switch between different implementations and between different sampling periods, DECNTR gives the resource allocation significantly more flexibility to adapt scheduling decisions to load changes, such as additional tasks in a new mode or increased demands during a mode change. For example, it can pick the best implementation for a task depending on the current resource availability; it can adapt the period within a safe range to effectively utilize CPU and shared resources, which helps increase performance and robustness; and it can relax some job deadlines to avoid transient overloads during mode transitions for better schedulability. Our evaluation on an automotive case study and resource-intensive benchmarks shows that DECNTR is highly effective in maximizing schedulability and robustness while ensuring safety, and that it significantly outperforms the state of the art in multi-core resource allocation for multi-mode systems.

Index Terms—Control and platform co-design; Safety; Multi-mode systems; Multi-core resource allocation.

I. Introduction

Modern cyber-physical systems (CPS), such as self-driving vehicles and autonomous marine systems, are inherently adaptive. A self-driving car, for instance, needs to execute different control tasks depending on detected obstacles, road conditions, or potential hardware/software faults. A standard way to model and analyze such systems is to use *multi-mode formalism*, where each mode represents a system configuration (composed of tasks that are active in the mode), and each transition represents a switch from one configuration to another in response to a mode-change event (such as a detected obstacle).

Guaranteeing the safety, robustness, and timeliness of multi-mode CPS on multi-core platforms is highly challenging. On the one hand, the control design must ensure that the output (or state) of the plant controlled by each task is inside a safe region; on the other hand, the scheduling and resource allocation of the control tasks must ensure schedulability in each mode and during each mode transition. Achieving these two (contradictory) goals concurrently is difficult, especially when the set of control tasks – and thus the system load – changes as the system moves from one mode to another.

One reason for this difficulty is that, to enable implementation on a platform, the controller (implemented as a control task) is designed for a particular sampling period (which is used as the task period). The controller guarantees the safety constraint only at the sampling instances, whereas the
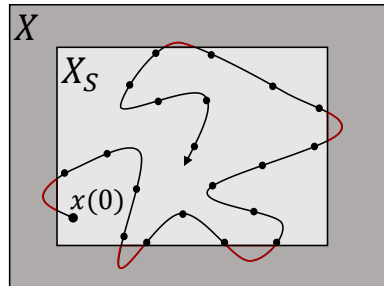

Fig. 1: The plant is connected to a feedback controller that guarantees the invariance of X_S at sampling time (denoted by dots on the curve) but not during inter-sampling intervals.

behavior at inter-sampling intervals might be unsafe due to plant dynamics or external disturbance. Fig. 1 illustrates this situation: the inner rectangle denotes the safe region X_S, the shaded rectangle denotes all possible plant states X, the curve represents the state during a simulation time interval, and the dots on the curve represent the state at sampling times. As highlighted in this figure, while the state at sampling times is always inside the safe region X_S, certain parts of the curve (highlighted in red) are outside X_S. Therefore, a goal of our work is also to increase the *robustness* of the controlled system – namely, the system's ability to remain safe even in the presence of slight disturbance affecting the state measurements at the sampling instant. In other words, the more robust a controller is, the more likely that the state is also inside X_S during inter-sampling intervals.

One way to increase robustness is to reduce the size of X_S or to impose a bound on the possible external disturbances, but this also reduces the controller's capability. For this reason, reducing the sampling period to reduce the uncertainty about the plant's inter-sample behaviors is a common choice (see [49] and references therein). However, a smaller period also makes a task harder to schedule. During a mode transition, this problem is exacerbated because one needs to schedule not only jobs released in the new mode but also unfinished jobs from the old mode. Unfinished jobs of tasks that continue in the new mode are hard to handle. In the worst case, they need to be executed immediately to meet their deadlines, at the cost of delaying the execution of new tasks, which can lead to deadline misses.

Multi-mode systems have been studied extensively in the real-time community, but the focus is mostly on schedulability analysis (particularly during mode transitions), and existing work generally considers a single core. Recently, Omni [21] presented an end-to-end task mapping and resource allocation solution for multi-mode systems on multi-core platforms. By dynamically adjusting the mapping of tasks to cores and the allocation of shared resources (such as cache and memory

bandwidth) to cores at mode transitions, it can improve schedulability substantially. However, Omni *completely ignores control aspects* and assumes instead that the sampling period for each task in each mode is given a priori. Thus, if the task periods need to be small to avoid safety violations during inter-sampling times (to enhance robustness). it may not be able to schedule the tasks. In addition, to minimize the number of re-allocations during transitions, Omni works by finding an allocation for each mode that is as similar to one another as possible. Consequently, its allocation may not be optimal for any individual mode.

In this paper, we present a co-design approach to achieving safety, robustness, and timeliness for multi-mode CPS on multi-core platforms. Our insights are twofold: First, given a control task, it is possible to design safety conditions for switching between different controllers (implementations), which may have different sampling periods. Having multiple implementations under different sampling periods enables the resource allocation to determine the 'best' period for each task in each mode to maximize schedulability and robustness, while still guaranteeing safety. Second, it is possible to 'delay' the deadline of an unfinished job (or even to kill them) without compromising safety, as long as the extended deadline is within one of the safe periods of the controller used. In other words, we can relax the *periodicity* requirement for continuing tasks, often imposed in existing work (including Omni). This relaxation helps reduce the load during mode transitions, which not only increases schedulability but also enables us to use allocations that are ideal for each mode individually, thus further improving resource efficiency.

To realize this approach, we introduce DECNTR, a concrete co-design method for co-optimizing safety, robustness, and schedulability. At the core, DECNTR consists of a control design technique for developing safe controllers that can switch between different implementations (sampling periods) and that can accommodate a certain delay of unfinished jobs during a switch. DECNTR resource allocation algorithm intelligently exploits these properties to determine, for each mode, a sampling period for each task, a mapping of tasks to cores, and an allocation of cache and memory bandwidth to each core so as to maximize schedulability while minimizing periods (to increase robustness). This is done in tandem with delaying unfinished jobs, if necessary, to improve schedulability during mode transitions. By co-designing the controller and resource allocation this way, DECNTR substantially increases schedulability and robustness compared to the state of the art, while guaranteeing safety. As a side benefit, DECNTR resource allocation can be used for optimizing real-time performance by adapting the best periods, or allowing graceful degradation of service as well.

In summary, the paper makes the following contributions:
- A co-design approach for jointly developing controllers and resource allocation algorithms to maximize safety, robustness and schedulability.
- A set of controllers that guarantees the CPS's safety under known sampling-time variations and transitions between

two different control implementations (Section IV).
- A novel multi-core task and resource allocation algorithm that guarantees safety while maximizing robustness and resource efficiency (Section V).

Our evaluation using a CPS case study and real-time benchmarks shows that DECNTR can substantially improve robustness while increasing schedulability by up to $11\times$ compared to the state of the art.

II. RELATED WORK

There is a large body of work on multi-mode systems. Prior work in this area often focuses on one of two key areas: 1) new models and timing analysis techniques (see e.g., [12], [13], [22], [3], [34], [45], [35], [36], [29]), and 2) mode-change protocols for ensuring schedulability during mode transitions (e.g., see [10], [39] and references therein). Recently, multi-mode scheduling and analysis have been extended towards multiprocessors [23], [32], [17], [5], [38], [1]. The majority only considers CPU, but some recent work, like Omni [21], considers shared resources [31], [28].

Several multi-resource and task co-allocation techniques have been developed. For example [54], [53] propose holistic resource allocation techniques that find the assignments of tasks, cache and memory bandwidth to cores. DNA/-DADNA [20] does the same dynamically at run time for soft real-time tasks but also does not consider multiple modes. These techniques, however, focus on single-mode systems.

Safety controllers are extensively studied for CPS applications. One particular approach is the design of so-called barrier functions [37], [2], [27]. This method is applicable to systems with nonlinear dynamics and circumvents the direct computation of the *controlled invariant set*, which delineates the region within which the system can maintain safety. Nonetheless, a level-set of barrier functions, which constitutes the controlled invariant set, can be overly conservative due to the inherent conservatism of barrier functions. Other research concentrates on systems with linear dynamics [6], [8], [9], which can compute the maximal controlled invariant set via operations over sets.

Lastly, there is extensive research in control and scheduling co-design. The result in [4] is one of the first to introduce the concept of modifying a control task's sampling period for the sake of improving CPU efficiency. Since then, there has been a large collection of techniques [19], [44], [40], [42], [11], [58], [46], [15], [43], [24], [16], [47], [48] that apply co-design principles to resource allocation, to scheduling, or to multi-mode systems. However, to the best of our knowledge, no prior work has considered a holistic co-design of multi-mode controllers with multi-mode allocations of tasks and shared resources (such as cache and memory bandwidth) on multi-core platforms.

III. SYSTEM MODELING AND MODE-CHANGE PROTOCOL

A. *Multi-mode system and platform modeling*

Platform. The system is deployed on a multi-core platform consisting of r identical cores that can access a shared

cache and shared memory bandwidth (BW). The cache and BW are divided into $C_{\max}$ and $W_{\max}$ equal-size partitions, respectively. At run time, distinct sets of cache and BW partitions are distributed to cores, and each core has exclusive access to its allocated partitions. Cache and BW allocation can be performed using existing mechanisms, such as Intel's CAT [26] for cache and MemGuard [55] for BW.

System model. We consider a multi-mode system defined by $\{\mathcal{M}, \mathcal{R}, m_0, \mathcal{T}\}$, where $\mathcal{M}$ is the set of modes, $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ is the set of transitions, $m_0 \in \mathcal{M}$ is the initial mode, and $\mathcal{T}$ is the set of control tasks. Each mode m is associated with a set of periodic tasks $\mathcal{T}^m \subseteq \mathcal{T}$ that are active in m. Each transition (m', m) is triggered by a mode-change request event (MCR), upon which the system moves from mode m' to mode m by executing a mode-change protocol (described below). We assume that when an MCR occurs, another MCR can only occur after the system has completely moved to the new mode. Within a mode, each active task is mapped onto a core, and it remains on the core while the system is in this mode. Tasks on the same core are scheduled under the Earliest Deadline First (EDF) policy.

Task model. Each task τ_i in the system controls a particular physical plant of the CPS. It is associated with a set of q_i controllers (implementations) $\mathcal{K}_i = \{\mathcal{K}_{i,j} \mid 1 \leq j \leq q_i\}$, from which it can pick to execute in a mode. Each controller $\mathcal{K}_{i,j}$ has a safe range of sampling periods, $[\rho_{i,j}, \varrho_{i,j}]$, and the task can switch between any two sampling periods in this range that are multiples of ρ without compromising safety, where $\rho > 0$ is a configurable control parameter. We denote by $\rho_{i,\min} = \min_{j=1}^{q_i} \rho_{i,j}$ and $\varrho_{i,\max} = \max_{j=1}^{q_i} \varrho_{i,j}$ the minimum and maximum safe period of τ_i, respectively. In each mode m where τ_i is active, our resource allocation algorithm will assign a specific controller $\mathcal{K}_i^m \in \mathcal{K}_i$ and a period value p_i^m in the period range of $\mathcal{K}_i^m$ for execution. For simplicity, we assume the task's deadline is the same as its assigned period; it should be straightforward to extend our algorithm to the case where deadlines are smaller than periods.

We consider a resource-aware task model where a task's worst-case execution time (WCET) depends on the resource it is given. Specifically, $e_i(c, w)$ represents the WCET of τ_i when it is given c cache partitions and w BW partitions, for all $1 \leq c \leq C_{max}$ and $1 \leq w \leq W_{max}$.

Mode-change protocol design. To define the execution behavior during a mode transition from m' to m, we distinguish three types of tasks associated with the transition: (i) *old* tasks are active in m' but not in m; (ii) *new* tasks are active in m but not in m'; and (iii) *carry-over* tasks are active in both m and m'. If a carry-over task has an unfinished job, we call the job a carry-over job during the mode transition.

We drop old tasks (and their unfinished jobs) at the MCR instant. However, the mode-change semantics for carry-over and new tasks are designed to leverage the capability of our controllers in tolerating certain delay for the next actuation (sampling) time during a mode transition and in allowing more than one sampling period.

Formally, for each transition (m', m), we determine for each task τ_i in mode m whether we need to delay the deadline (within a safe range) of its first job to complete after the MCR (to improve schedulability). This could either be its carry-over job (if exists) or the first job that τ_i releases in mode m. If we do, we mark the task as *delayable* by setting $\text{del}_i(m', m) = 1$, and compute a corresponding 'delayed' deadline. If τ_i has a carry-over job, then the delayed deadline must be a safe period value for $\mathcal{K}_i^{m'}$ (the controller used in the old mode); otherwise, the delayed deadline must be a safe period value of $\mathcal{K}_i^m$. Hence, we will associate with each task τ_i in each transition $(m', m) \in \mathcal{R}$ a delayed deadline $d_i(m', m)$ that may be applied to the first job τ_i releases in mode m, for all $\tau_i \in \mathcal{T}^m$. In addition, we also compute a delayed deadline $d_i^c(m', m)$ for the carry-over job of each carry-over task τ_i of the transition.

Mode-change semantics. When a mode transition (m', m) is triggered, we execute the following mode-change actions:

Step 0) Drop all old tasks, including their unfinished jobs.

Step 1) For each new task, release its first job immediately.

Step 2) For each carry-over task that has no carry-over job, release the next job at p_i^m time units after its last job release.

Step 3) For each task τ_i in m that is delayable, if it has a carry-over job, we set the deadline of the carry-over job to be $d_i^c(m', m)$; otherwise, we set the deadline of its first job in m to be $d_i(m', m)$. All subsequent jobs are released and assigned deadlines according to the task's assigned period p_i^m. For all tasks τ_i that are not delayable, all jobs will follow the periods determined by the mode in which they are released.

In the new mode m, the tasks and resources allocated to each core may change. We assume that any such change take effect immediately, including for carry-over jobs.

B. Controller design

As discussed earlier, each task in a mode implements a feedback controller that controls a physical plant. We now discuss our control model, starting with some necessary notation.

Given sets X and Y, the projection map of X onto Y is denoted by $\Pi_Y(X)$. As usual, $\mathbb{N}, \mathbb{Z}, \mathbb{Z}_{\geq 0}, \mathbb{R}, \mathbb{R}_{>0}$ and $\mathbb{R}_{\geq 0}$ denote the set of natural, integer, nonnegative integer, real, positive, and nonnegative real numbers, respectively. Notations $[a, b],]a, b[, [a, b[$ and $]a, b]$ denote closed, open, and half-open sets in $\mathbb{R}$. Likewise, $[a; b],]a; b[, [a; b[$ and $]a; b]$ denote closed, open, and half-open sets in $\mathbb{Z}$. Thus, $[a, b] \cap \mathbb{Z} = [a; b]$.

Given sets A and B, $f : A \rightrightarrows B$ denotes a *set-valued map*, whereas $f : A \to B$ denotes a *single-valued map* (i.e., a *function*). We denote the identity map by $\mathcal{I}$. The set of maps from Y to X is denoted by X^Y. The set of all signals with image on X defined on intervals $[0; T[$ is denoted by $X^{[0;T[}$. Finally, $X^\infty = \bigcup_{T \in \mathbb{Z}_{\geq 0} \cup \{\infty\}} X^{[0;T[}$.

We consider plants as linear control systems evolving in continuous-time, as defined below.

Definition 1 (ct-LTI). *A linear control system is described by:*

$$\dot{\xi}(t) = A_c \xi(t) + B_c \nu(t), \tag{1}$$

where $A_c \in \mathbb{R}^{n \times n}$, $B_c \in \mathbb{R}^{n \times d}$, $\xi(t) \in \bar{X} \subseteq \mathbb{R}^n$, and $\nu(t) \in \bar{U} \subseteq \mathbb{R}^d$ for all $t \in \mathbb{R}_{\geq 0}$. ξ and ν are called state and input trajectories of the system, respectively. Given $\rho \in \mathbb{R}_{>0}$ and interval $I \subseteq [0, \rho]$, a solution of (1) on I under input ν is defined as an absolutely continuous function $\xi : I \to \bar{X}$ whose time derivative $\dot{\xi}(t)$ satisfies (1) for almost every $t \in I$.

To enable implementation on a digital platform, we consider a sampled-and-hold version of the plant's model for the sake of design as introduced below.

Definition 2 (dt-LTI). *A discrete-time linear control system associated with the control system in* (1) *and sampling time $\rho > 0$ is a tuple*

$$\mathcal{S} = (X, X_0, U, A, B, \rho), \tag{2}$$

where $X = \bar{X}$ is the state set, $U = \bar{U}$ is the input set, $X_0 \subseteq X$ is a set of initial states, and the evolution of the system is described as:

$$x(t+1) = Ax(t) + Bu(t), \tag{3}$$

with $A = e^{A_c \rho}$ and $B = \int_0^\rho e^{A_c t} dt \cdot B_c$.

A tuple $(\mathbf{u}, \mathbf{x}) \in U^{[0;T[} \times X^{[0;T[}$ is a *solution* of the system in (2) over $[0;T[$ if for $T \in \mathbb{N} \cup \{\infty\}$, (3) holds $\forall t \in [0; T-1[$ and $\mathbf{x}(0) \in X_0$.

Here, for a given safety set $X_S \subseteq X$, we are interested in designing feedback controllers $\mathcal{K} : D \rightrightarrows U$, for some $D \subseteq X_S$, forcing solutions of (3) to evolve within $U^\infty \times D^\infty$, i.e. $x(t) \in D$ for all $t \in \mathbb{N} \cup \{\infty\}$, where $u(t) \in \mathcal{K}(x(t))$. In particular, we denote the set of such controllers by $\bar{\mathcal{K}}(U)$. Such a family of controllers is characterized by the maximal controlled invariant set contained in X [6], [8], [9], [51], formally defined as follows.

Definition 3. *Consider a safety set $X_S \subseteq X$. A set $R \subseteq X_S$ is called controlled invariant w.r.t.* (3) *and U, if there exists a feedback controller $\mathcal{K} \in \bar{\mathcal{K}}(U)$ so that every solution $(\mathbf{u}, \mathbf{x})$ of* (3) *with initial state $x(0) \in R$ and $\mathbf{u} \in \mathcal{K}(\mathbf{x})$, evolves in $U^\infty \times R^\infty$. Moreover, we denote the maximal controlled invariant set of* (3) *and U as $R(X_S)$.*

We use the following iteration for the computation of $R(X_S)$ as proposed in [6]:

$$R_0 = X_S, \quad R_{i+1} = \text{pre}(R_i) \cap X_S, \tag{4}$$

where $\text{pre}(R) = \{x \in X \mid \exists\, u \in U \text{ s.t. } Ax + Bu \in R\}$. If it exists, the fixed point of the iteration is the set $R(X_S)$.

C. Problem formulation

Our first goal is to design a set of safety controllers (see Fig. 2) for each plant. Specifically, given a plant as in Definition 1, for a given $\rho > 0$ and dt-LTI $\mathcal{S} = (X, X_0, U, A, B, \rho)$ associated to the plant, and a *safety* set $X_S \subseteq X$, we aim to develop a set of feedback controllers of the form $\mathcal{K} : R(X_S) \rightrightarrows U$ guaranteeing that for any $\mathbf{u} \in \mathcal{K}(\mathbf{x})$, $(\mathbf{u}, \mathbf{x}) \in U^\infty \times X_S^\infty$ (see Definition 3). Specifically, we will develop for each task τ_i a set of $q_i \in \mathbb{N}$ *safety controllers* $(\mathcal{K}_{i,j}, p_{i,j})$, where $\mathcal{K}_{i,j} :$
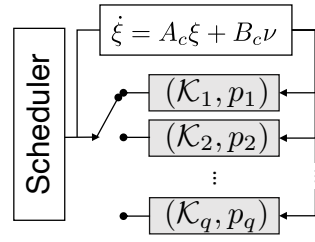

Fig. 2: The scheduler selects one of the controllers (implementation) $\mathcal{K}_j$ with period p_j. Task index i is omitted for clarity.

$R(X_{Si}) \rightrightarrows U_i$ and respective sampling period $p_{i,j}$ with $j \in \{1, \dots, q_i\}$, which guarantee the safety even under switching implementations (see Fig. 2).

Our second goal is to design a resource allocation algorithm that exploits the designed controllers to optimize schedulability and robustness. Specifically, for each mode $m \in \mathcal{M}$ and each task $\tau_i \in \mathcal{T}^m$, we need to determine:

1) c_k^m and w_k^m, the numbers of cache and BW partitions allocated to each core k ($0 \leq k < r$) in mode m;
2) core_i^m, a core assigned to τ_i in m; and
3) $\mathcal{K}_i^m$ and p_i^m, a controller and a period assigned to τ_i in m, where $\mathcal{K}_i^m \in \mathcal{K}_i$ and p_i^m is a period of $\mathcal{K}_i^m$.

In addition, for each incoming mode transition $(m', m) \in \mathcal{R}$, we need to determine

- $\text{del}_i(m', m)$, a boolean variable indicating whether we should delay a job of τ_i during the transition (m', m);
- $d_i^c(m', m)$, a delayed deadline for the carry-over job of τ_i, if τ_i is a carry-over task; and
- $d_i(m', m)$, a delayed deadline for the first job of τ_i released in mode m. At run time, if τ_i has a carry-over job, then we only delay the deadline of its carry-over job (using $d_i^c(m', m)$) but not its first new job in m.

Our objective is to maximize schedulability, while minimizing the assigned periods and delayed-deadlines of all tasks in all modes and mode transitions.

IV. CONTROLLER DESIGN

In this section, we focus on the controller design for just one plant; thus, we drop the task index i and only use τ for the sake of simple presentation. First, we discuss the design of an isolated safety feedback controller with a constant sampling period. We then present a switching condition for a task τ that guarantees safety while switching between different controllers with different sampling periods, which will be used by our resource allocation algorithm (Section V).

A. Safety feedback controller with constant sampling period

We consider a dt-LTI $\mathcal{S}$ for a given plant and the following regularity assumptions, which are for the *tractable computation* of the maximal controlled invariant set [8].

Assumption 4. *Assume the pair (A, B) is controllable, i.e., the controllability matrix $\mathcal{C}(A, B) = [B\ AB\ A^2B\ \dots\ A^{n-1}B]$ is full rank.*

Assumption 5. *Sets X_S and U are polytopes. Namely, there are matrices H_x, H_u and vectors h_x, h_u such that $X_S = \{x \in \mathbb{R}^n \mid H_x x \leq h_x\}$ and $U := \{u \in \mathbb{R}^d \mid H_u u \leq h_u\}$ are compact sets.*

Moreover, we enforce the following condition on the sampling period of the controllers we aim to design for a given task τ.

Assumption 6. *The sampling period of every controller is a natural multiple of ρ, i.e., for each $j \in \{1, \ldots, q\}$, there is $\alpha_j \in \mathbb{N}$ such that $p_j = \rho_{\alpha_j} = \alpha_j \rho$.*

Assumption 6 does not restrict practical applications since the sampling period of the controller is related to the sampling period of the sensor from which the controller receives the feedback signal.

Now, let us consider only one controller j, and let Assumptions 4-6 hold. Then, for $\rho_\alpha = \alpha \rho$, the value of $u(t)$ applied for α consecutive instants must ensure that if $x(t) \in X_S$, then $x(t + \alpha) \in X_S$. One can compute the evolution of the system in this interval as follows:

$$x(t + 1) = Ax(t) + Bu(t)$$
$$x(t + 2) = A^2 x(t) + (A + \mathcal{I})Bu(t)$$
$$\vdots \quad = \quad \vdots$$
$$x(t + \alpha) = A^\alpha x(t) + (A^{\alpha - 1} + A^{\alpha - 2} + \ldots A + \mathcal{I})Bu(t).$$

We abuse notation and denote the "α"-maximal controlled invariant set as $R^\alpha(X_S)$, and the set $\text{pre}^\alpha(R) = \{x \in \mathbb{R}^n \mid \exists\, u \in U \text{ s.t. } A^\alpha x + A_{\alpha-1}Bu \in R\}$, where $A_\gamma := \sum_{i=0}^{\gamma} A^i$, and $A_0 = \mathcal{I}$. In particular, the iteration in (4) is initialized as follows:

$$\text{pre}^\alpha(R_0) = \tag{5}$$
$$\Pi_{X_S} \left(\left\{ (x, u) \;\middle|\; \begin{bmatrix} H_x A^\alpha & H_x A_{\alpha-1}B \\ 0 & H_u \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \begin{bmatrix} h_x \\ h_u \end{bmatrix} \right\} \right),$$

where $R_0 = X_S$ and $R_{i+1} = \text{pre}^\alpha(R_i)$. Assuming the iteration admits a fixed point in a finite iteration, the fixed point is the polytope $R^\alpha(X_S) := \{x \in X \mid H_{R^\alpha}x \leq h_{R^\alpha}\}$ when it is not empty [9].

Remark 7. *The linearity of the dt-LTI and Assumption 5 provide an LMI in (5) which simplifies the computation of Π_{X_S}. However, these assumptions are insufficient to guarantee the finite convergence of the iteration. Modifications to the iteration can be made to allow the computation of outer- and inner-approximations of $R^\alpha(X)$, ensuring the finite-time convergence of the iteration (see [41] and references therein). These modifications are omitted for the sake of clarity.*

Note that $R^\alpha(X_S)$ is the domain of only one safety feedback controller with a *fixed* period $p = \rho_\alpha$.

B. Safety Feedback Controllers with Multiple Sampling Time

We now consider the condition that allows the platform scheduler to switch among multiple (q) safety feedback controllers, each of which has a set of sampling periods that guarantee safety. This has two implications: 1) The scheduler may change the period of a chosen controller, namely, a control task τ implements a controller $(\mathcal{K}_j, p_j)$, where $p_j \in [\rho_j, \varrho_j]$ with $\rho_j \leq \varrho_j < \infty$. However, in the previous section, we designed $(\mathcal{K}_j, \rho_j)$ assuming the control input $u(t)$ is applied

at time instants $\rho_j t$ with $t \in \mathbb{Z}_{\geq 0}$. Hence, for any $p_j \neq \rho_j$, we cannot claim safety satisfaction; 2) The scheduler may change the controller of a given task, and the transition must be feasible and ensure the safety property. This means that if the scheduler executes controller $(\mathcal{K}_{j'}, p_{j'})$ after controller $(\mathcal{K}_j, p_j)$, then it should hold that $x(t + \alpha_j) \in R^{\alpha_{j'}}(X_S)$. Otherwise, $\mathcal{K}_{j'}(x(t + \alpha_j))$ might be empty, and one cannot ensure the safety of the closed-loop system.

To address the aforementioned issues, we consider a maximum delay value between executions. Without loss of generality, we consider $(\mathcal{K}_j, \rho_j)$, and $(\mathcal{K}_{j'}, \rho_{j'})$ with $j, j' \in \{1, 2, \ldots, q\}$, and the maximum delay between two executions $\rho_{jj'}$ as in Fig. 3. We provide $\rho_{jj'}$ for every combination of j and j' even for $j = j'$; thus, $p_j \in [\rho_j, \varrho_j]$ with $\varrho_j = \rho_j + \max_{j' \in \{1, \ldots, q\}} \rho_{jj'}$.
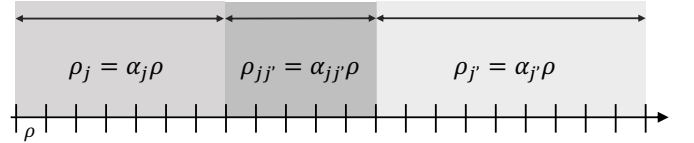
Fig. 3: Switching between two (different) controllers may be delayed in case the next implementation, $(\mathcal{K}_{j'}, \rho_{j'})$ in our case, cannot be allocated immediately after the final execution of $(\mathcal{K}_j, \rho_j)$.

Similar to Assumption 6, we assume the following condition for the maximum delay time.

Assumption 8. *For all j, j' with $j, j' \in \{1, \ldots, q\}$, the maximum delay is either zero or a natural multiple of ρ, namely, $\rho_{jj'} = \alpha_{jj'}\rho$ where $\alpha_{jj'} \in \mathbb{Z}_{\geq 0}$.*

The value of $\rho_{jj'}$ is a design parameter that allows $p_j \in [\alpha_j \rho; (\alpha_j + \alpha_{jj'})\rho]$, providing flexibility for the scheduler to delay a task in the event of an overloaded mode transition. For a given dt-LTI $\mathcal{S}$, a safe set X_S, and sampling times ρ_j, $\rho_{j'}$, with maximum delay $\rho_{jj'}$, we define the following inequality:

$$\begin{bmatrix} H_x A^{\alpha_j + \alpha_{jj'}} & H_x A_{\alpha_j + \alpha_{jj'}-1}B \\ H_x A^{\alpha_j + \alpha_{jj'}-1} & H_x A_{\alpha_j + \alpha_{jj'}-2}B \\ \vdots & \vdots \\ H_x A^{\alpha_j + 1} & H_x A_{\alpha_j}B \\ H_x A^{\alpha_j} & H_x A_{\alpha_j - 1}B \\ 0 & H_u \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \begin{bmatrix} h_x \\ h_x \\ \vdots \\ h_x \\ h_x \\ h_u \end{bmatrix}. \tag{6}$$

The design of each controller must ensure the safety property under any switching of controllers – that is, we need to provide a controlled invariant set $R^{\cap \alpha}$ such that for all $j, j' \in \{1, \ldots, q\}$, the inequality in (6) is satisfied. Under Assumptions 4-8, we modify iteration (4) as:

$$\text{pre}(R_0) = \Pi_{X_S} \left(\{(x, u) \in X_S \times U \mid \right. \tag{7}$$
$$\left. \forall j, j' \in \{1, \ldots, q\} \text{ inequality (6) holds} \} \right).$$

Assuming (7) converges in finite time to a not empty set, we have that the controlled invariant set is given by $R^{\cap \alpha} = \{x \in X_S \mid H_{R^{\cap \alpha}} x \leq h_{R^{\cap \alpha}}\}$.
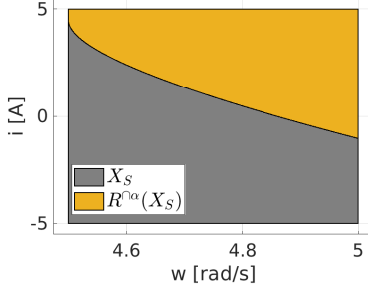
5

Fig. 4: Safe set and controlled invariant set for the DC Motor example.

Now, we provide the main result of this section.

Theorem 9. *Consider a ct-LTI as in Definition 1, its associated dt-LTI $\mathcal{S}$ with sampling period $\rho > 0$, and a safe set $X_S \subseteq X$. Let Assumptions 4-8 hold and $\bar{\alpha}_{jj'} := \max_{j'}\{\alpha_{jj'}\}$. The safety controllers $(\mathcal{K}_j, p_j)$ with $j \in \{1, \ldots, q\}$ have the following form:*

$$\mathcal{K}_j(x) = \{u \in U \mid \tag{8}$$

$$\begin{bmatrix} H_{R^{\cap \alpha}} A_{\alpha_j + \bar{\alpha}_{jj'} - 1} B \\ \vdots \\ H_{R^{\cap \alpha}} A_{\alpha_j} B \\ H_{R^{\cap \alpha}} A_{\alpha_j - 1} B \end{bmatrix} u \leq \begin{bmatrix} h_{R^{\cap \alpha}} - H_{R^{\cap \alpha}} A^{\alpha_j + \bar{\alpha}_{jj'}} x \\ \vdots \\ h_{R^{\cap \alpha}} - H_{R^{\cap \alpha}} A^{\alpha_j + 1} x \\ h_{R^{\cap \alpha}} - H_{R^{\cap \alpha}} A^{\alpha_j} x \\ h_{R^{\cap \alpha}} \end{bmatrix} \Big\}.$$

Note that the computation of $R^{\cap \alpha}(X_S)$ is done offline. Given a state measurement x, the controller consists of an algorithm that returns a subset of U fulfilling the Linear Matrix Inequality (LMI) in (8).

C. Example

For the sake of illustration, consider a DC motor [56], with states (w, i) representing, respectively, the rotational speed and electric current and DC voltage V as the input. The ct-LTI model is given by:

$$\begin{bmatrix} \dot{w} \\ \dot{i} \end{bmatrix} = \begin{bmatrix} -b/J & K/J \\ -K/L & -R/L \end{bmatrix} \begin{bmatrix} w \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 1/L \end{bmatrix} V,$$

with $J = 0.25$, $b = 0.05$, $K = 0.05$, $R = 0.1$, and $L = 0.1$. Consider the safe set $X = X_S = [4.5, 5] \times [-5, 5]$, input set $U = [-1, 1]$, and $\rho = 0.01s$. We consider three controllers with $p_1 = 12\rho$, $p_2 = 7\rho$, and $p_3 = 5\rho$. We implement the iteration in (4) with $\text{pre}^\alpha(R)$ as in (5), and obtain the results as in Fig. 4.

Next, we introduce an algorithm for task and resource allocation that leverages possible values of ρ and $p \in [\alpha_j \rho, (\alpha_j + \max_{j'}\{\alpha_{jj'}\})\rho] = [\rho_j, \varrho_j]$ to improve the resource efficiency, schedulability, and robustness of the system.

V. DECNTR RESOURCE ALLOCATION ALGORITHM

A. Key ideas and algorithm overview

Key ideas. The DECNTR algorithm allocates tasks and resources to cores in a way that optimizes each mode individually. By tightening task periods and delaying job deadlines during mode transitions if necessary, it achieves safety and schedulability while improving system robustness. We highlight three key insights of DECNTR that make this possible:

Insight #1: Group tasks with similar cache and BW requirements on the same cores to best consolidate resources to tasks that need them the most. If we instead place tasks with high resource requirements on many separate cores, there may not be sufficient resources for such cores to meet their demands.

Insight #2: Tightening each task's period to better improve system robustness. As discussed in Section IV, shorter task periods lead to measurable improvements in system robustness. Our allocation minimizes task periods whenever possible (i.e., as long as the system is still schedulable).

Insight #3: Delaying deadlines of some jobs within a safe interval during mode transitions to reduce transient overloads and improve transition schedulability. During a mode transition, the combination of both new jobs and carry-over jobs, with the latter often having very little slack time in the worst case, can lead to an overload in resource demands right after the MCR. We leverage the ability to safely delay controller actuation time to reduce this transient overload, making the transition easier to schedule.

Overview. Based on the above insights, DECNTR first computes an initial allocation for each mode. The initial mode allocation aims to (1) maximize resource efficiency, (2) minimize task periods (i.e., to improve controller robustness), and (3) guarantee each mode's schedulability. DECNTR achieves this by grouping tasks with similar resource needs onto the same cores. During this step, the algorithm assumes each task has the smallest possible period and increases it gradually – after having attempted to migrate tasks between cores of similar resource needs – until the mode becomes schedulable. The outcome of the initial allocation is a mapping of tasks and resources to cores, as well as a controller and a period assigned to each task, for each mode such that each mode is schedulable in isolation. If mode schedulability cannot be guaranteed, the system is deemed unschedulable and the algorithm terminates.

Following the initial allocation, DECNTR makes sure that each mode transition is schedulable. For this, it uses breath-first search to iterate over all mode transitions. For each transition (m', m), it checks whether the system is schedulable during the transition under the current mode allocation. If not, it adds a minimum delay – as small as possible within the safe duration allowed by the controller – to the deadline of the first job to be completed for a task on a core that is unschedulable, until either the transition becomes schedulable or no further delaying is possible. In the former case, it saves the delayed deadlines for this transition and moves to the next transition. In the latter case, it will attempt to increase the task periods in the new mode and re-check if the transition is schedulable. If so, it saves the assigned periods and moves to the next transition. If the periods have been increased up to their maximum sampling periods allowed by the controllers and the transition remains unschedulable, DECNTR reports system unschedulability and returns the current allocations. The algorithm outputs schedulability and the corresponding

Algorithm 1 Mode Allocation

```
 1: function MODEALLOC(m)                          ▷ m: mode
 2:     InitTasks(𝒯ᵐ)
 3:     SortByResourceSensitivity(𝒯ᵐ)
 4:     reset = true
 5:     while (true) do
 6:         if reset then
 7:             ResetCoreAlloc(cores)
 8:             cores[0].cache = C_max
 9:             cores[0].bw = W_max
10:             cores[0].tasks = 𝒯ᵐ
11:
12:         for i = 0; i < r and IsSchedulable(cores[i]); i++ do
13:             if i ≥ r then return SCHEDULABLE
14:             for j = i + 1; j < r and cores[j].util ≥ 1; j++ do;
15:             if j < r then
16:                 splitPoint = GetSplitPoint(cores[i].tasks)
17:                 SplitCore(cores[i], cores[j], splitPoint))
18:                 ResRedistrubte(cores)
19:                 reset = false
20:                 continue
21:         success = MIGRATE(m)
22:         if success then return SCHEDULABLE
23:
24:         if INFLATE(m) == false then break
25:         reset = true
26:     return UNSCHEDULABLE
```

Algorithm 2 Task Migration

```
 1: function MIGRATE(m)                             ▷ m: mode
 2:     while (true) do
 3:         for (i = 0; i < r and IsSchedulable(cores[i]); i++) do
 4:             if i ≥ r then return true
 5:             for (dist = 1; dist < r; dist++;) do
 6:                 if i − dist ≥ 0 and
 7:                     TaskResRedist(m, i, i − dist) then break
 8:                 if i + dist < r and
 9:                     IsSchedulable(cores[i + dist]) and
10:                     TaskResRedist(m, i, i + dist) then break
11:             if dist ≥ r then return false
```

allocation if all transitions are schedulable.

We next discuss the key procedures for allocation.

B. Mode allocation

Algorithm 1 shows the pseudo-code for finding an allocation for a mode m. DECNTR initializes each task τ_i in m by assigning its period to be its minimum safe period, i.e., $p_i^m = \rho_{i,\min}$. In addition, it sets $\mathcal{K}_i^m$ to be the controller $\mathcal{K}_{i,j}$ with the highest maximum period $\varrho_{i,j}$ where $\rho_{i,j} = p_i^m$. It then sorts all of m's tasks in non-increasing order of resource sensitivity (Line 3), where the resource sensitivity of a task τ_i in m is defined as:

$$\gamma_{i,m} = \frac{e_i(1,1) - e_i(C_{max}, W_{max})}{e_i(1,1)}. \tag{9}$$

Intuitively, the resource sensitivity of a task is a metric that depicts how much a task's WCET will be reduced when provided more resources. Here, we capture it simply as the relative difference in WCETs when provided minimum and maximum resources, since this already works well; however, a different estimation can also be used.

After sorting, DECNTR begins its first iteration where it resets all core allocations (Line 7), assigns all resources to core 0 (Lines 8-9), and assigns all the sorted tasks to core 0 (Line 10). DECNTR then begins checking for the schedulability of each core in increasing order of core index (Line 12), from 0 to $r - 1$ (where r is the number of cores), using the schedulability analysis in Section VI. If all cores are schedulable, DECNTR returns with the current allocation for m (Line 13), and it will move on to find an allocation for the next mode. If a core is found to be unschedulable, DECNTR will attempt to *split* the core's taskset (Lines 14-17).

To split, we first find the closest higher-index core j whose utilization is less than 1 (Line 14) and a *split point*, defined as the index of the task with the highest difference in resource sensitivity compared to the next task's on this core (tasks are sorted in non-increasing order of resource sensitivity). All tasks after the split point are moved onto core j, keeping their current ordering (Lines 16-17). This way, cores are automatically ordered in non-increasing resource sensitivity – a lower-index core contains tasks that benefit more from having additional resources than a higher-index core does.

After splitting, DECNTR redistributes resources onto all cores (Line 18). This is done by reassigning them one partition at a time, going from lower-utilization cores to higher-utilization cores to balance utilizations across cores. It then continues with rechecking schedulability for all cores (Line 20 then Line 12) and performs splitting of tasks on unschedulable cores as before. If we run out of a valid core to split onto, DECNTR attempts to migrate tasks between cores (Line 21).

Migration. Algorithm 2 shows the migration function, which performs a more fine-grained movement of tasks from unschedulable cores to some *nearby* schedulable cores to best leverage our resource sensitivity groupings. In each while-loop iteration, it first finds the first core i that is unschedulable, checking in increasing core index order (Line 3). If none exists, the function returns true (Line 4). Otherwise, it attempts to migrate tasks from core i to a neighboring core, checking cores in increasing distance from itself (Lines 5-10). For each distance value, dist, it checks whether a migration to the core $(i - \text{dist})$ is successful (Lines 6-7) before checking the core $(i + \text{dist})$ (Lines 8-10). Since all cores with index less than i are already schedulable, we only need to check schedulability for the higher-index core before attempting migration (Line 9). If a migration is successful, we continue to the next iteration of the while loop and repeats the process (Line 7 and Line 10). If it is not feasible to migrate tasks to any of the cores, the function returns false (Line 11).

To migrate tasks, we attempt to move a single task to the chosen target core, followed by a resource redistribution. As long as the target core remains schedulable after this redistribution, we keep the migration. Otherwise, we revert to the previous resource allocation and attempt to migrate

Algorithm 3 Task Inflation

```
 1: function INFLATE(m)                                    ▷ m: mode
 2:     i = −1
 3:     dist = 0
 4:     for j ∈ Tᵐ do
 5:         if ϱ_{j,max} − p_j^m > dist then
 6:             dist = ϱ_{j,max} − p_j^m
 7:             i = j
 8:     if i < 0 then return false
 9:     while p_i^m ≤ ϱ_{i,max} do
10:         p_i^m += ρ                                      ▷ Update period
11:         𝒦_i^m = GetController(𝒦_{(i)}, p_i^m)           ▷ Find the controller
12:         if 𝒦_i^m then return true                       ▷ Eventual guarantee
```

Algorithm 4 Transition Allocation

```
 1: function TRANSITION(m′, m)   ▷ m: new mode, m': old mode
 2:     jobType = 0            ▷ 0: carry-over job, 1: first new job
 3:     InitDelays(𝒯ᵐ′, 𝒯ᵐ)
 4:     while true do
 5:         delayed = false
 6:         failedCore = McpAnalysis(m′, m)
 7:         if !failedCore then return SCHEDULABLE
 8:         for ; jobType ≤ 1; jobType++ do
 9:             i = MaxSlackTask(jobType, m, m′, failedCore)
10:             j = 𝒦_i^{m′}
11:             if !jobType and d_i^c(m′, m) < ϱ_{i,j} then
12:                 d_i^c(m′, m) += ρ                        ▷ Delay
13:                 delayed = true
14:             j = 𝒦_i^m
15:             if d_i(m′, m) < ϱ_{i,j} then
16:                 d_i(m′, m) += ρ                          ▷ Delay
17:                 delayed = true
18:                 break
19:         if delayed then continue
20:         if !INFLATE(m) then break
21:         jobType = 0
22:         InitDelays(𝒯ᵐ′, 𝒯ᵐ)
23:     return UNSCHEDULABLE
```

another task. To select a candidate task, we iterate over the sorted list of tasks in the same *direction* of the target core. For migrating to a higher-index core, we consider tasks in reverse order of $\gamma_{i,m}$, and vice versa. Intuitively, we move less (resp. more) resource-sensitive tasks onto less (resp. more) resource-sensitive cores.

Inflation. The goal of task inflation is to increment a task's current assigned period by ρ (the control parameter) and then find the controller with the largest ϱ such that the increased period falls within its range. Algorithm 3 shows how this is done. We select the task τ_i whose current assigned period p_i^m is farthest away from its maximum safe period $\varrho_{i,max}$ for inflation (Lines 2-7). If no such task exists, the function returns false. Otherwise, in Lines 9-12, we increase the period and search for a (potentially) new controller for τ_i. If there is no controller in $\mathcal{K}_i$ that has a safe period equal to the updated p_i^m, we increment p_i^m again until a controller is found. Note that a controller is *guaranteed* to be found eventually, since the p_i^m will eventually reach the maximum safe period defined by some controller. Once a controller is found, we save it as $\mathcal{K}_i^m$ and keep the updated p_i^m, then return true.

C. Mode transition allocation

After obtaining a feasible allocation for every mode, DECNTR proceeds to check whether the system is schedulable during every mode transition, using the transition schedulability test in Section VI. If a transition $(m′, m)$ is not schedulable, then there exists at least one core in mode m that is unschedulable, and this happens because of the additional load from carry-over jobs from mode $m′$ (since m is schedulable in isolation). Leveraging our controllers' ability to tolerate some delay between sampling time points and to allow changes of sampling periods, we can reduce the transition load by extending the deadlines of carry-over jobs and/or increasing the task periods in mode m. The former strategy has only a transient effect on robustness, since it delays the actuation time of at most one job per task during the transition. In contrast, the latter strategy affects all jobs released in the new mode, and may affect robustness for a longer duration. To minimize potential negative impact on robustness, we prioritize extending the deadlines of carry-over jobs over increasing task periods.

Algorithm 4 shows the pseudo-code for achieving schedulability for a mode transition. DECNTR invokes this function for each mode transition $(m′, m) \in \mathcal{R}$ in the system. It first initializes $d_i(m′, m) = p_i^m$ for each task τ_i in m, and $d_i^c(m′, m) = p_i^{m′}$ for each carry-over task τ_i (Line 3). It then checks if the transition is schedulable (Lines 6-7). Here, $\mathsf{McpAnalysis}(m′, m)$ implements the transition schedulability test in Section VI; it returns 0 if the transition is schedulable, and the index of an unschedulable core (failedCore) otherwise. If the transition is schedulabe, the function returns schedulable and DECNTR moves to the next transition. Otherwise, we call $\mathsf{MaxSlackTask}(\dots)$ to retrieve the task i that has the largest difference between the maximum period of its assigned controller and its assigned period. If jobType $= 0$ (computing the delayed deadline for a carry-over job from $m′$), the assigned controller is $\mathcal{K}_i^{m′}$; otherwise, the assigned controller is $\mathcal{K}_i^m$. We then extend this task's delayed deadline(s) by the minimum amount, which is the configurable parameter ρ (Lines 12,16). Algorithm 4 attempts to delay the deadlines of all carry-over tasks before delaying deadlines of new tasks. After any increment, we repeat the process and recheck mode transition schedulability. If the transition is still unschedulable after extending the first deadlines of new tasks, we will attempt to inflate task periods for the target mode m using the same procedure (Algorithm 3) as in the single mode case (Line 20). If inflation is possible, we restart the process in a new iteration, checking carry-over tasks first again (Line 21) as inflating a task can change its controller; otherwise, the function returns with the failed allocation.

VI. SCHEDULABILITY ANALYSIS

This section presents the schedulability analysis used by DECNTR during its allocation. Given a multi-mode system

with the mode-change protocol as defined in Section III. Let $\mathcal{A}$ be an allocation of the system, which specifies

$$\left\{ c_k^m, w_k^m, \mathsf{core}_i^m, \mathcal{K}_i^m, p_i^m, \mathsf{del}_i(m', m), d_i^c(m', m), d_i(m', m) \right\}$$

for all $m \in \mathcal{M}$, $0 \le k < r$, $\tau_i \in \mathcal{T}^m$, $\mathcal{K}_i^m \in \mathcal{K}_{(i)}$, p_i^m is a valid period of $\mathcal{K}_i^m$, and $(m', m) \in \mathcal{R}$. Then, the system is schedulable under $\mathcal{A}$ iff it is schedulable in each mode $m \in \mathcal{M}$ and in each transition $(m', m) \in \mathcal{R}$.

Notation. We begin by recalling the notation used in $\mathcal{A}$. First, c_k^m and w_k^m are the numbers of cache and bandwidth partitions assigned to core k in mode m, respectively. Second, core_i^m, $\mathcal{K}_i^m$ and p_i^m are the core, controller and sampling period assigned to task τ_i in mode m. Finally, $\mathsf{del}_i(m', m)$ indicates whether we will delay the deadline of a job of τ_i during the transition (m', m); if so, $d_i^c(m', m)$ is the delayed deadline of the carry-over job of τ_i (if τ_i has a carry-over job) and $d_i(m', m)$ is the delayed deadline of the first job of τ_i released in m. We denote by e_i^m the WCET of τ_i in mode m, i.e., $e_i^m = e_i(c_k^m, w_k^m)$ where $k = \mathsf{core}_i^m$. Let $\mathcal{T}_k^m$ denote the set of tasks that are mapped onto core k in mode m under the allocation $\mathcal{A}$. Then, the utilization of a core k is $U_k^m = \sum_{\tau_i \in \mathcal{T}_k^m} \frac{e_i^m}{p_i^m}$.

A. Mode schedulability

The schedulability of a mode m in isolation can be determined using the standard demand-based EDF schedulability test, except that the WCET e_i^m of each task τ_i in mode m is dependent on the allocation. Specifically, the demand bound function (DBF) of a task τ_i in mode m is $\forall t \ge 0 : dbf_{\tau_i}^m(t) = \lfloor \frac{t}{p_i^m} \rfloor e_i^m$. Since tasks on each core are scheduled under EDF, the DBF of a core k in mode m is given by

$$\forall t \ge 0 : dbf_k^m(t) = \sum_{\tau_i \in \mathcal{T}_k^m} dbf_{\tau_i}^m(t). \tag{10}$$

The next theorem states the schedulability condition for mode m. Its proof follows directly from the existing analysis in [18].

Theorem 10. *The system is schedulable in mode m if for all $0 \le k < r$, $U_k^m \le 1$ and for all $t < L_k^m$, $dbf_k^m(t) \le t$, where*

$$L_k^m = \max \left\{ D_k^m, \frac{1}{1 - U_k^m} \right\}.$$

and D_k^m denotes the maximum of the assigned periods (deadlines) of the tasks in $\mathcal{T}_k^m$.

B. Mode transition schedulability

The mode transition schedulability can be established using a similar analysis as in Omni [21]. However, since Omni does not delay jobs during a transition, we extend its analysis slightly to handle delayed jobs. Below, we sketch key schedulability conditions and accompanying arguments.

Intuitively, for each core, we compute each task's worst-case demand for an interval t during a mode transition; if their total demand of all tasks on a core is no more than t, for all $t > 0$, then the core is schedulable. Towards this, we treat each carry-over job as a new job released at the MCR

instant, with its WCET equal to the carry-over job's maximum remaining execution time and its absolute deadline the same as the carry-over absolute deadline (if the task is not delayable) or the carry-over delayed deadline (otherwise). As the worst-case demand of a core for any interval that begins *after* the MCR instant will only include demands from jobs released in the new mode, we can compute its demand using the same method as in an isolated mode m. Thus, we only need to consider the worst-case demands for intervals that begin at the MCR instant (referred to as transition demands).

Like in Omni [21], a task's worst-case demand occurs when (i) it has a job with deadline at the end of the interval; (ii) new jobs are released as soon as possible; and (iii) the carry-over (unfinished) job was executed as late as possible. Based on these conditions, we can bound the transition demands of the jobs for each task type, using conventional demand-bound analysis, while taking into consideration that 1) a job's WCET depends on the current resource allocation (of the new mode), and 2) a task may be delayable, in which case either their carry-over job or their first job will be delayed, and thus the delayed deadline is used for computing the demand instead of the original deadline.

More concretely, to establish the schedulability of a mode transition, we derive the worst-case demands of all jobs on a core during the transition. The carry-over demand during a mode transition (m', m) consists of (i) the demands of carry-over jobs of carry-over tasks, and (ii) the demands of new jobs of both carry-over and new tasks.

Transition demands of new tasks. Since jobs of a new task are only released in the new mode, the task's transition demand can be computed similarly to the task's demand in an isolated mode. The only exception is that, if the task is delayable, then the delayed deadline is used for its first job.

Lemma 11. *The DBF of a **new** task τ_i during a transition (m', m) is given by*

$$dbf_{i,N}^{m',m}(t) = \lfloor \frac{t - d_i(m', m) + p_i^m}{p_i^m} \rfloor e_i^m \tag{11}$$

where e_i^m and p_i^m are the WCET and sampling period of τ_i in mode m, respectively. Further, $d_i(m', m)$ is the delayed deadline of τ_i's first job if it is a delayble task (that is, $\mathsf{del}_i(m', m) = 1$), and $d_i(m', m) = p_i^m$ otherwise.

Proof: First, suppose τ_i is a non-delayable new task. Then, its worst-case demand during the transition is identical to its worst-case demand in mode m in isolation, since all jobs are released and have deadlines in the new mode and following their corresponding new-mode periods. Thus, $dbf_{i,N}^{m',m}(t) = \lfloor \frac{t}{p_i^m} \rfloor e_i^m$, which is equivalent to Eq. (11) since $d_i(m', m) = p_i^m$ in this case.

Next, consider the case where τ_i is a delayable task. Then, the (delayed) deadline for its first new job is $d_i(m', m)$, whereas the deadline of a subsequent job is the same as the period p_i^m. Further, according to our mode change protocol (c.f. Section III), τ_i releases its first job immediately after the MCR arrives, and each subsequent job is released at the

absolutely deadline of the previous job. Hence, the maximum demand of τ_i in the transition interval of length t starting from the MCR instant is at most $\lfloor \frac{t - d_i(m', m) + p_i^m}{p_i^m} \rfloor e_i^m$. In other words, Eq. 11 holds. ∎

Transition demands of carry-over tasks. For the demand of a carry-over task τ_i however, we need to consider two cases: 1) τ_i has a carry-over job and its carry-over job has a deadline of $d_i^c(m', m)$, and 2) τ_i has no carry-over job and its first job has a deadline of $d_i(m', m) > p_i^m$. We can easily show that the worst-case demand of τ_i in the first case is always higher than the demand in the second case, for all time intervals t. Hence, we can derive the maximum demand for τ_i based on the worst-case scenario with the carry-over job. Let $d_i^{m'}$ be the deadline of the carry-over job. Then, $d_i^{m'} = p_i^{m'}$ if τ_i is not delayable and $d_i^{m'} = d_i^c(m', m)$ if it is. With this, we can compute the DBF of τ_i during a transition (m', m) as follows:

Lemma 12. *The DBF of a **carry-over** task τ_i during a transition (m', m) is given by*

$$dbf_{i,CO}^{m',m}(t) = \lfloor \frac{t}{p_i^m} \rfloor e_i^m + E_i^{m',m}, \qquad (12)$$

where $E_i^{m',m}$ is τ_i's maximum carry-over demand, defined by

(a) *If $t \le d_i^{m'} - p_i^{m'}$, then $E_i^{m',m} = 0$.*

(b) *If $d_i^{m'} - p_i^{m'} < t < p_i^m$, then*

$$E_i^{m',m} = \min\{e_i^m, t + \max\{0, e_i^m - e_i^{m'}\}\}.$$

(c) *Otherwise, $t' = (t - p_i^m) \mod p_i^m$, and*

$$E_i^{m',m} = \min\{e_i^m, \max\{0, t'\} + \max\{0, e_i^m - e_i^{m'}\}\}.$$

Proof sketch: The demand of a carry-over task during the transition consists of (1) the demand from all new job releases of τ_i in mode m, which is given by $\lfloor \frac{t}{p_i^m} \rfloor e_i^m$ (the first term in the RHS of Eq. (12)); and (2) the demand from the carry-over job of τ_i itself. Thus, to prove the lemma, we need to show that $E_i^{m',m}$ correctly bounds this carry-over demand.

Recall that the carry-over job has a (delayed) deadline of $d_i^{m'}$. Thus, the amount of time that its deadline is delayed is equal to $d_i^{m'} - p_i^{m'}$, since $p_i^{m'}$ is its original deadline (since it is released in the old mode m'). There are two scenarios:

If $t \le d_i^{m'} - p_i^{m'}$, then the length of the time interval we are considering, t, is no more than the amount of time we can delay the carry-over job. Thus, the carry-over job imposes no demand, since its deadline falls outside the interval. Thus, $E_i^{m',m} = 0$ and the case (a) of the lemma holds.

Otherwise, the maximum carry-over demand can be conservatively computed using the analysis from Omni. This is possible, because Omni does not delay carry-over jobs and thus the worst-case carry-over demand under Omni is at least equal to or larger than the carry-over demand under DECNTR. The two cases (b) and (c) of the lemma come directly from the results in Omni [21] (see Lemma 2 and Lemma 3 in [21]), modified appropriately to reflect our different notation. ∎

The next lemma gives the mode change demand of a core during a transition (m', m). Its proof is established based directly on Eq. (11) and Eq. (12).

Lemma 13. *The maximum demand of a core k during a mode transition (m', m) is bounded by*

$$dbf_k^{m',m}(t) = \sum_{\tau_i \in \mathcal{T}_k^N} dbf_{i,N}^{m',m}(t) + \sum_{\tau_i \in \mathcal{T}_k^{CO}} dbf_{i,CO}^{m',m}(t)$$

where $\mathcal{T}_k^N$ and $\mathcal{T}_k^{CO}$ are the set of new tasks, and carry-over tasks that are mapped onto core k in mode m.

Based on Lemma 13 and the mode schedulability, we can directly imply the next theorem.

Theorem 14. *The system is schedulable during a mode transition from m' to m if (1) it is schedulable in modes m' and m (Theorem 10), and (2) for all $0 \ge k < r$, for all $t > 0$, $dbf_k^{m',m}(t) \le t$.*

VII. EVALUATION

To evaluate the effectiveness of DECNTR, we conducted a series of experiments using real-time benchmarks and an automotive case study. Our goal was to evaluate (1) how much DECNTR improves schedulability over the state of the art, and (2) how effective DECNTR is in increasing system robustness.

Algorithms. We compare DECNTR against Omni [21], the state-of-the-art technique for multi-mode, multi-core task and resource allocation. Like DECNTR, Omni computes an allocation of tasks, cache and memory bandwidth to cores in each mode to ensure the multi-mode system is schedulable. However, Omni assumes the tasks' periods are given a priori and jobs' deadlines cannot be delayed. To enable a more direct comparison for Omni, we develop two extensions of Omni (which performed strictly better than Omni in our evaluation):

1) Omni-P: we set task periods to be the periods assigned by DECNTR, then use Omni to find an allocation.

2) Omni-D: we use Omni to find an allocation assuming tasks are assigned their reference periods (defined below), but extend it to support delaying job deadlines as DECNTR does.

A. Schedulability evaluation with benchmarks

Workload. To evaluate DECNTR's effectiveness across a range of loads and timing parameters, we used a collection of 11 benchmarks from the PARSEC [7], SPLASH2x [52], DIS [30], and Isolbench [50] suites as workloads. To obtain their WCETs, we ran each benchmark under all possible cache and BW configurations on an Intel Xeon E5-2683 v4 processor with 16 cores and a 40MB L3 cache. The cache is divided into $C_{\max} = 20$ equal partitions using Intel's CAT. Using the method from [55], we measured a maximum guaranteed bandwidth of 1.4 GB/s, which we divided into $W_{\max} = 20$ partitions of 70MB/s each using MemGuard. We disabled any hardware feature that can lead to nondeterministic timings and used a single-threaded execution mode.

We then generated tasksets using a similar approach to [21] by randomly picking tasks from our benchmarks to fill a target taskset utilization. Each individual task's utilization was
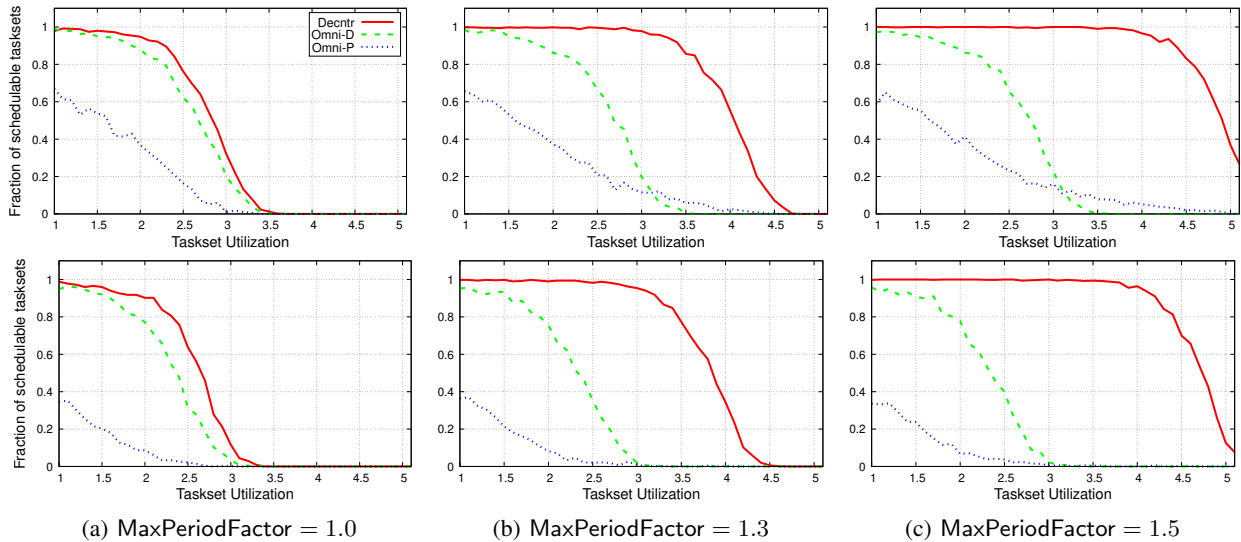
Fig. 5: System schedulability under different max period factors, 2 modes top, 4 modes bottom.

selected from a bimodal distribution with a $\frac{6}{9}$ likelihood to fall within [0.01, 0.4] and a $\frac{3}{9}$ likelihood to fall within [0.4, 0.9].

Each task τ_i was assigned a *reference period* (p_i^{ref}), defined as the ratio of its reference WCET, $e_i(C_{\max}, W_{\max})$, to its utilization, rounded to the nearest multiple of ρ. We set $\rho = 1$ms in all experiments. We then defined the period range $[\rho_i, \varrho_i]$ for τ_i based on the reference period, where the minimum period ρ_i is the smallest multiple of ρ that is above the task's reference WCET, and the maximum period $\varrho_i = k_{\max} \times p_i^{\text{ref}}$ for some configurable value $k_{\max}$ called the *max period factor* (MPF). The task's period can be any value $k\rho$ in the range $[\rho_i, \varrho_i]$, where k is a positive integer. Note that Omni-D uses the reference period as the task's period, whereas DECNTR assigns the task's period based on its period range.

We generated tasksets with (reference) utilizations in the range [1.0, 5.0] at steps of 0.1. For each utilization step, we generated 500 independent tasksets per mode, for a total of 41,000 tasksets in a 2 mode system (82,000 tasksets with 4 modes). We generated 10 experiments (half with 2 modes and half with 4), with varying MPFs (1.0 − 1.5). The generated tasks had a 20% probability of being a carry-over task.

Results. Fig. 5 shows the fraction of schedulable tasksets under the three algorithms for tasksets with MPFs being 1.0, 1.3 and 1.5. The results show that DECNTR is significantly better at scheduling tasksets compared to Omni when they use the same assigned periods (DECNTR vs. Omni-P). Our improvement factor at a medium load of 2.0 reference utilization ranges from 2.4× (2-mode systems, MPF = 1.0) to 11× (4-mode system, MPF = 1.5). Note that as the MPF increases from 1.0 to 1.5, both DECNTR and Omni-P can schedule more tasks. This is expected, since DECNTR can assign larger periods to tasks in all modes with larger MPF. Interesting, this trend holds for DECNTR in both 2- and 4-mode systems, whereas Omni-P only achieves improved schedulability under 2 modes, highlighting that DECNTR

scales better with the number of modes. We also see that when the MPF is limited to 1.0, DECNTR not only schedules more tasksets than Omni-D but also reduces task periods by 5-21% on average.

Runtime. Table I reports the running time of each algorithm for the 2-mode systems (MPF = 1.3) in our experiments. We observe that DECNTR is consistently more efficient than both Omni variants (3.4-8.2× faster on average).

B. CPS Case Study

To evaluate the benefits of our co-design approach in a real-world CPS, we conducted a case study of an electric vehicle system. The system contains 5 different control tasks: battery [14], DC motor [56], active suspension, [33], automatic cruise control [25], and lane-keeping assistant [57]. We consider both robustness and schedulability in our evaluation.

Robustness. For each plant, we designed multiple safety controllers $(\mathcal{K}_j, \rho_j)$ as in (8), and simulated the closed-loop performance under additive Gaussian noise in the control input. Due to space constraints, we present three of the five simulations to show the effect of the increase in sampling rate on the safety of CPSs.

DC Motor: We use the model and safe set presented in the example of Section IV-C. The results are presented in Fig. 6a for the state evolution of w under the input sequence V. Our safety property states that $w(t) \in [4.5, 5]$ and $V(t) \in [-1, 1]$ for all $t \geq 0$. The figure demonstrates that w drifts into the unsafe region for longer intervals when p is larger.

Automatic Cruise Control (ACC): We use the model and safety conditions from [25], which maximizes speed while keeping a safe distance from a forward vehicle. The input is acceleration, a, and the states are the two vehicle velocities and the distance from the forward car (v, v_f, d). We implemented the ct-LTI given by $\dot{v} = a$, $\dot{v}_f = 0$, and $\dot{d} = v_f - v$ and we impose the safe set given by the conditions $a = [-1, 1]$,

	1.00 Taskset utilization				2.00 Taskset utilization				3.00 Taskset utilization				4.00 Taskset utilization			
	min	max	**avg.**	99th	min	max	**avg.**	99th	min	max	**avg.**	99th	min	max	**avg.**	99th
DECNTR	0.02	3.18	**0.06**	0.35	0.11	4.39	**0.24**	1.68	0.14	5.11	**0.51**	3.11	0.24	12.5	**0.80**	6.72
Omni-P	0.03	24.3	**0.24**	2.28	0.13	88.0	**0.96**	12.8	0.21	57.5	**1.71**	19.9	0.28	193	**3.37**	54.0
Omni-D	0.03	33.7	**0.29**	3.42	0.13	102	**1.71**	30.2	0.20	85.9	**4.21**	54.7	0.30	64.2	**5.24**	38.6

TABLE I: Runtime of the three algorithms in seconds.

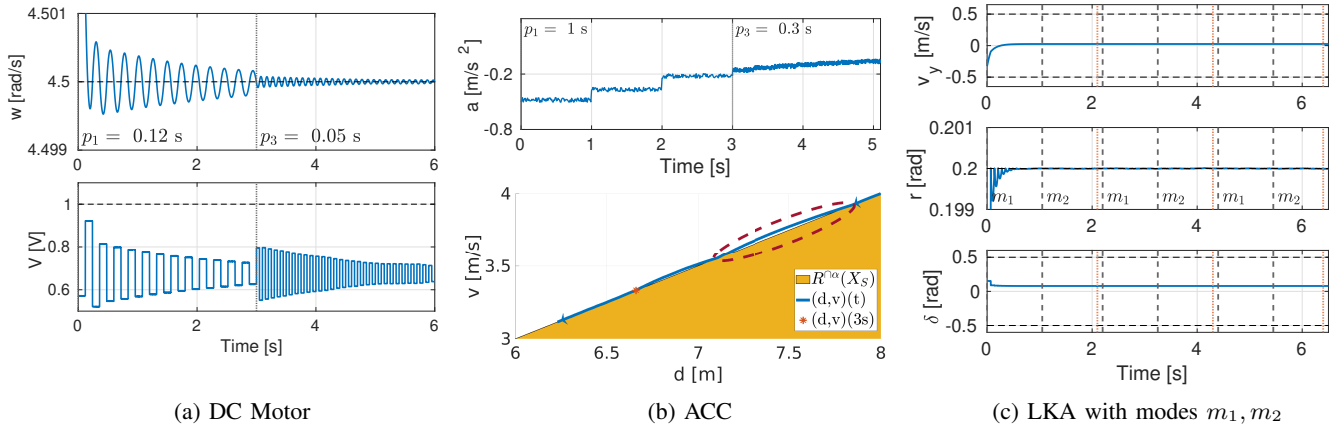(a) DC Motor   (b) ACC   (c) LKA with modes m_1, m_2

Fig. 6: Control simulations demonstrating robustness as sampling rates change

$v \in [-5, 5]$, $v_f \in [-5, 5]$, $d \in [5, 20]$, $v \leq \frac{d}{t_{safe}}$, where $t_{safe} = 2s$, to represent a minimum time-gap that allows the first vehicle to react in an emergency. Fig. 6b shows the results of the control reducing the distance between the vehicles while trying to keep the minimum time-gap with initial state values $(4.4, 3, 9) \in R^{\cap \alpha}(X_S)$. The controller with $p = 1s$ cannot satisfy the safety property due to significant inter-sampling time (highlighted with a dashed red ellipsoid). After $3s$ we switch to a controller with $p = 0.3$ with no safety issues.

Lane Keeping Assistant (LKA): The LKA control steers the horizontal speed and the yaw angle rate of the vehicle (v_y, r), respectively, provided changes in the steering wheel angle δ as input. We define $\rho = 0.01s$, and the safe set given by $v_y \in [-0.5, 0.5]$, $r \in [-0.2, 0.2]$, and $\delta \in [-0.5, 0.5]$. We repeatedly run the controller in two modes m_1, m_2 and simulate periods provided by DECNTR. Fig. 6c shows the result where the periods are $p_1 = 7\rho$ in m_1 and $p_2 = 7\rho$ in m_2. Notably, DECNTR's allocation requires the LKA task to have a delayed job release ($p = 10\rho$) for every transition (m_2, m_1). The result confirms that even when extending jobs' deadlines during mode transitions, the system under DECNTR allocation remains safe.

Schedulability. We generated tasksets with the controllers in our case study, using the methodology in Section VII-A. To generate a taskset at a target utilization, we randomly selected a single task until the taskset utilization reaches the target utilization, where each task's reference utilization is computed as the reference WCET over the *max* period: $\frac{e_i(C_{max}, W_{max})}{\varrho_{i,max}}$. Since the reference utilization is small compared to the utilization when assigned resources and a period from DECNTR, we generate tasksets with utilization range of [0.1,4]. For WCETs, we profiled the controllers on resource constrained hardware relevant for our case study: a Raspberry Pi 3 Model B+ with 512 KB of shared cache and a guaranteed BW throughput of
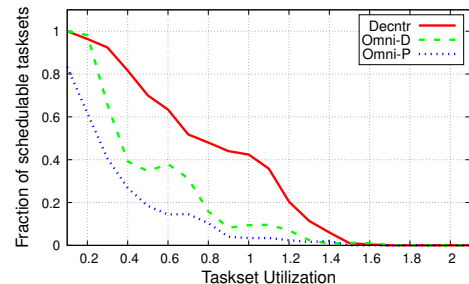

Fig. 7: Case study schedulability, 2 modes.

100 MB/s (20 partitions of 5 MB/s each).

Fig. 7 shows the schedulability results for systems with 2 modes and 500 tasksets per utilization step. As expected, DECNTR outperforms both Omni variants. For example, at reference utilization of 0.5, DECNTR can schedule $3.5\times$ more tasksets than Omni-P and $1.8\times$ than Omni-D. Notably, DEC-NTR achieves this schedulability improvement even with much smaller task periods (43% on average) than Omni-D, and thus it also provides much higher robustness.

VIII. CONCLUSION

In this paper, we introduced DECNTR, along with a controller-switching co-design approach, which draws inspiration from the domains of multi-mode resource allocation and control theory. Our solution demonstrates remarkable performance in terms of safety, robustness, and timeliness for multi-mode CPS deployed on multi-core platforms. DECNTR accomplishes this by harnessing multiple control implementations to dynamically adjust task periods, thereby enhancing system safety and significantly improving schedulability when compared to existing state-of-the-art solutions. As part of our future endeavors, we aim to expand our techniques to encompass plants with nonlinear dynamics.

REFERENCES

[1] M. Ahmed and N. Fisher. Tractable schedulability analysis and resource allocation for real-time multimodal systems. *ACM Trans. Embed. Comput. Syst.*, 13(2s), jan 2014.

[2] A. D. Ames, X. Xu, J. W. Grizzle, and P. Tabuada. Control barrier function based quadratic programs for safety critical systems. *IEEE Transactions on Automatic Control*, 62(8):3861–3876, 2016.

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, 2003.

[4] K.-E. Arzen, A. Cervin, J. Eker, and L. Sha. An introduction to control and scheduling co-design. In *Conference on Decision and Control (CDC)*, volume 5, pages 4865–4870 vol.5, 2000.

[5] H. Baek, K. G. Shin, and J. Lee. Response-time analysis for multi-mode tasks in real-time multiprocessor systems. *IEEE Access*, 8:86111–86129, 2020.

[6] D. Bertsekas. Infinite time reachability of state-space regions by using feedback control. *IEEE Transactions on Automatic Control*, 17(5):604–613, 1972.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[8] F. Blanchini. Set invariance in control. *Automatica*, 35(11):1747–1767, 1999.

[9] F. Blanchini and S. Miani. *Set-theoretic methods in control*, volume 78. Springer, 2008.

[10] A. Burns. System mode changes-general and criticality-based. In *WMC*, 2014.

[11] G. Buttazzo, M. Velasco, and P. Marti. Quality-of-control management in overloaded real-time systems. *IEEE Transactions on Computers*, 56(2):253–266, 2007.

[12] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Real-Time Systems Symposium (RTSS)*, 1999.

[13] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[14] Y. Chen, D. Huang, Q. Zhu, W. Liu, C. Liu, and N. Xiong. A new state of charge estimation algorithm for lithium-ion batteries based on the fractional unscented kalman filter. *Energies*, 10(9):1313, 2017.

[15] H. S. Chwa, K. G. Shin, and J. Lee. Closing the gap between stability and schedulability: A new task model for cyber-physical systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 327–337. IEEE, 2018.

[16] X. Dai, S. Zhao, Y. Jiang, X. Jiao, X. S. Hu, and W. Chang. Fixed-priority scheduling and controller co-design for time-sensitive networks. In *Conference on Computer-Aided Design*, pages 1–9, 2020.

[17] D. de Niz and L. T. X. Phan. Partitioned Scheduling of Multi-Modal Mixed-Criticality Real-Time Systems on Multiprocessor Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[18] A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *Real-Time Systems Symposium (RTSS)*, pages 78–87, 2013.

[19] M. Gaid, A. Cela, and Y. Hamam. Optimal integrated control and scheduling of networked control systems with communication constraints: application to a car suspension system. *IEEE Transactions on Control Systems Technology*, 14(4):776–787, 2006.

[20] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen. Dna: Dynamic resource allocation for soft real-time multicore systems. In *RTAS*, 2021.

[21] R. Gifford and L. T. X. Phan. Multi-mode on multi-core: Making the best of both worlds with omni. In *Real-Time Systems Symposium (RTSS)*, pages 118–131, 2022.

[22] S. Goddard and X. Liu. A variable rate execution model. In *ECRTS*, pages 135–143, 2004.

[23] J. Goossens and P. Richard. Partitioned scheduling of multimode multiprocessor real-time systems with temporal isolation. In *Conference on Real-Time Networks and Systems (RTNS)*, pages 297–305, 2013.

[24] A. Gujarati, M. Nasri, and B. B. Brandenburg. Quantifying the Resiliency of Fail-Operational Real-Time Networked Control Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 106, pages 16:1–16:24, 2018.

[25] G. Gunter and D. Work. Safe driving with control barrier functions in mixed autonomy traffic when cut-ins occur. In *European Control Conference (ECC)*, pages 411–416. IEEE, 2022.

[26] Intel. Improving real-time performance by utilizing cache allocation technology, Apr. 2015. White Paper.

[27] P. Jagtap, S. Soudjani, and M. Zamani. Formal synthesis of stochastic systems via control barrier certificates. *IEEE Transactions on Automatic Control*, 2020.

[28] O. Kwon, G. Schwäricke, T. Kloda, D. Hoornaert, G. Gracioli, and M. Caccamo. Flexible cache partitioning for multi-mode real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1156–1161, 2021.

[29] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[30] J. Musmanno. Data intensive systems (dis) benchmark performance summary. *AFRL Technical Report AFRL-IF-RS-TR-2003-198*, 2003.

[31] M. Negrean, S. Klawitter, and R. Ernst. Timing analysis of multi-mode applications on autosar conform multi-core systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013.

[32] V. Nelis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Euromicro Conference on Real-Time Systems*, 2009.

[33] V. P. Patel, V. S. Jatti, and V. S. Jatti. Design of quarter car model for active suspension system and control optimization. In *Optimization Methods for Structural Engineering*, pages 211–225. Springer, 2023.

[34] L. Phan, S. Chakraborty, and P. Thiagarajan. A multi-mode real-time calculus. In *Real-Time Systems Symposium (RTSS)*, 2008.

[35] L. T. X. Phan, S. Chakraborty, and I. Lee. Timing analysis of mixed time/event-triggered multi-mode systems. In *Real-Time Systems Symposium (RTSS)*, 2009.

[36] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional analysis of multi-mode systems. In *Euromicro Conference on Real-Time Systems*, 2010.

[37] S. Prajna and A. Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *International Workshop on Hybrid Systems: Computation and Control*, pages 477–492, 2004.

[38] P. Rattanatamrong and J. A. Fortes. Mode transition for online scheduling of adaptive real-time systems on multiprocessors. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 25–32, 2011.

[39] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[40] D. Roy, S. Ghosh, Q. Zhu, M. Caccamo, and S. Chakraborty. Goodspread: Criticality-aware static scheduling of cps with multi-qos resources. In *Real-Time Systems Symposium (RTSS)*, pages 178–190. IEEE, 2020.

[41] M. Rungger and P. Tabuada. Computing robust controlled invariant sets of linear systems. *IEEE Transactions on Automatic Control*, 62(7):3665–3670, 2017.

[42] L. Scheuvens, T. Hößler, A. N. Barreto, and G. P. Fettweis. Wireless control communications co-design via application-adaptive resource management. In *5G World Forum (5GWF)*, pages 298–303, 2019.

[43] M. Schmitz, B. Al-Hashimi, and P. Eles. A co-design methodology for energy-efficient multi-mode embedded systems with consideration of mode execution probabilities. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 960–965, 2003.

[44] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty. Constraint-driven synthesis and tool-support for flexray-based automotive control systems. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 139–148, 2011.

[45] Y. Shin, D. Kim, and K. Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems. In *DAC*, 2000.

[46] D. Simon, D. Robert, and O. Sename. Robust control/scheduling co-design: application to robot control. In *Real Time and Embedded Technology and Applications Symposium*, pages 118–127, 2005.

[47] D. Soudbakhsh, L. T. X. Phan, A. Annaswamy, and O. Sokolsky. Co-design of arbitrated network control systems with overrun strategies. *IEEE Transactions on Control of Network Systems*, 5(1):128–141, 2018.

[48] D. Soudbakhsh, L. T. X. Phan, A. Annaswamy, O. Sokolsky, and I. Lee. Co-design of control and platform with dropped signals. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[49] A. J. Taylor, V. D. Dorobantu, R. K. Cosner, Y. Yue, and A. D. Ames. Safety of sampled-data systems with control barrier functions via approximate discrete time models. In *Conference on Decision and Control (CDC)*, pages 7127–7134. IEEE, 2022.

[50] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

[51] R. Vidal, S. Schaffert, J. Lygeros, and S. Sastry. Controlled invariance of discrete time systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 437–451. Springer, 2000.

[52] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[53] M. Xu, R. Gifford, and L. T. X. Phan. Holistic multi-resource allocation for multicore real-time virtualization. In *DAC*, page 168, 2019.

[54] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[55] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

[56] L. Zaccarian. Dc motors: dynamic model and control techniques, 2012.

[57] X. Zhou, H. Shen, Z. Wang, and J. Wang. Individualizable vehicle lane keeping assistance system design: A linear-programming-based model predictive control approach. *IFAC-PapersOnLine*, 55(37):518–523, 2022.

[58] Q. Zhu and A. Sangiovanni-Vincentelli. Codesign methodologies and tools for cyber–physical systems. *Proceedings of the IEEE*, 106(9):1484–1500, 2018.