Controllers for Edge-Cloud Cyber-Physical Systems

Tingan Zhu*, Prateek Ganguli*, Arkaprava Gupta*, Shengjie Xu*,
Luigi Capogrosso†, Enrico Fraccaroli*†, Marco Cristani† Franco Fummi†, and Samarjit Chakraborty*
*The University of North Carolina at Chapel Hill, USA †University of Verona, Italy

***Abstract*—Deep Neural Networks (DNNs) are now widely used in Cyber-Physical Systems (CPSs), both for sensor data or perception processing and also as neural network controllers. However, resource constraints often prevent a full local deployment of the DNNs, *e.g.*, on edge devices. Implementing them on the cloud, on the other hand, is associated with large delays and large volumes of data transfers. This has resulted in the emergence of *Split Computing (SC)*, where a part of the DNN is implemented on an edge device and the rest in the cloud. However, how to design control strategies with such a setup has not been sufficiently investigated in the past. In this paper, we study controller design strategies where state estimates from sensor data processed on an edge device are combined with estimates obtained from the cloud. While the former is associated with low delays, the state estimates have higher errors or uncertainties. The estimates from the cloud, obtained with larger DNNs, are, however, delayed. We show that the problem of sizing the DNNs on the edge and the cloud can be formulated as an optimization problem with the goal of maximizing system safety.**

***Index Terms*—Split Computing, Early Exit, Deep Neural Networks, Cyber-Physical Systems.**

I. Introduction

The algorithmic core of most Cyber-Physical Systems (CPSs) like autonomous cars or industrial production systems comprises multiple feedback control loops [1]. Today, the sensor data processing in such controllers is mainly implemented as Deep Neural Networks (DNNs). The control strategy is often also a Neural Network (NN) controller. Figure 1 illustrates such a setup. Here, the system to be controlled (called a "*plant*" in control theory) is modeled as a set of differential equations $\dot{\mathbb{x}} = \mathcal{S}(\mathbb{x}, \mathbb{u})$, where the system's *state* $\mathbb{x}$ is a vector of state components $[x_1, \ldots x_n]$, $\mathcal{S}$ could be a non-linear function. The *control input* $\mathbb{u}$ is a vector $[u_1, \ldots u_m]$.

The state of the plant $\mathbb{x}(t)$ at any time t needs to be inferred from the system's output $\mathbb{y} = \mathcal{G}(\mathbb{x}, \mathbb{u})$, where $\mathcal{G}$, like $\mathcal{S}$, is a function that models the dynamics of the plant. Based on the inferred value $\hat{\mathbb{x}}$ of the state $\mathbb{x}$, the controller computes a control input $\mathbb{u} = \xi(\hat{\mathbb{x}})$ and applies it to the plant using an actuator, to enforce a desired behavior on it. In the case of an autonomous vehicle, such a behavior might require moving the vehicle from a location A to B while satisfying safety properties like collision avoidance. In Figure 1, the control strategy ξ is implemented using Machine Learning (ML) as a NN controller. Further, the different components $y_1, \ldots, y_k$ of the output $\mathbb{y}$ are read by different sensors such as cameras, lidars, and radars in an autonomous vehicle. Figure 1 shows that the estimation $\hat{\mathbb{x}}$ of the state $\mathbb{x}$, that serves as an input to the NN controller, is done using a DNN responsible for sensor data processing. The input to such a DNN is the different components $y_1, \ldots, y_k$ of $\mathbb{y}$.

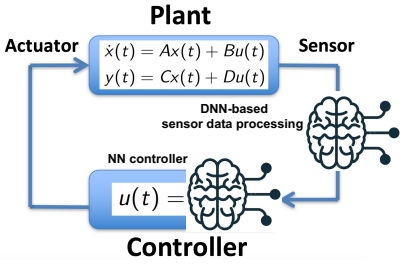


Figure 1: ML-enabled control.

However, implementing such DNNs locally on edge devices might not be possible due to resource constraints. A purely local implementation might require a smaller NN architecture, resulting in large uncertainties or errors in the state estimation. In other words, $|\mathbb{x} - \hat{\mathbb{x}}|$ would be large. On the other hand, the DNNs could be implemented on the cloud, where there are no resource constraints, thereby allowing larger DNNs to be deployed. The state of the plant could then be more accurately estimated, *i.e.*, $|\mathbb{x} - \hat{\mathbb{x}}|$ would be small. However, this would involve large volumes of data transfer from the local sensors to the cloud, which, along with the time necessary for computation or DNN inference, would result in large sensor-to-actuator delays.

A potential solution to the above problems lies in *Split Computing (SC)*, where the DNN is partitioned into "Head" or edge, and "Tail" or cloud components. The part of the DNN implemented on an edge device returns an inference quickly, albeit with potentially significant uncertainty in the inference quality. The part of the DNN implemented on the cloud returns a much higher quality inference but with a significant delay (see Figure 2).

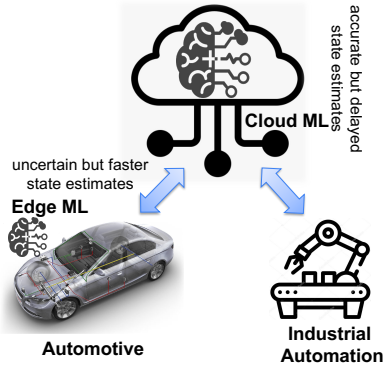


Figure 2: Edge-cloud CPS.

A. Contributions of this paper

While SC has been studied in the past, it is unclear how should a control strategy be designed to simultaneously exploit these two types of DNN inferences on the plant's state estimate. Further, as more resources are allocated to the edge device, its inference quality improves. Similarly, the smaller the DNN implemented on the cloud, its inference quality might reduce, but its associated delay might also decrease. This raises the question: What is the optimal DNN sizing (and resource allocation) on the edge and on the cloud? Answering this

question requires the identification of an objective function to be optimized. In this paper, we use the size of the reachable state space of the closed-loop system as an objective junction. Given an initial state (or a set of initial states), the *reachable state space* at any time t is the set of all states that can be reached by the plant & controller at time t. If the state $\mathbb{x}$ of the system can be accurately estimated, then the size of the reachable state space at any time t will be less than if uncertainty exists in the estimated state. In other words, the higher the value of $|\mathbb{x} - \hat{\mathbb{x}}|$, the more prominent will be the set of potential states that may be reached. Similarly, the variability in the time taken to infer $\hat{\mathbb{x}}$ (the estimate of $\mathbb{x}$) will also impact the states that may be reached. Since a larger reachable state space increases the chances that some unsafe states may be reached, the size of the state space is considered to be a common notion of safety. Other similar notions are discussed in Section II.

In addition to the size of the reachable state space, we also consider the magnitude of the control input as a metric to optimize. In an electric vehicle, for example, the magnitude of the control input might determine the size of a battery or the maximum current that may be drawn from it. We show that both the only cloud and the only edge solutions result in larger reachable sets and a higher magnitude of the control input $\mathbb{u}$. SC, or the hybrid edge-cloud solution, improves both metrics. However, changing the resource dimensioning and NN sizing in this hybrid setting impacts the solution, showing that this design space needs to be explored. Prompted by these results, in Section VII, we discuss some potential control strategy templates, whose optimization should be studied as a part of future work.

Paper organization: The following section provides some background on control theory and discusses two notions of safety, one of which we pursue in this paper. Section III provides a background on split or edge-cloud ML. This is followed by a controller design strategy in Section IV to exploit such an edge-cloud ML-based sensor data processing. Section V reports our experimental results, showing the performance of the only-edge, only-cloud, and two edge-cloud setups. Related work is discussed in Section VI, followed by some concluding remarks and an outline of potential future work in Section VII.

II. Control Theory Background

While the description of control systems in the earlier section was very general, for the rest of the paper, we will restrict ourselves to Linear Time-Invariant (LTI) systems and use the standard *state-space model*, where the state of the system is represented by a state vector $x(t) \in \mathbf{R}^n$ and the control input to the system by $u(t) \in \mathbf{R}^m$. Using these notations, the state-space model of a continuous, LTI system is given by:

$$\dot{x}(t) = A_c x(t) + B_c u(t) \,, \tag{1}$$

where $A_c \in \mathbf{R}^{n \times n}$ and $B_c \in \mathbf{R}^{m \times m}$ are matrices describing the dynamics of the system. Feedback control is enabled when $u(t)$ is adjusted based on the current system state $x(t)$. It is usually computed by a periodic real-time software task, which requires discretizing the continuous state-space model with a constant sampling period h. Assuming periodic sampling, *i.e.*, $t_{k+1} - t_k = h$, matrices A and B can be derived from Equation (1) such that: $x(t_{k+1}) = Ax(t_k) + Bu(t_k)$. We denote $x(t_k)$ as $x[k]$ and $u(t_k)$ as $u[k]$ and obtain the discrete model:

$$x[k+1] = Ax[k] + Bu[k] \,. \tag{2}$$

In the simplest case, the control input $u[k]$ is computed as:

$$u[k] = Kx[k] \,, \tag{3}$$

where $K \in \mathbf{R}^{m \times n}$ is the feedback gain. Many methods exist to design the feedback gain K with various stability, energy, and complexity considerations. While we restrict ourselves to LTI systems and controllers of the form in Equation (3) for the ease of exposition and the simplicity of the reachability analysis that is involved, the conclusions we draw also hold for more complex systems and controllers.

A. System-level safety

As mentioned earlier, we need an objective function to optimize while sizing the edge and the cloud DNNs. One possibility is to use system-safety as such an objective. There are different possible notions of system safety. One measure of system-level safety is the deviation of the closed-loop system's trajectory in its state space from its ideal trajectory or desired behavior. In Figure 3, the ideal (or nominal) trajectory is the black line in the state space, and a *safety pipe* is marked around this nominal trajectory (the light blue region). Trajectories that stay within this safety pipe over the entire time horizon of interest are considered safe. The optimization goal, in this case, is to size the edge and cloud DNNs and dimension edge and cloud resources to minimize the deviation from the nominal trajectory.

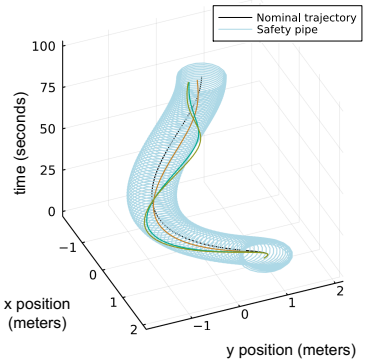


Figure 3: Safety: Deviation from ideal.

Another measure of system-level safety, as outlined in Section I, is the size of the reachable set in the state space. In Figure 4, the system evolves from the initial region (labeled 1 in the figure). This is a 1×1 square in the state space centered around $(10, 10)$. The reachable sets in the state space as time progresses are marked as $2, 3, 4, \ldots, 20$. Since the underlying

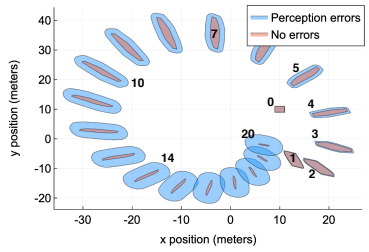


Figure 4: Safety: Reachability.

system (plant + controller) is stable, in the ideal case, with no state estimation uncertainty, the system converges with the passage of time. However, in the presence of state estimation uncertainty, *i.e.*, $|\mathbb{x} - \hat{\mathbb{x}}| \neq 0$, the uncertainty or error accumulates over time and the size of the reachable set might be much larger, as seen in Figure 4. In other words, smaller reachable sets (*e.g.*, when no errors are present) indicate that the states of the system are more predictable; and larger reachable sets (*e.g.*, when there are errors/uncertainties in perception processing) indicate greater uncertainty in the system's evolution.

III. Edge-Cloud ML & Split Computing

In this section we present a brief overview of *split computing* or edge-cloud ML. Over the past decade, DNNs have become highly efficient at solving complex problems, leading to increased deployment on edge devices, referred to as Local-only Computing (LoC). However, DNNs often demand more computing power than resource-limited edge devices can provide, requiring compression techniques that lead to accuracy loss. Consequently, cloud-based Remote-only Computing (RoC) is the more commonly used approach, where data is sent to a server for processing, and inference results are returned to the edge device. While RoC maintains the highest model accuracy, network delays can be large.

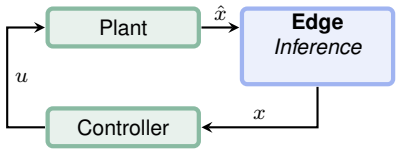
As a compromise between the LoC and the RoC, recently suggested SC frameworks divide a DNN between the edge and the cloud, reducing latency and bandwidth demands. Early SC implementations partitioned models at specific layers without further modification to the models, while recent advances attempt to optimize latency. This section examines these approaches—LoC, RoC, and SC—in detail and explores applications of SC in resource-constrained systems, with particular emphasis on the role of SC in enhancing performance and safety in CPSs.

A. LoC, RoC, SC: A closer look

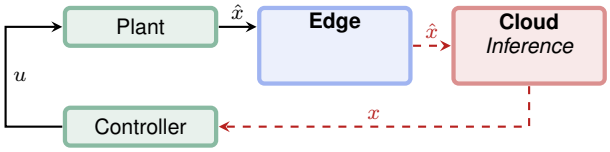
We refer to any architecture LoC, RoC, or SC, that uses a DNN model, as $\mathcal{M}(\cdot)$. It produces the inference output y from an input x.

Local-only Computing (LoC): In LoC, all computations are performed on the edge device, which fully executes $\mathcal{M}(x)$. Its structure is depicted in Figure 5a. This approach minimizes latency due to the proximity of the DNN to the sensor but may not support large DNN models that require more resources. To address this, lightweight models $\bar{\mathcal{M}}(x)$, such as MobileNetV3 [2], use techniques like depth-wise separable convolutions for efficient processing. Recent advances in DNN compression, including network pruning and quantization [3], or knowledge distillation [4], further optimize edge model efficiency, though with potential quality trade-offs.

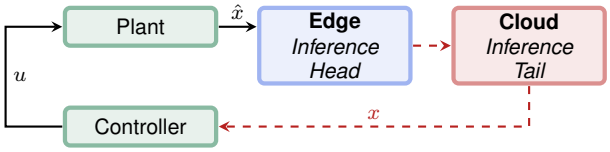
Remote-only Computing (RoC): In RoC, the input x is sent to the cloud where $\mathcal{M}(x)$ is processed. Its structure, along with a plant and a controller, is depicted in Figure 5b. This setup achieves high inference accuracy, thanks to the server's processing power, but incurs higher latency and bandwidth usage due to data transmissions.



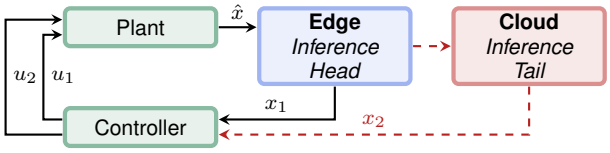
(a) With LoC, the edge device entirely executes the model inference.



(b) With RoC, the input is transferred to the cloud, where it is processed, and the result is then sent back to the edge device.



(c) With SC, the computation is split between edge and cloud. Transmission bandwidth is reduced by using an encoder and decoder to compress data.



(d) When combining SC and early-exit, the computation is split between the edge and the cloud; however, we have intermediate branches, producing an estimate x_1 of the final inference x_2.

Figure 5: Different learning-enabled CPS architectures.

Split Computing (SC): As illustrated in Figure 5c, SC divides the DNN into a "head", processed on the edge device, and a "tail", processed remotely on the cloud. SC combines the advantages of LoC and RoC by reducing the latency and bandwidth needs, often using an auto-encoder for data compression before transmission [5]. In this paradigm, the encoder function $z_l = \mathcal{F}(x)$ runs on the edge device, and a decoder function $\bar{x} = \mathcal{G}(z_l)$ runs on the server, with encoding performance defined by the distance $d(x, \bar{x})$.

Early SC studies like [6] suggested that initial DNN layers are ideal split points for balancing latency and energy use. Advanced methods, such as quantization [7], predefined sparsity [8], and lossy compression [9], can also be combined to further improve performance. For efficiency, lossless encoding techniques like those in [10] and auto-encoder-based compression [11] have also been explored.

Methods for selecting split points have evolved from architecture-based approaches to more refined neuron-based techniques. Architecture-based methods, such as those in [12], identify split points at layers where the network size decreases, assuming that compression at these points is more efficient due to reduced computational complexity. Indeed, neuron-based techniques assess the importance of individual neurons, as shown in [13], [14], by analyzing neuron gradients in relation to decision accuracy. Optimal split points are thus placed after

layers containing highly influential neurons to preserve critical information until then. Advanced learning techniques, such as Multi-Task Learning (MTL), are also being adapted to SC scenarios to allow concurrent task processing and improve model generalization [15]. By incorporating MTL into SC, these designs manage multiple learning tasks simultaneously, shifting the focus from single-task learning and achieving better performances for the main learning task.

B. Implementing Split Computing in Cyber-Physical Systems

We now examine how SC partitions can influence the performance of a feedback controller [16]. Figure 5a presents a basic scenario where the entire DNN is deployed locally on an embedded system. In this case, the limited computational power of the embedded platform constrains the DNN size, affecting its accuracy in classification or estimation tasks. A higher estimation error results in a larger reachable set, compromising system safety.

Figure 5c also shows a typical SC architecture where initial inference occurs on a local embedded platform, while further inference runs on a cloud server. Cloud resources allow for a larger DNN and higher inference accuracy, thereby reducing state estimation error compared to full local "*compressed*" computation. However, this setup introduces additional sensor-to-actuator delay due to communication with the cloud. Data loss may also occur depending on the network protocol, which impacts the reachable set size and, consequently, system safety.

Finally, Figure 5d depicts an early-exit scenario where sufficient inference accuracy on the local DNN triggers a "local exit," allowing the control input u_1 to be computed using a state estimate x_1. This approach reduces latency by minimizing sensor-to-actuator delay. If the local inference accuracy is insufficient, additional computation is conducted on the cloud, and x_2 is additionally used by the controller to compute a second control input u_2.

This paper aims to design a learning-enabled controller suitable for use with a SC architecture. In this regard, it is important to note that the DNN architecture with the highest inference accuracy may not be the best for maximizing safety in SC configurations due to the sensor-to-actuator delay. This observation highlights the need to carefully select split points in SC for control systems design.

C. Example of Split Computing in Autonomous Vehicle

In this example, we explore the application of SC for real-time pedestrian distance estimation within an autonomous vehicle. This system uses a split DNN to analyze visual data and estimate the distance to pedestrians in the vehicle's surroundings. The DNN is divided into two segments, referred to as the "head" and "tail", each running on a separate device. This division enables efficient parallel processing, which helps reduce computation time.

The following paragraphs detail each phase of real-time inference in this split DNN architecture, from image acquisition and data pre-processing, to distance estimation, carried out by the model's head and tail.

***Step 1: Image Acquisition**:* The inference process begins with continuous image acquisition, where the autonomous vehicle's camera captures frames at a consistent rate, typically 30 frames per second, to provide a real-time view of the surroundings. Each captured frame is then preprocessed to meet the neural network's input requirements. This preprocessing step may include resizing or cropping the frame to the necessary dimensions and applying normalization techniques, such as pixel intensity scaling, to standardize the input.

***Step 2: Data Preprocessing**:* In this stage, the image frames undergo preprocessing to prepare them for pedestrian detection and distance estimation. An object detection model, such as YOLO [17], analyzes each frame to identify pedestrians and generate bounding boxes around each detected individual. These bounding boxes specify the location of each pedestrian in the scene and serve as input for the subsequent distance estimation network. After detection, the bounding boxes are transmitted to the head part of the split model, which resides on the same edge device, where they will be used for pedestrian distance estimation.

***Step 3: Head Computation**:* In this step, the head of the split model processes the detected pedestrian regions within the bounding boxes. The bounding box features are fed into this network section, consisting of a series of feature extraction layers designed to capture relevant information for distance estimation. The head network transforms these features into an intermediate representation, encoding essential details about each detected pedestrian. This intermediate representation is used for a quick but uncertain distance estimation and may be used for a fast control action. But it is then transmitted to the tail network for final processing and a more accurate distance estimation.

***Step 4: Tail Computation**:* The tail network completes the distance estimation process upon receiving the intermediate feature representation from the head network. This tail network refines these features using fully connected or specialized layers optimized for regression, mapping the data to precise distance values. The final output is a scalar for each pedestrian, representing the estimated distance, in meters, between the vehicle and each detected individual. This distance information is relayed to the vehicle's navigation and safety system, providing essential data for real-time decision-making and a more refined final control action.

IV. Controller Design for Edge-Cloud Platforms

We finally propose a learning-enabled controller suitable for use with a split computing architecture. In this setup, sensor devices sense the environment, and a DNN uses the output from these sensors to estimate one or more of the state variables modeled by the state space. In addition to the pedestrian example outlined earlier, another example is that of a cruise controller in a autonomous car, where sensors such as camera devices capture raw image data from the environment. A DNN then processes this image data stream to estimate values such as the distance between the ego car and the car in front. The control input (such as the acceleration to be

applied) is then calculated based on this estimated distance as produced by the DNN. Here, the system behavior depends on how accurate is the estimation produced by the NN and how quickly that estimation is available for calculation and, thus, the application of the control input.

Since such control systems need to react quickly to changes in the environment in which they have been deployed, there must be a balance between the computation delays encountered in implementing the controller and the accuracy of its learning-based components. A smaller NN will be able to produce output for a given input quicker than a larger NN at the cost of reduced accuracy of the estimated result. As described in Section I, we, therefore, propose using a SC architecture where a smaller NN is deployed on the edge device, which produces quick estimates but is less accurate than another, larger DNN deployed in the cloud. The controller can then use the early (but inaccurate/uncertain) results to calculate a control input based on the output from the edge DNN while simultaneously querying the cloud DNN. When the more accurate output from the cloud DNN is available, a new control input is calculated and applied.

A. System setup

We have two DNNs: a smaller and less accurate NN_E, deployed on the edge device, and a larger, more accurate NN_C, deployed on the cloud. We assume that the data a sensor, available at the start of a control period, is sent as input to both NN_E and NN_C simultaneously. The output from NN_E is available earlier than that from NN_C because of the larger size of NN_C and the need to transmit the input to the cloud and the results back to the edge device.

The plant receives *three* different control inputs within a single sampling period: (i) the control input calculated on the output from NN_C from the *previous* sampling period is applied until NN_E produces its output using the sample obtained in the *current* sampling period. (ii) The output of this less accurate NN_E is then used to calculate a control input that is applied until (iii) the one from NN_C for the current sampling arrives, which is used for calculating a more accurate control input, and this cycle continues into the next time step. See Figure 6.

To design the controller based on such a system setup, we use the computation delay encountered with the use of the edge DNN, NN_E. This delay, denoted as D_{c_1}, is the time from the instant the input to the NN is available till the instant the control input based on the output from this network is ready. Similarly, D_{c_2} is the equivalent delay associated with the cloud DNN, NN_C. We also use the simplifying assumption that both D_{c_1} and D_{c_2} are less than the sampling period h of the discrete-time controller. If $D_{c_2} > h$, the design of the controller needs to be suitably modified. Again, see Figure 6 for a timeline of the different control inputs within a sampling period. At each sampling point, $t = k-1, k, k+1, \ldots$, the state of the plant is sensed using different sensors as discussed earlier. The control inputs calculated from the estimations provided by NN_E and NN_C are denoted as u_1 and u_2, respectively. The sampling period, h, as well as the delays incurred by NN_E and NN_C are also marked as D_{c_1} and D_{c_2} in Figure 6.

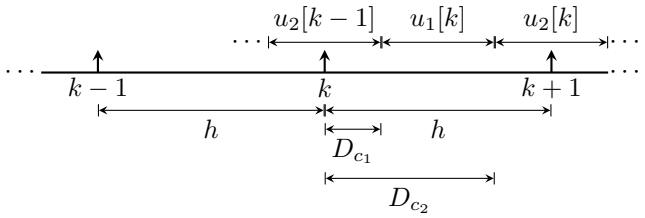


Figure 6: Control inputs (i) $u_2[k-1]$, (ii) $u_1[k]$, and (iii) $u_2[k]$.

B. State-space model

Suppose that the plant has a state vector $x[k] \in \mathbf{R}^n$ and a control input $u[k] \in \mathbf{R}^m$. The discrete states-space representation of the closed-loop system is given by:

$$x[k+1] = Ax[k] + Bu[k]. \tag{4}$$

Now, we have two control inputs $u_1[k]$ and $u_2[k]$, with a sampling period h. As outlined earlier, the applied control inputs within the k to $k+1$-th sampling period (see Figure 6) can be divided into three stages: In the first stage, from time 0 (the start of the k-th sampling period) to D_{c_1}, the system is controlled by $u_2[k-1]$, derived from the states calculated by NN_C in the previous or $k-1$-th period. In the second stage, from time D_{c_1} to D_{c_2}, control is provided by $u_1[k]$, based on the states calculated by NN_E in the k-th sampling period. Finally, in the third stage, from time D_{c_2} to h, the system is controlled by $u_2[k]$, derived from the states calculated by NN_C in the current or k-th period. By reconstructing the model, the state of the plant at the $(k+1)$-th time step is updated as follows:

$$\begin{aligned} x[k+1] = Ax[k] + \Gamma_1(D_{c_1}, D_{c_2})u_1[k] + \Gamma_2(D_{c_2})u_2[k] \\ + \Gamma_3(D_{c_1}, D_{c_2})u_2[k-1], \end{aligned}$$

where $\Gamma_1(D_{c_1}, D_{c_2})$, $\Gamma_2(D_{c_2})$, and $\Gamma_3(D_{c_1}, D_{c_2})$ are given by:

$$\Gamma_1(D_{c_1}, D_{c_2}) = \int_0^{D_{c_2}-D_{c_1}} e^{As} B \, ds,$$

$$\Gamma_2(D_{c_2}) = \int_0^{h-D_{c_2}} e^{As} B \, ds,$$

$$\Gamma_3(D_{c_1}, D_{c_2}) = \int_{h-D_{c_2}}^{h-D_{c_2}+D_{c_1}} e^{As} B \, ds.$$

To transform this into a standard state-space model (as in Equation (4)), we define a new augmented state z by combining the state x with the control input u_2 from the previous period, and a new control input vector u consisting of u_1 and u_2, as follows:

$$z[k] = \begin{bmatrix} x[k] \\ u_2[k-1] \end{bmatrix},$$

$$u[k] = \begin{bmatrix} u_1[k] \\ u_2[k] \end{bmatrix}.$$

The new state-space equation is then represented as:

$$z[k+1] = \Phi z[k] + \Gamma u[k],$$

$$u[k] = Kz[k],$$

where the system matrix Φ and input matrix Γ are defined as:

$$\Phi = \begin{bmatrix} A & \Gamma_3 \\ 0 & 0 \end{bmatrix}, \quad \Gamma = \begin{bmatrix} \Gamma_1 & \Gamma_2 \\ 0 & I \end{bmatrix}.$$

C. Controller Design

We compute the control input by multiplying the state vector $z[k]$ by the gain matrix K, obtained using the Linear–Quadratic Regulator (LQR) method.

$$u[k] = Kz[k].$$

The LQR is a widely used method for designing optimal state-feedback controllers. The goal of the LQR approach is to compute a gain matrix K that minimizes a quadratic cost function, balancing between minimizing system state deviations and control effort. The cost function is defined as:

$$J = \sum_{k=0}^{\infty} \left(z[k]^T Q z[k] + u[k]^T R u[k] \right),$$

where $z[k]$ is the state vector, $u[k]$ is the control input, Q is the state cost matrix that penalizes deviations in the state, and R is the control cost matrix that penalizes control effort. The Q and R matrices can be customized to adjust the balance between the cost of control effort and deviations.

To find the optimal gain matrix K, the LQR method solves the discrete-time algebraic Riccati equation, given by:

$$P = \Phi^T P \Phi - \Phi^T P \Gamma (\Gamma^T P \Gamma + R)^{-1} \Gamma^T P \Phi + Q,$$

where Φ and Γ represent the system dynamics matrices in the states-space model. Solving the above equation yields the matrix P, which is then used to compute K as:

$$K = (\Gamma^T P \Gamma + R)^{-1} \Gamma^T P \Phi.$$

Since $z[k] \in \mathbf{R}^{n+m}$ and $u[k] \in \mathbf{R}^{2m}$, $K \in \mathbf{R}^{2m \times (n+m)}$, its entries are given by the following:

$$K = \begin{bmatrix} k_{x_{11}} & k_{x_{12}} & \cdots & k_{x_{1n}} & k_{u_{11}} & k_{u_{12}} & \cdots & k_{u_{1m}} \\ k_{x_{21}} & k_{x_{22}} & \cdots & k_{x_{2n}} & k_{u_{21}} & k_{u_{22}} & \cdots & k_{u_{2m}} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ k_{x_{(2m)1}} & k_{x_{(2m)2}} & \cdots & k_{x_{(2m)n}} & k_{u_{(2m)1}} & k_{u_{(2m)2}} & \cdots & k_{u_{(2m)m}} \end{bmatrix}$$

We then introduce the uncertainty values of the edge and the cloud DNNs. The uncertainty for NN_E is denoted by λ_{1_i}, and the uncertainty for NN_C is denoted by λ_{2_i}. We then have:

$$K = \begin{bmatrix} k_{x_{11}}\lambda_{1_1} & \cdots & k_{x_{1n}}\lambda_{1_n} & k_{u_{11}} & \cdots & k_{u_{1m}} \\ k_{x_{21}}\lambda_{1_{n+1}} & \cdots & k_{x_{2n}}\lambda_{1_{2n}} & k_{u_{21}} & \cdots & k_{u_{2m}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ k_{x_{(m+1)1}}\lambda_{2_1} & \cdots & k_{x_{(m+1)n}}\lambda_{2_n} & k_{u_{(m+1)1}} & \cdots & k_{u_{(m+1)m}} \\ k_{x_{(m+2)1}}\lambda_{2_{n+1}} & \cdots & k_{x_{(m+2)n}}\lambda_{2_{2n}} & k_{u_{(m+2)1}} & \cdots & k_{u_{(m+2)m}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ k_{x_{(2m)1}}\lambda_{2_{(m-1)n+1}} & \cdots & k_{x_{(2m)n}}\lambda_{2_{mn}} & k_{u_{(2m)1}} & \cdots & k_{u_{(2m)m}} \end{bmatrix}$$

In our new state $z[k]$, the original state of the system $x[k]$ is multiplied by the $k_{x_{ij}}$ entries in the K matrix. As a result, we introduce our uncertainty values only to the $k_{x_{ij}}$ entries, because our objective is solely to simulate the estimation error in the system states.

V. Experimental Results

We examine the effectiveness of the SC architecture by computing the reachable states of the closed-loop system in four different scenarios:

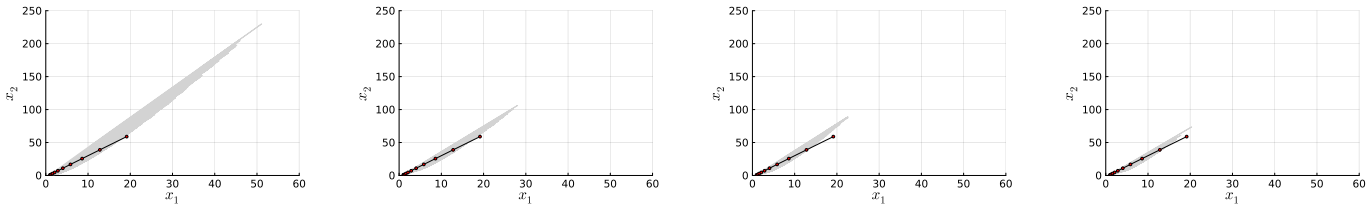
(a) Only edge computation: Only the edge DNN, NN_E, is used for estimating the plant state, which is then used to compute the control input. The edge DNN has a lower inference accuracy but also a lower latency than the cloud DNN.

(b) Only cloud computation: Here, only the cloud DNN, NN_C, is used for estimating the plant state, which is then used to compute the control input. The cloud DNN has a higher accuracy than the edge DNN, but also suffers from increased latency due to having the input and output be transmitted over a network instead of being processed locally on the edge device itself.

(c) Split-computation (*i.e.,* (a) + (b)): This is a naïve combination of scenarios (a) and (b). Both the edge and the cloud DNNs send the processed sensor output as inputs to their respective controllers. Since NN_E has a lower latency than NN_C, the estimation output from NN_E is available earlier than that of NN_C, which is used to compute and apply an initial control input, u_1. When the more accurate output from NN_C becomes available, we compute and apply a control input u_2 for the rest of the sampling period, h. Here, the delays and the accuracies of NN_E and NN_C, are assumed to be the same as those in the two earlier scenarios (a) and (b).

(d) Split-computation with different resource distribution: In contrast to the previous Scenario (c), here we consider the case where both the DNNs are used as before, and hence, have the same inference errors. But the edge device is run slower and the cloud device is run faster. Hence, D_{c_1} is increased and D_{c_2} is decreased compared to the values in Scenario (c). We show that this change or redistribution of delays impacts both, the size of the reachable state space and the magnitude of the maximum control input. This implies that DNN sizing and edge-cloud resource allocation needs to be appropriately done to optimize CPS safety and performance.

For our experiments, we used the state-space model of an F1Tenth [18] racing car. This model has two state variables: x_1 and x_2. The sensor processing for estimating the states x_1 and x_2 used an edge-cloud based ML as outlined earlier. The controller used a sampling period of $h = 0.02$ seconds. The state estimation uncertainty values, λ_1 for the edge DNN NN_E, and λ_2 for the cloud DNN NN_C were sampled from normal distributions. For each of scenarios (a) – (d) listed above, the evolution of the system was studied for 10 iterations. In other words, control inputs with state estimations from



(a) Only edge DNN (b) Only cloud DNN (c) Split-computation: (a) + (b) (d) (a) + (b) with different inference delays

Figure 7: Reachable sets for different edge-cloud partitions for sensor data processing.

NN_E and NN_C were used to perform a reachability analysis that obtained all possible trajectories (evolutions of x_1 and x_2) of the closed-loop system.

The normal distribution describing the uncertainty of the output of the edge DNN was assumed to be $\mathcal{N}_E\ (\mu = 0, \sigma = 0.3)$, and that of the cloud DNN was assumed to be $\mathcal{N}_C\ (\mu = 0, \sigma = 0.2)$. We assumed that both the edge and the cloud DNNs produce correct state estimates on an average, but the standard deviation of the inferences of NN_E are higher than those of NN_C .

Scenario	max. x_1	max. x_2
(a) Only edge DNN	51.10	229.94
(b) Only cloud DNN	28.05	106.28
(c) Split computation: naïve (a) + (b)	22.65	89.06
(d) (a) + (b) with different resource allocation	20.33	73.65
Ideal behavior (zero delay & uncertainty)	19.15	58.96

Table I: Size of reachable state space for different scenarios.

To plot the reachable sets, we sampled $10,000$ random trajectories for each scenario, along with the trajectory resulting in the ideal case when the system states can be inferred accurately, *i.e.,* without any uncertainty, and the sensor-to-actuator delay is zero, *i.e.,* $D_{c_1} = 0$. With the single initial state of $(0, 0)$, the reachable set in this ideal case is a single trajectory in the $(x_1, x_2$ state space. Figure 7 shows the reachable sets obtained under different scenarios, along with the ideal trajectory as a bold line. In Figure 7a, the size of the reachable set (and hence the deviation from the ideal behavior) is the highest since only the edge DNN with the highest inference uncertainty is used. This reachable set reduces to that depicted in Figure 7b because of the increased accuracy of the cloud DNN. Here, the sensor-to-actuator delays of NN_E and NN_C were respectively assumed to be $D_{c_1} = 0.005$ seconds and $D_{c_2} = 0.015$ seconds (the sampling period $h = 0.02$s).

Figure 7c shows the reachable state space for the scenario where an early estimate is obtained from the edge DNN, used to compute the control input u_1, which is applied until the output from the cloud DNN is available for computing and applying control input u_2. We observe that the size of the reachable set, in this case, is even smaller than that in the previous scenarios where only one DNN was used throughout the sampling period (here, D_{c_1} and D_{c_2} were the same as before). This reachable set reduces further in size, and the randomly sampled trajectories become close to the ideal trajectory if the transition to the control input u_2 is done even earlier (*i.e.,* when D_{c_2} is changed from 0.0015 to 0.009), as shown in Figure 7d. Table I shows the boundaries and hence the sizes of the reachable sets for the four different scenarios. It shows that Scenario (d) has the smallest reachable state space, and is therefore best in terms of optimizing system safety.

Scenario	max. u_1
(a) Only edge computation	326.03
(b) Only cloud computation	54.99
(c) Split computation: naïve (a) + (b)	141.27
(d) (a) + (b) with different resource allocation	31.05
Ideal behavior (zero delay & uncertainty)	27.19

Table II: Maximum control input u_1 in different scenarios.

Table II shows the maximum control input u_1 that was necessary in each scenario. The edge DNN only scenario requires the maximum control input/effort. The lower the magnitude of the control input, the better it is, since this translates to the size of the energy source required to power the controller. Using the cloud-only solution requires a lower control input, and also results in a smaller reachable state space. Hence, a cloud-only DNN is strictly better than using the edge DNN alone. Using a combination of edge and cloud DNNs results in a higher maximum control input, but it results in improving system safety, *i.e.,* a smaller reachable state space. However, by appropriately allocating resources to the edge and the cloud, in particular, by reducing the cloud inference latency, the maximum control input can be significantly reduced, while additionally decreasing the size of the reachable state space. This is shown in Scenario (d).

We did not explore the impact of adjusting the sizes of the DNNs in addition to changing the resource allocation. Doing so will further impact the results. But results we already obtained show that edge-cloud based sensor data processing using ML involves a large design space that needs to be explored to be able to optimize system-level CPS performance.

VI. Related Work

We already provided a brief introduction to split computing in Section III. The other line of work closely related to this paper is control/architecture co-design [19]–[21]. The design of control algorithms is typically driven by simplistic assumptions about the implementation architecture. These might include assumptions on delay, numerical accuracy, or

the faithfulness of the translation of controller models to software code. If the realities of a software implementation are not accommodated in the controller model, model-level verification and certification [22]–[24] results no longer carry over to an implementation.

In practice, this is often addressed by progressively refining the controller model with more implementation details while simulating and testing [25] the system after each refinement step [26], [27]. This is to ensure that the introduced refinements continue to satisfy the same specifications used to construct the controller model. However, such simulation and testing is associated with significant overheads.

An alternative is to systematically explore all possible implementation options, derive the timing and other relevant properties associate with each such option, and incorporate these in the design of the controller. Such co-design or co-synthesis [28]–[30] – as opposed to designing the controller and deciding on its implementation choices in isolation – ensures that the safety and performance of the controller is preserved in its implementation. Controller implementation choices like task mapping, task scheduling, and communication message scheduling have been studied in [31]–[35]. Based on such implementation choices, methods to synthesize controllers have been explored in [36]–[38].

The notion of system safety used in this paper is not entirely new. The idea of allowing a safety pipe around an ideal behavior was studied in [39]–[43]. This notion was exploited to allow a scheduler to miss the deadlines of control tasks – thereby resulting in the trajectory of the closed-loop system to deviate from its nominal trajectory – but only to an extent that ensured the behavior of the system to be constrained within the safety pipe.

The verification problem of checking whether a schedule is safe for a given set of controllers was addressed in [44], [45]. The other notion of system safety, *viz.,* the size of the reachable state space was used in [46] to identify optimal sizing of DNNs on a shared graphics processing unit (GPU). Associating DNN inference uncertainty to safe control of autonomous systems, but in a different setup than what we addressed in this paper, has also been studied [47].

While we considered inference delay and uncertainty in sensor data processing, communication with the cloud over a wireless medium also results in potential data loss, which needs to be studied as a part of future work. The domain of *networked control systems* [48]–[51] studies this problem of designing control strategies that are robust to data loss over a wireless network.

Similarly timing analysis of both wired and wireless communication protocols [52]–[54] to design timing-aware controllers is broadly related to our work in this paper. While we assumed in this paper that computation and communication delays are given, there are several techniques [55], [56] to compute them in different setups, including where streaming data is sent [57], [58]. Similarly, scheduling to meet specified timing constraints in the context of CPS design has also been extensively studied [59]–[63].

VII. Concluding Remarks and Future Directions

In this paper we have explored the problem of controller design in an edge-cloud split computing framework for ML-enabled CPS. Our experiments show that by appropriately splitting a DNN for perception processing between the edge and the cloud, we can make use of the early (but uncertain) state estimate from the edge DNN, followed by a late (but accurate) state estimate from the cloud DNN. Such a controller exhibits a smaller reachable state space (and is therefore more safe) compared to other a edge-only or a cloud-only DNN. By appropriately dimensioning the edge and cloud DNNs and resources, the size of the reachable state space and the magnitude of the necessary control input may be minimized. How should this optimization problem be formulated and solved is a part of future work. Finally, our problem formulation should also be extended to incorporate cloud DNN inference latencies that are more than one sampling period long.

Acknowledgments

This work was supported by the US NSF grant 2038960 and the Marie Sklodowska-Curie grant No. 101109243. Zhu and Ganguli made equal contributions to this paper.

References

[1] S. Chakraborty *et al.*, "Automotive cyber-physical systems: A tutorial introduction," *IEEE Des. Test*, vol. 33, no. 4, pp. 92–108, 2016.

[2] A. Howard *et al.*, "Searching for MobileNetV3," in *International Conference on Computer Vision (ICCV)*, 2019.

[3] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and Quantization for Deep Neural Network Acceleration: A Survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.

[4] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge Distillation: A Survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.

[5] Y. Matsubara *et al.*, "Distilled Split Deep Neural Networks for Edge-Assisted Real-Time Systems," in *Workshop on Hot Topics in Video Analytics and Intelligent Edges at Mobicom*, 2019.

[6] Y. Kang *et al.*, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," *SIGPLAN Notices*, vol. 52, no. 4, 2017.

[7] G. Li *et al.*, "Auto-tuning Neural Network Quantization Framework for Collaborative Inference Between the Cloud and Edge," in *Artificial Neural Networks and Machine Learning (ICANN)*. Springer, 2018.

[8] L. Capogrosso *et al.*, "Enhancing Split Computing and Early Exit Applications through Predefined Sparsity," in *Forum on Specification & Design Languages (FDL)*. IEEE, 2024, pp. 1–8.

[9] H. Choi and I. V. Bajić, "Deep Feature Compression for Collaborative Object Detection," in *25th International Conference on Image Processing (ICIP)*. IEEE, 2018.

[10] D. Carra and G. Neglia, "DNN Split Computing: Quantization and Run-Length Coding are Enough," in *Global Communications Conference (GLOBECOM)*. IEEE, 2023.

[11] Y. Matsubara *et al.*, "BottleFit: Learning Compressed Representations in Deep Neural Networks for Effective and Efficient Split Computing," in *International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2022.

[12] M. Sbai, M. R. U. Saputra, N. Trigoni, and A. Markham, "Cut, Distil and Encode (CDE): Split Cloud-Edge Deep Inference," in *International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2021.

[13] F. Cunico *et al.*, "I-SPLIT: Deep Network Interpretability for Split Computing," in *International Conference on Pattern Recognition (ICPR)*. IEEE, 2022.

[14] L. Capogrosso *et al.*, "Split-Et-Impera: A Framework for the Design of Distributed Deep Learning Applications," in *International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2023.

[15] L. Capogrosso, E. Fraccaroli, S. Chakraborty, F. Fummi, and M. Cristani, "MTL-Split: Multi-Task Learning for Edge Devices using Split Computing," in *Design Automation Conference (DAC)*, 2024.

[16] L. Capogrosso *et al.*, "Learning-enabled CPS for edge-cloud computing," in *14th International Symposium on Industrial Embedded Systems (SIES)*, 2024.

[17] G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics YOLO," https://github.com/ultralytics/ultralytics, accessed: Sep. 16 2024.

[18] M. O'Kelly, H. Zheng, D. Karthik, and R. Mangharam, "F1TENTH: An Open-source Evaluation Environment for Continuous Control and Reinforcement Learning," in *Proceedings of Machine Learning Research (PMLR)*, vol. 123. PMLR, 2020, pp. 77–89.

[19] D. Roy *et al.*, "Multi-objective co-optimization of FlexRay-based distributed control systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[20] D. Goswami, R. Schneider, and S. Chakraborty, "Co-design of cyber-physical systems via controllers with flexible delay constraints," in *16th Asia South Pacific Design Automation Conference (ASP-DAC)*, 2011.

[21] D. Roy *et al.*, "Semantics-preserving cosynthesis of cyber-physical systems," *Proc. IEEE*, 2018.

[22] P. Kumar *et al.*, "A hybrid approach to cyber-physical systems verification," in *49th Annual Design Automation Conference (DAC)*, 2012.

[23] G. Georgakos *et al.*, "Reliability challenges for electric vehicles: from devices to architecture and systems software," in *50th Annual Design Automation Conference (DAC)*, 2013.

[24] W. Chang and S. Chakraborty, "Resource-aware automotive control systems design: A cyber-physical systems approach," *Found. Trends Electron. Des. Autom.*, vol. 10, no. 4, pp. 249–369, 2016.

[25] M. Broy *et al.*, "Cross-layer analysis, testing and verification of automotive control software," in *11th International Conference on Embedded Software, (EMSOFT)*, 2011.

[26] G. Tibba *et al.*, "Testing automotive embedded systems under X-in-the-loop setups," in *35th International Conference on Computer-Aided Design (ICCAD)*, 2016.

[27] J. Oetjens *et al.*, "Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges," in *The 51st Annual Design Automation Conference (DAC)*. ACM, 2014.

[28] L. Zhang *et al.*, "Task- and network-level schedule co-synthesis of Ethernet-based time-triggered systems," in *19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.

[29] D. Roy *et al.*, "Tool integration for automated synthesis of distributed embedded controllers," *ACM Trans. Cyber Phys. Syst.*, 2022.

[30] R. Schneider *et al.*, "Constraint-driven synthesis and tool-support for Flexray-based automotive control systems," in *9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.

[31] F. Sagstetter *et al.*, "Schedule integration framework for time-triggered automotive architectures," in *51st Annual Design Automation Conference (DAC)*, 2014.

[32] D. Roy *et al.*, "Goodspread: Criticality-aware static scheduling of CPS with multi-qos resources," in *41st IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[33] S. Chakraborty and L. Thiele, "A new task model for streaming applications and its schedulability analysis," in *Design, Automation and Test in Europe Conference and Exposition (DATE)*, 2005.

[34] M. Lukasiewycz *et al.*, "Modular scheduling of distributed heterogeneous time-triggered automotive systems," in *17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.

[35] H. Voit *et al.*, "Optimizing hierarchical schedules for improved control performance," in *IEEE Fifth International Symposium on Industrial Embedded Systems (SIES)*, 2010.

[36] D. Goswami, R. Schneider, and S. Chakraborty, "Re-engineering cyber-physical control applications for hybrid communication protocols," in *Design, Automation and Test in Europe (DATE)*, 2011.

[37] W. Chang *et al.*, "OS-aware automotive controller design using non-uniform sampling," *ACM Trans. Cyber Phys. Syst.*, vol. 2, no. 4, pp. 26:1–26:22, 2018.

[38] D. Roy *et al.*, "Tighter dimensioning of heterogeneous multi-resource autonomous CPS with control performance guarantees," in *56th Annual Design Automation Conference (DAC)*, 2019.

[39] C. Hobbs *et al.*, "Safety analysis of embedded controllers under implementation platform timing uncertainties," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4016–4027, 2022.

[40] B. Ghosh *et al.*, "Statistical verification of autonomous system controllers under timing uncertainties," *Real Time Syst.*, vol. 60, no. 1, pp. 108–149, 2024.

[41] S. Xu *et al.*, "Safety-aware implementation of control tasks via scheduling with period boosting and compressing," in *29th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2023.

[42] S. Xu *et al.*, "Safety-aware flexible schedule synthesis for cyber-physical systems using weakly-hard constraints," in *28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2023.

[43] C. Hobbs *et al.*, "Quantitative safety-driven co-synthesis of cyber-physical system implementations," in *15th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2024.

[44] A. Yeolekar *et al.*, "Checking scheduling-induced violations of control safety properties," in *20th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, ser. Lecture Notes in Computer Science. Springer, 2022.

[45] A. Yeolekar, R. Metta, and S. Chakraborty, "SMT-based control safety property checking in cyber-physical systems under timing uncertainties," in *37th International Conference on VLSI Design and 23rd International Conference on Embedded Systems (VLSID)*, 2024.

[46] S. Xu *et al.*, "Neural architecture sizing for autonomous systems," in *15th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2024.

[47] Z. Wang, C. Huang, Y. Wang, C. Hobbs, S. Chakraborty, and Q. Zhu, "Bounding perception neural network uncertainty for safe control of autonomous systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

[48] S. Ghosh *et al.*, "Proactive feedback for networked CPS," in *36th ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2021.

[49] M. Balszun *et al.*, "Effectively utilizing elastic resources in networked control systems," in *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

[50] D. Goswami *et al.*, "Characterizing feedback signal drop patterns in formal verification of networked control systems," in *IEEE International Symposium on Computer-Aided Control System Design (CACSD)*, 2013.

[51] A. M. Annaswamy *et al.*, "Arbitrated network control systems: A co-design of control and platform for cyber-physical systems," in *Control of Cyber-Physical Systems*, ser. Lecture Notes in Control and Information Sciences, D. C. Tarraf, Ed., vol. 449. Springer, 2013, pp. 339–356.

[52] P. H. Kindt *et al.*, "Energy modeling for the Bluetooth low energy protocol," *ACM Trans. Embed. Comput. Syst.*, 2020.

[53] P. H. Kindt *et al.*, "Neighbor discovery latency in BLE-like protocols," *IEEE Trans. Mob. Comput.*, vol. 17, no. 3, pp. 617–631, 2018.

[54] P. H. Kindt *et al.*, "Optimizing BLE-like neighbor discovery," *IEEE Trans. Mob. Comput.*, vol. 21, no. 5, pp. 1779–1797, 2022.

[55] E. Fraccaroli *et al.*, "Timing predictability for SOME/IP-based service-oriented automotive in-vehicle networks," in *Design, Automation & Test in Europe Conference (DATE)*, 2023.

[56] D. Roy *et al.*, "Timing debugging for cyber-physical systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

[57] A. Maxiaguine *et al.*, "Rate analysis for streaming applications with on-chip buffer constraints," in *Asia South Pacific Design Automation Conference (ASP-DAC)*, 2004.

[58] Y. Liu, S. Chakraborty, and R. Marculescu, "Generalized rate analysis for media-processing platforms," in *12th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2006.

[59] M. Lukasiewycz, S. Chakraborty, and P. Milbredt, "FlexRay switch scheduling - A networking concept for electric vehicles," in *Design, Automation and Test in Europe (DATE)*, 2011.

[60] R. Schneider *et al.*, "Optimized schedule synthesis under real-time constraints for the dynamic segment of FlexRay," in *IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing (EUC)*, 2010.

[61] P. Mundhenk *et al.*, "Policy-based message scheduling using FlexRay," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2014.

[62] R. Schneider *et al.*, "Quantifying notions of extensibility in FlexRay schedule synthesis," *ACM Trans. Design Autom. Electr. Syst.*, vol. 19, no. 4, pp. 32:1–32:37, 2014.

[63] F. Sagstetter, M. Lukasiewycz, and S. Chakraborty, "Generalized asynchronous time-triggered scheduling for FlexRay," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 36, no. 2, pp. 214–226, 2017.