Co-designing Perception-based Autonomous Systems on CPU-GPU platforms

Suraj Singh, Ashiqur Rahaman Molla, Arijit Mondal, *Member, IEEE,*, Soumyajit Dey, *Senior Member, IEEE*

***Abstract*—Perception-based autonomous system design methods are widely adopted in various domains like transportation, industrial robotics, etc. However, attaining safe and predictable execution in such systems depends on the platform-level integration of perception and control tasks. This paper presents a novel methodology to co-optimize these tasks, assuming a CPU-GPU-based real-time platform, a common choice of compute resource in this domain. Unlike traditional methods that separately address AI-based sensing and control concerns, we consider that the overall performance of the system depends on the inferencing accuracy of perception tasks and the performance of the control tasks iteratively executing in a feedback loop. We propose a design-space exploration methodology that considers the above concern and validates the same on an autonomous driving use case using a novel simulation setup.**

***Index Terms*—Control Scheduling Co-Design, CPU-GPU Optimizations, Object Detection, Model Predictive Control, Autonomous Driving.**

I. Introduction

Design and deployment of controllers for autonomous Cyber-Physical Systems (CPS) activated by perception inputs from the environment has been greatly aided by Artificial Intelligence (AI) enabled computer vision. Such systems typically involve the usage of complex inference pipelines that fuse the periodically sampled data from various sensors. The pipelines infer environmental states and pass the information to decision systems for real-time control and actuation. For example, emerging technologies like (semi-autonomous vehicles that employ real-time object detection pipelines for various image-based control applications like Autonomous Emergency Braking (AEB), Lane Keep Assist System (LKAS), etc. Note that the corresponding control tasks need to wait for the AI pipeline to provide output while also requiring to satisfy the end-to-end sense-to-actuation delay of each control loop. Even in the weakly-hard paradigm of control task execution and scheduling [1], deadline misses beyond the minimum requirement may hamper the safety and stability of the system.

From the platform side, the usual practice has been to execute AI pipelines for perception systems on accelerators like Graphics Processing Units (GPUs). These are usually provided in modern heterogeneous System-on-Chips (SoCs) along with the CPU that runs the host firmware with control-intensive functionalities. Let us consider a task allocation scenario for an AI-based perception pipeline (e.g. YOLO [2]) and a control task (MPC) on such a platform, as shown in Fig. 1. We build our methodology around this specific use case, though the system model assumed will remain applicable in more generic settings.

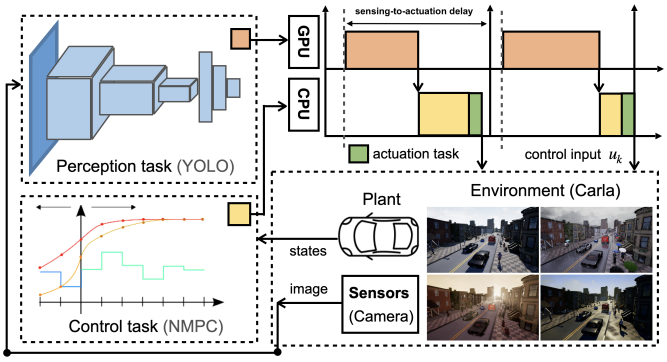

Fig. 1. Perception and control task allocation in CPU-GPU systems.

To illustrate the overall interdependency of object detection performance with adaptive control strategies, we set up several driving scenarios in CARLA [3] simulator where the ego vehicle navigates through a dynamic environment with various challenging settings. The widely adopted object detection pipeline YOLO [2] processes visual data in real-time for identifying and classifying objects. A Model Predictive Control (MPC) based periodic controller task uses this information to generate optimal control commands. Note in Fig. 1 that the MPC task instances on the CPU can only start when the corresponding GPU-based object detection task instances finish execution. If all the stages of the detection pipeline are executed, the detection inferences happen with higher *confidence score* but leave less time for the control task to execute. This may lead to a deadline miss scenario. However, suppose the detection pipeline architecture uses a preemptable model following [4]. In that case, it may be possible to attain early inferencing and provide the MPC task with a sufficiently large execution window. Such flexible task mappings for perception and control tasks is next described in the system model.

System Model and Contributions

We consider the perception task instances in Fig. 1, running on a GPU, to be a YOLO detection pipeline, implemented in a *preemptable* [5] manner, allowing to skip complete execution

Manuscript accepted June 30, 2024.

Suraj Singh and Soumyajit Dey are with the Department of Computer Science and Engineering, and Ashiqur Rahaman Molla is with the Center of Excellence in Artificial Intelligence at the Indian Institute of Technology Kharagpur, Kharagpur, India (email: surajsingh11@kgpian.iitkgp.ac.in, soumya@cse.iitkgp.ac.in, ashiqur.rahaman@kgpian.iitkgp.ac.in)
Arijit Mondal is with the Indian Institute of Technology Patna, Bihar, India.
The authors acknowledge the generous grants received from "AI4ICPS I-Hub Foundation - IIT Kharagpur" for partially supporting this work.

and sample the output from an intermediate depth layer. A generic perception pipeline comprises a sequence of 2-D convolution layers followed by max-pooling, where the final output is derived from a series of fully connected (FCN) layers. In a preemptable architecture (as shown in Fig. 2), the inference results from an intermediate convolution-pooling operation can be transformed by *flattening*, then passing through specific FCN layers to preserve the original output dimensions.

However, for ensuring detection-aided safe control, the choice of preemption stages should start from an intermediate depth layer l whose output satisfies a threshold confidence score $\mathcal{CS}_{\text{min}}$. It is to be noted that the later the stage, higher the confidence score associated with the detection but the larger the execution time (due to the large number of parameters in a deeper network architecture). In Sec. II, we choose this initial preemptive stage via Algorithm 1 and formally define the selection of later depth stages.

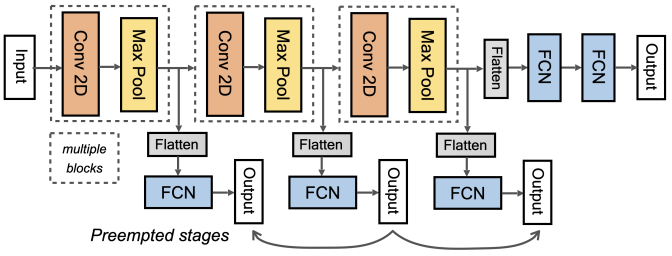

Fig. 2. Overview of stage preemption of perception pipeline

Using the sensory camera frames, the YOLO pipeline running on the GPU detects the class of objects present in the image and predicts the corresponding bounding box coordinates. The output of the detection system is fed to an MPC task running on the CPU. The MPC objective is to keep the vehicle within the lane boundary with minimal deviation while *safely* driving on a given trajectory to the destination. The required state information, namely the vehicle's position, orientation, and velocity, is estimated from the onboard sensors. Based on the current state and a given desired reference state, the MPC control task solves an online quadratic cost function $\mathcal{J}$, as defined in Eq. 1, over some time horizon length H (i.e., the timesteps $[k, k+H]$) and generates a sequence of control inputs $\mathcal{U}_k, \ldots, \mathcal{U}_{k+H-1}$. The system actuation uses $\mathcal{U}_k$ and the problem is again solved with updated states for the next time horizon. The cost function $\mathcal{J}$, balancing state ($\mathcal{X}_k$) and control input ($\mathcal{U}_k$) deviations from their references ($\mathcal{X}_k^{\text{ref}}, \mathcal{U}_k^{\text{ref}}$), can be given as

$$\min_{\mathcal{U}_k \ldots \mathcal{U}_{k+H-1}} \quad \mathcal{J} = \sum_{t=k}^{k+N-1} \big[(\mathcal{X}_k - \mathcal{X}_k^{\text{ref}})^T \mathcal{Q} (\mathcal{X}_k - \mathcal{X}_k^{\text{ref}}) + (\mathcal{U}_k - \mathcal{U}_k^{\text{ref}})^T \mathcal{R} (\mathcal{U}_k - \mathcal{U}_k^{\text{ref}})\big] + \mathcal{X}_N^T \mathcal{P} \mathcal{X}_N \quad (1)$$

$$\text{subject to } \mathcal{X}_0 = \mathcal{X}_{\text{init}}$$

$$\mathcal{X}_{k+1} = \mathcal{A}_k \mathcal{X}_k + \mathcal{B}_k \mathcal{U}_k, \;\; k = 0, 1, \ldots, N-1$$

$$\mathcal{X}_{\text{min}} \leq \mathcal{X}_k \leq \mathcal{X}_{\text{max}}, \quad k = 0, 1, \ldots, N$$

$$\mathcal{U}_{\text{min}} \leq \mathcal{U}_k \leq \mathcal{U}_{\text{max}}, \quad k = 0, 1, \ldots, N-1$$

where $\mathcal{U}$ is the control input vector, $\mathcal{X}$ is the state vector, $\mathcal{X}^{\text{ref}}$ and $\mathcal{U}^{\text{ref}}$ are the reference state and control input vectors, $\mathcal{Q}$ and $\mathcal{R}$ are the state and control input weight matrices, $\mathcal{P}$ is the terminal weight matrix, $\mathcal{A}_k$ and $\mathcal{B}_k$, define system dynamics at each timestep, $[\mathcal{X}_{\text{min}}, \mathcal{X}_{\text{max}}]$, $[\mathcal{U}_{\text{min}}, \mathcal{U}_{\text{max}}]$ are the bounds on the state and control input respectively, and $\mathcal{X}_{\text{init}}$ is the initial state. At each timestep, the MPC objective is to minimize the lateral deviation from the centerline of the road lanes, i.e *cross-track deviation error* (CTDE), and track a reference set velocity with minimum actuation effort while ensuring that the vehicle safely reaches the terminal waypoint in a given driving scenario.

We formally define a driving scenario $\mathcal{DS}$ as a 3-tuple $\{\mathcal{S}, \mathcal{W}, \mathcal{V}\}$ for a given scene $\mathcal{S}$, set of predefined waypoints $\mathcal{W}$, and a reference tracking velocity $\mathcal{V}$. For performance evaluation, we calculate *mean square error* (MSE) on CTDEs across the entire trajectory $\mathcal{T}$ over a window size $\mathcal{B}$ at timestep k as,

$$MSE(k) = \frac{1}{\mathcal{B}} \sum_{k'=k}^{k+\mathcal{B}} [CTDE(k')]^2, \; \forall \; k \; \leq \text{length}(\mathcal{T}) - \mathcal{B}.$$

It is to be noted that the execution time of the MPC increases with horizon size due to the need to optimize a larger set of constraints. However, larger horizon choices improve control quality. Given a periodic control loop with sampling period h and a bounded sense-to-actuation delay $\tau \leq h$, the total execution time of the YOLO on GPU and MPC on CPU must be within τ while satisfying the precedence constraint between perception and control for each task instance. In summary, our major contributions are as follows.

1) Given a driving scenario $\mathcal{DS}$, the joint parameter space of perception and control tasks, and a target CPU-GPU platform, we provide a methodology that reports suitable task parameter choices for attaining the least MSE.
2) We validate our methodology on an experimental testbed for various driving scenarios and report encouraging results.

II. Methodology

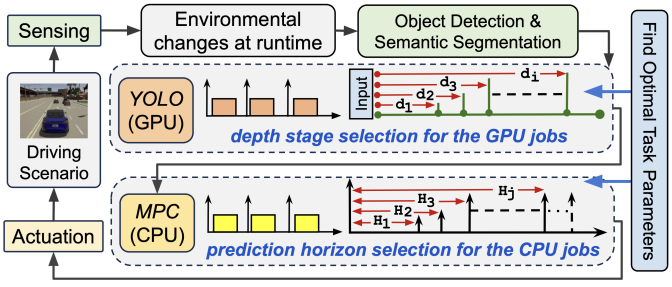

Fig. 3. Overall flowchart for finding suitable task configurations

A. Configurations for Multi-Stage Perception Pipeline

Let the depth layers of perception CNN architecture be indexed as $1, 2 \ldots, L$ where L-th layer denotes the final output. Then all the stages up to depth i can be defined as $d_i \in \{d_1 \leq d_2, \leq \ldots \leq d_L\}$. Let the associated confidence score for a depth-d perception task's prediction

at k-th timestep be defined by $\mathcal{CS}^d(k)$. As discussed earlier (Sec. I), the confidence score of the initial preemptive stage should be $\geq \mathcal{CS}_{min}$. Hence, the initial preemptive layer l defined by depth d_l is that lowest possible layer for which $\mathcal{CS}^{d_l}(k) \geq \mathcal{CS}_{min}\ \forall k \geq 0$.

Given a predefined driving scenario $\mathcal{DS}$, Algorithm 1 identifies d_l for the given pipeline using Binary Search over the choices $\{d_{low} = d_1, \cdots, d_{high} = d_L\}$. For each current choice of $d_l = (d_{low} + d_{high})/2$, it evaluates the confidence scores for K timesteps (by running a nominal MPC with prediction horizon H and YOLO preempted at depth d_l using `calc_confidence`. The algorithm compares the lowest confidence score in the entire run with CS_{min} and updates search limits to find d_l efficiently. Accordingly, we select a set of stage depths $d^s = \{d_i \,|\, i \in [l, L]\}$. The YOLO inference timings for each stage depth d_i are obtained through multiple profiling runs in closed loop simulation and are denoted by $e_Y^{d_i}$. Executing deeper stages yields better confidence scores but higher execution time values.

Algorithm 1 Find CNN preemption stages

Require: $\mathcal{CS}_{\text{min}}$, $\{d_1, d_2, \ldots d_L\}$, $\mathcal{DS}$, H, and K
1: $d_{low} \leftarrow d_1$
2: $d_{high} \leftarrow d_L$
3: **while** $(d_{low} \leq d_{high})$ **do**
4: $\quad d_l = (d_{low} + d_{high})/2$
5: $\quad$ **for** timestep $k \leftarrow 1$ to K **do**
6: $\quad\quad \mathcal{CS}^{d_l}(k) \leftarrow$ `calc_confidence`$(\mathcal{DS}, H, d_l)$
7: $\quad\quad score \leftarrow min(\mathcal{CS}^{d_l}(k), \mathcal{CS}^{d_l}(k-1))$
8: $\quad$ **end for**
9: $\quad$ **if** $(score \geq \mathcal{CS}_{min})$ **then**
10: $\quad\quad d_{low} = d_l$
11: $\quad$ **else**
12: $\quad\quad d_{high} = d_l - 1$
13: $\quad$ **end if**
14: **end while**
15: **return** d_l

B. Platform-Aware Design Space Exploration

As shown in Fig. 3, the CPU task (MPC) starts only after it receives updated inputs from the GPU task (YOLO). As previously defined in Sec. I, it is required for the tasks to satisfy the overall deadline requirement in every sampling period (i.e. the combined execution time must be $\leq \tau$). Let the MPC task running on CPU have N prediction horizons as available options, i.e. H_j for $j \in \{1, 2, \ldots, N\}$ to determine the optimal control actuation values. The higher the prediction horizon, the better the quality of control, but the larger the computation cost, thus, the execution time. Let $e_M^{H_j}$ be the computation time required when the prediction horizon is H_j. For a given platform, $e_M^{H_j}$ is computed through multiple profiling runs in closed-loop simulation. Given these profiled timing executions, our framework uses Algorithm 2 (as depicted in Fig. 3) to select a suitable pipeline stage depth d_i for the GPU task, and a corresponding *admissible* MPC prediction horizon for the CPU task such that overall timing constraint is satisfied and the MSE is minimized for a given *driving scenario* $\mathcal{DS}$.

Algorithm 2 considers as input the following: (i) a set of preemptable pipeline depth choices $d^s = \{d_i \,|\, i \in [\, l, L\,]\}$, (ii) their inference computation times $e_Y^s = \{e_Y^{d_i} \mid d_i \in d^s\}$ (iii) a set of N selected MPC prediction horizons $H^s = \{H_j \mid j \in 1, 2, \ldots, N\}$, (iv) their task execution times $e_M^s = \{e_M^{H_j} \mid H_j \in H^s\}$, (v) sensing-to-actuation delay τ, (vi) a maximum allowable MSE for deviation MSE_{max}, (vii) driving scenario $\mathcal{DS}$.

For each choice of pipeline depth ($i \in [l, L]$) (line 2), the algorithm creates a $list$ of *feasible* (w.r.t. timing) task configurations $[d_i, H_j]$ (lines 7-10). For each element in such a list of configurations, the simulation loop computes MSE values over scenario $\mathcal{DS}$ as long as all the waypoints in $\mathcal{W}$ are not covered and the maximum observer MSE is stored. If this value is less than MSE_{max}, the configuration is considered *admissible* and stored in $configs$ (lines 12-23). Finally, the configuration with minimum MSE is returned.

Algorithm 2 Exploring Configurations for Perception and Control Tasks

Require: d^s, e_Y^s, H^s, e_M^s, τ, MSE_{max} and $\mathcal{DS}$
1: $configs \leftarrow [\,]$
2: **for** $i \leftarrow L$ down to l **do**
3: $\quad$ **if** $d_i \notin d^s$ **then**
4: $\quad\quad$ continue
5: $\quad$ **end if**
6: $\quad list \leftarrow [\,]$
7: $\quad$ **for** $j \leftarrow 1$ to N **do**
8: $\quad\quad$ **if** $e_Y^{d_i} + e_M^{H_j} \leq \tau$ **then**
9: $\quad\quad\quad list$`.append(`$[\, d_i, H_j\,]$`)`
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: $\quad$ **for** each $\mathcal{C} := < d_i, H_j > \in list$ **do**
13: $\quad\quad k \leftarrow 0$
14: $\quad\quad$ **while** ($\mathcal{W}$ not covered) **do**
15: $\quad\quad\quad MSE(k) \leftarrow$ `evaluate_MSE`$(\mathcal{C}, \mathcal{DS})$
16: $\quad\quad\quad mse \leftarrow max(MSE(k), MSE(k-1))$
17: $\quad\quad\quad k \leftarrow k + 1$
18: $\quad\quad$ **end while**
19: $\quad\quad$ **if** $mse < MSE_{max}$ **then**
20: $\quad\quad\quad configs$`.append(`$[\, mse, \mathcal{C}\,]$`)`
21: $\quad\quad$ **end if**
22: $\quad$ **end for**
23: **end for**
24: `sort_ascending(`$configs$`,key='mse')`
25: **return** `head(`$configs$`)`

III. Experimental Setup and Results

We conduct our experiments on CARLA [3], a state-of-the-art simulator for self-driving vehicle research. However, for in-the-loop simulation of driving scenarios with an actual embedded platform running CPU-GPU tasks, we execute the corresponding compute tasks on a NVIDIA Jetson Xavier NX board, which features a 6-core ARM v8.2 64-bit CPU, a 384-core NVIDIA Volta GPU with 48 Tensor Cores, and 8 GB LPDDR4x RAM, running Ubuntu 20.04. We run the CARLA

simulator server on a desktop with Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, 16 GB RAM, and NVIDIA GeForce RTX 2070 Super GPU. We adjust the CARLA server settings to run the simulator in an interrupt-driven manner based on the commands of a client-side script running in the embedded platform. The communication between the simulator on the desktop and the compute task on Jetson is set using the widely known Robot Operating System (ROS) framework. The pre-empted perception tasks are specifically modified architecture of pre-trained *ultralytics* YOLOv8 model, accelerated through *TensorRT* framework. We implement the following sequence of executions in the aforementioned setup.

1) The simulator server script waits for a tick to be passed by the client script running on Jetson. Only upon receiving the tick, the server computes and updates the next snapshot of the simulator.
2) The (updated) observations (i.e. vehicle states and camera frames) are transmitted from the server to the Jetson client through ROS publisher nodes at a fixed rate. The required information is received from respective tasks running on Jetson through ROS subscriber nodes.
3) The YOLO task (with specified d_i) runs on Jetson GPU to perform inference on the received image and updates the buffer with its results. The MPC task (with specified H_j) then executes to compute the optimal control input.
4) The Jetson client sends the control command, simulation tick, and time spent information back to the server so that the server knows up to what time the simulation should progress with the older control input followed by the latest computed input.

We consider three driving scenarios $\mathcal{DS}$ based on the scene $\mathcal{S}$ selected from the set $\{$*'day'*, *'night'*, *'rainy'*$\}$. The number of waypoints $\mathcal{W}$ and the reference tracking velocity $\mathcal{V}$ are set to 60 and 10 m/s respectively. We specify the parameter ranges for d_i and H_j within $[d_l = 168, d_L = 268]$ and $[H_1 = 4, H_N = 20]$ respectively, after l is determined by Algorithm 1. Then we profile the execution times $e_Y^{d_i}$ and $e_M^{H_j}$ over several simulation runs. We consider the time delay for a frame (read by the vehicle camera in CARLA) to reach the YOLO task running on Jetson through ROS communication nodes to be negligible. Finally, we evaluate Algorithm 2 to get a set of best possible configurations.

We consider two baselines: (A) YOLO with preemption depth d_L and MPC with prediction horizon H_1, and (B) YOLO with preemption depth d_l and MPC with prediction horizon H_N. Fig. 4 illustrates the experimental results for three driving scenarios: *day*, *night*, and *rainy*. The top row provides a snapshot of the ego vehicle's point of view in the simulator for each scenario, annotating the predicted bounding boxes from the YOLO task running on Jetson's GPU. The subsequent row shows the vehicle trajectories and velocity profiles within road lanes. The bottom row plots MSE values computed over a window length 10 (parameter $\mathcal{B}$) for the entire run in each setting. Comparing MSE of baseline configurations, Baseline **A** performs poorly with a very high MSE since it has the lowest chosen prediction horizon H_1. Baseline **B** performs well on the *day* scenario due to sufficiently visible features captured in the perception task. However, in scenarios *night* and *rainy*, given that the inference takes place at the lowest depth d_l, the YOLO pipeline fails to detect certain features (like lane markings, bounding boxes etc.) that lead to poor driving guidance by MPC. This also results in large MSE. On the other hand, our methodology of executing Algorithm 1 followed by Algorithm 2 for all the three driving scenarios, leads to choice of suitable task parameters that offer better perception and control performance. The resulting parameter configurations, referred as *adaptive choice* in Fig. 4, yields the least MSE in all three cases.

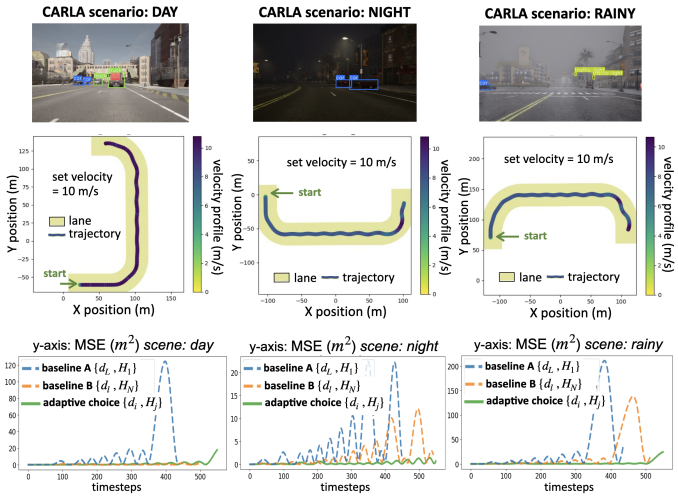

Fig. 4. MSE for various driving scenarios with CARLA snapshots.

IV. Conclusion

The present work uses a novel testbed setup to demonstrate a basic offline method, establishing the benefits of resource-aware co-designing of perception and control for autonomous CPS. For a realistic and deployable solution, there needs to be several possible future extensions like 1) considerations for other controller design paradigms, and 2) incorporating delay awareness and dynamic task mappings for sudden changes in driving scenarios. Another option can also be to consider various perception pipeline choices instead of one pipeline with preemption support.

References

[1] N. Vreman, P. Pazzaglia, V. Magron, J. Wang, and M. Maggio, "Stability of linear systems under extended weakly-hard constraints," *IEEE Control Systems Letters*, vol. 6, pp. 2900–2905, 2022.

[2] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, (Honolulu, HI, USA), pp. 7263–7271, IEEE, 2017.

[3] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, vol. 78 of *Proceedings of Machine Learning Research*, pp. 1–16, PMLR, 13–15 Nov 2017.

[4] C. Hobbs, D. Roy, P. S. Duggirala, F. D. Smith, S. Samii, J. H. Anderson, *et al.*, "Perception computing-aware controller synthesis for autonomous systems," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 457–462, IEEE, 2021.

[5] S. Yao, Y. Hao, Y. Zhao, H. Shao, D. Liu, S. Liu, T. Wang, J. Li, and T. Abdelzaher, "Scheduling real-time deep learning services as imprecise computations," in *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–10, IEEE, 2020.