Building Safe Autonomous Systems Using Imperfect Components

Shengjie Xu, Prateek Ganguli, Tingan Zhu, Arkaprava Gupta, Bineet Ghosh[#], Kurt Wilson[*], Abdullah Al Arafat[*,+], John Baugh[*], Zhishan Guo,[*] Benjamin Berg, Parasara Sridhar Duggirala, and Samarjit Chakraborty

The University of North Carolina at Chapel Hill, USA
[#]University of Alabama, Tuscaloosa, USA
[*]North Carolina State University, USA
[+]Florida International University, USA
{sxunique,pganguli,tzhu,arka,ben,psd,samarjit}@cs.unc.edu
bineet@ua.edu, {kwilso24,jwb,zguo32}@ncsu.edu, aarafat@fiu.edu

Abstract. Modern autonomous systems are composed of interacting components for control, scheduling, and learning, each operating under practical limitations such as timing uncertainty and imperfect perception. Traditional design and verification approaches aim for component-level perfection, an assumption that is increasingly untenable for complex systems. This paper advocates a shift toward *quantitative, system-level safety* that explicitly accounts for imperfect components and characterizes how their combined effects impact closed-loop behavior. We introduce safety metrics based on trajectory deviation and reachable set expansion, develop methods to check and synthesize safe schedules under deadline misses, and extend the framework to learning-enabled systems via edge–cloud control and safety-driven resource allocation. Together, these results demonstrate how safe autonomous systems can be systematically designed without requiring their individual components to be perfect.

Keywords: Cyber-Physical System · Split Computing · CPS safety

1 Introduction

Unlike classical machines that are defined by their physical structure (think mechanical watches and combustion engines), modern autonomous systems are often *software-defined*, meaning that their behaviors are primarily specified in software. Modern-day cars, for example, include software with several million lines of code distributed across numerous Electronic Control Units (ECUs). Each ECU executes multiple feedback control tasks responsible for functions such as engine management, braking, suspension, and vibration control. This software-centric design has enabled sophisticated features, such as adaptive cruise control, lane-keeping assist, and advanced stability control systems.

While software-defined systems enable highly customizable system design, the complexity of the software and the hardware platforms also introduces potential for safety violations. For example, a processor may be overwhelmed if too many

tasks are sharing it, causing real-time control tasks on it to miss their deadlines. These deadline misses may cause the autonomous system to deviate from its intended behavior and potentially violate its safety requirements, especially if the control strategies are designed under the assumption that the control tasks are always executed on time. *Formal verification* methods are techniques used to check the safety of autonomous systems. The formal verification of a system involves creating a mathematical model and analytically proving that it satisfies certain properties. While it can provide mathematical proofs of system safety, it often focuses on ensuring that each component operates in an idealized manner—for example, deadlines of real-time tasks are never missed. These requirements may not always be feasible given the practical constraints of the systems. As a result, while verification techniques are constantly evolving, they face difficulties in being adopted for practical systems. On the other hand, industry practice often relies on measurement-based testing and heuristic safety margins—such as adding a "50% buffer" to processor utilization—without a formal understanding of the actual safety boundaries.

We consider these practical limitations as defining characteristics of **imperfect components**, as opposed to the abstract, idealized components often assumed during the design of an autonomous system. Such imperfections can manifest in many forms. It could be competing tasks on a shared computational platform, where it is impossible for every task to operate in its idealized condition (e.g., with no deadline misses). It could be a neural network providing measurements of system states, or a communication network that intermittently drops packets. It could also be a conscious design trade-off, such as limited memory preventing the deployment of the most accurate neural network, or a cost ceiling that requires a system to tolerate a certain hardware error rate. Given the increasing variety and popularity of modern autonomous systems, we argue that the most effective path to verifying autonomous system safety is an approach that combines the rigor of formal methods with the pragmatism of acknowledging that components are not perfect. Instead of insisting that individual components be perfect, we accept that they may be *imperfect* but strive to *quantify* the impact of such imperfections on overall system-level properties like control safety. The common theme of the methods in this paper, therefore, is a shift from **qualitative, component-level requirements** (e.g., "the task must never miss its deadline" or "the feedback controller is stable") to **quantitative, system-level properties** (e.g., "the combined effects of task deadline misses, controller behavior, and estimator inaccuracies should not cause the system's state to deviate more than a specified distance from its ideal trajectory"). We introduce below a few specific instances of what we consider imperfect components and discuss their implications on analyzing system safety.

Worst-Case Execution Time (WCET) and the Long Tail: With the continuous increase in the complexity of both software functionality and hardware platforms, guaranteeing that every deadline is met has become increasingly difficult. One reason is that the heterogeneous architecture of modern hardware, with its caches, pipelines, and memory hierarchies, makes task execution times highly variable,

often with a long-tail distribution. This introduces two distinct challenges for traditional real-time analysis. First, it's difficult to determine a tight, true upper bound on the worst-case execution time (WCET). Second, any analysis based on a conservative WCET is often overly pessimistic, since these worst-case scenarios occur with extremely low frequency. We argue that not all deadline misses are created equal. For example, a deadline miss may have vastly different implications for system safety depending on whether it occurs when the system is in a stable, steady state or during a critical maneuver requiring large, rapid corrections. Instead of treating every WCET overrun as a system failure, we should focus on quantifying the **effects of these overruns on system-level behavior**.

Neural Network Sizing and Performance Trade-offs: Modern cyber-physical systems increasingly rely on machine learning, particularly deep neural networks (Deep Neural Networks (DNNs)), for perception and decision-making tasks. This integration creates new challenges for implementation and safety. Designing DNNs for real-time systems requires a careful balance of **accuracy, latency, and resource consumption**. Model complexity is a key factor: deeper networks generally achieve higher accuracy but demand more computation, prompting the use of techniques such as pruning, quantization, and knowledge distillation. Batch processing introduces another trade-off, as GPUs are most efficient with large batch sizes, yet real-time systems typically demand low-latency, single-sample inference. The memory footprint also constrains design, since large models may not fit in memory alongside other applications. These trade-offs must be considered holistically, because perception accuracy directly influences downstream control performance and system safety. The central challenge lies in jointly optimizing perception and control subsystems under strict real-time constraints.

Edge–Cloud Computing: Modern CPS increasingly rely on edge–cloud architectures to balance computational demand, latency, and energy efficiency. In such systems, the edge device processes time-sensitive tasks locally, while the cloud provides additional computational power for more complex workloads. A central design choice is how to partition DNNs across the edge and the cloud, a strategy often referred to as **split computing**. By placing the initial layers of a network on the edge, raw sensor data can be compressed into intermediate features before transmission, reducing communication bandwidth. The remaining layers are then executed in the cloud, where abundant resources enable deeper inference and higher accuracy. This approach, however, introduces new challenges. Communication latency and network variability can undermine real-time guarantees, particularly in safety-critical control loops. Partitioning also impacts security and privacy. Effective edge–cloud computing requires joint consideration of latency, accuracy, bandwidth, and safety trade-offs, ensuring that perception pipelines integrate seamlessly with downstream control logic.

1.1 Paper Organization

The rest of the paper is organized as follows. Section 2 provides the mathematical foundation for analyzing system-level safety, including control system models and

formal methods for characterizing imperfect timing behavior. Section 3 defines two quantitative, system-level safety metrics that capture how imperfections affect overall system behavior, forming the basis for all subsequent analysis. Section 4 develops methods to verify whether a system remains safe under timing uncertainties, using the quantitative safety metric instead of traditional qualitative metrics like control stability. Section 5 demonstrates the complementary problem of Section 4: how to synthesize schedules that guarantee safety when resource constraints make deadline misses unavoidable. Section 6 introduces a statistical approach to the schedule synthesis problem in Section 5 that reduces the conservatism of deterministic methods while still providing rigorous safety guarantees. Section 7 shifts gears and addresses imperfections in neural network estimators by designing controllers that strategically combine fast but less accurate edge inference with accurate but delayed cloud inference. Section 8 shows how to allocate limited computational resources among multiple estimators by prioritizing accuracy for state variables that most strongly influence system-level safety. Section 9 introduces the notion of environment-aware system-level safety, and finally Section 10 discusses related work. Section 11 concludes with reflections on the shift from component-level to system-level reasoning and directions for future work.

2 System Model and Notation

2.1 Control System Models

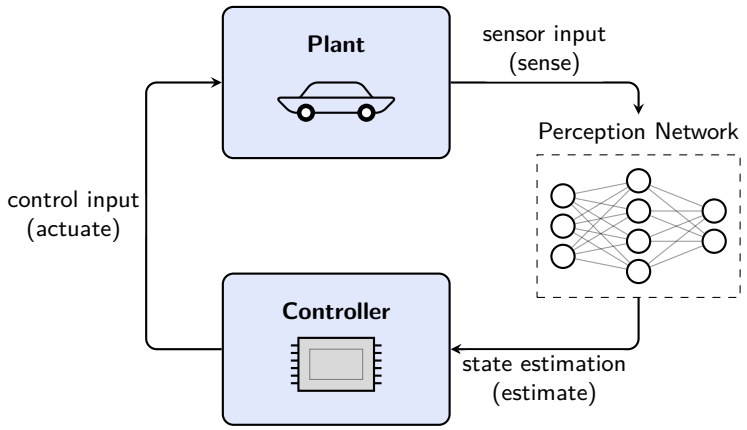


Fig. 1: A basic feedback control loop.

Control theory provides the mathematical foundation for cyber-physical systems. At their core, these systems implement a *sense-compute-actuate* paradigm (Figure 1): sensors measure the physical state of the plant, a computational unit processes this information to determine an appropriate control action, and actuators apply the action back to the physical environment. The interaction of

these three elements closes the feedback loop that underlies modern embedded control systems.

A standard mathematical representation of such systems is given by the *state-space model*. In its most general form, the state evolves according to a system of first-order differential equations,

$$\dot{x}(t) = f(x(t), u(t), t),$$
$$y(t) = h(x(t), u(t), t),$$

where $x(t) \in \mathbb{R}^p$ denotes the state vector, $u(t) \in \mathbb{R}^q$ the control input, and $y(t) \in \mathbb{R}^r$ the measured output. The functions f and h capture the physical laws governing the plant and are in general nonlinear.

Definition 1 (Time-Invariant System). *A* time-invariant *system is a dynamical system that does not depend explicitly on time:*

$$\dot{x}(t) = f(x(t), u(t)),$$
$$y(t) = h(x(t), u(t)).$$

Within the class of time-invariant systems, Linear Time-Invariant (LTI) systems are of particular interest due to their relevance to many important physical systems, either directly or through linear approximation around fixed operation points.

Definition 2 (Linear Time-Invariant System). *An LTI system is expressed as:*

$$\dot{x}(t) = Ax(t) + Bu(t), \tag{1}$$
$$y(t) = Cx(t) + Du(t), \tag{2}$$

where $A \in \mathbb{R}^{p\times p}$, $B \in \mathbb{R}^{p\times q}$, $C \in \mathbb{R}^{r\times p}$, and $D \in \mathbb{R}^{r\times q}$ are constant matrices.

Controllers are designed to influence this dynamical system so that the output follows a desired reference signal. The most common paradigm is *feedback control*, in which the controller computes its action as a function of the current state or output. In general, a feedback controller has the form:

$$u(t) = \pi(x(t), x_r(t)),$$

where π denotes the control policy and $x_r(t)$ the reference. A classical example is the proportional-integral-derivative (PID) controller,

$$u(t) = K_p e(t) + K_i \int_0^t e(s)\, ds + K_d \frac{de(t)}{dt},$$

with error $e(t) = r(t) - x_r(t)$.

While dynamics of physical plants are continuous, controllers are implemented as software tasks running on digital processors and thus operate on discrete time intervals. This requires *discretization* of the continuous dynamics with stepwise updates over a fixed sampling period T.

Definition 3 (Discrete LTI System). *The discretization of an LTI system has the form*

$$x[k+1] = \Phi x[k] + \Gamma u[k], \tag{3}$$

where

$$\Phi = e^{AT}, \quad \Gamma = \int_0^T e^{As}\, ds\, B.$$

The choice of T is important: it must be small enough to capture the system dynamics accurately, yet large enough to be realizable within computational and platform constraints. This trade-off between control performance and implementation feasibility is one of the recurring themes in the design of real-time control systems. A discrete control input can be computed as

$$u[k] = -Kx[k] + x_r[k], \tag{4}$$

where K is the feedback gain matrix and $x_r(t)$ is the reference state.

Since control signal computation and actuation introduce a delay, the new control input cannot be applied immediately at the start of each sampling period. We capture this behavior as follows:

Definition 4 (Delayed Discrete LTI System). *A discrete LTI system with a sensor-to-actuator delay of D is modeled as*

$$x[k+1] = \Phi x[k] + \Gamma_1(D)u[k-1] + \Gamma_0(D)u[k], \tag{5}$$

with

$$\Gamma_0(D) = \int_0^{T-D} e^{As}\, ds\, B, \qquad \Gamma_1(D) = \int_{T-D}^{T} e^{As}\, ds\, B.$$

For convenience, Equation (5) can be expressed in augmented form:

$$x_{\text{aug}}[k+1] = \Phi_{\text{aug}} x_{\text{aug}}[k] + \Gamma_{\text{aug}} u[k], \tag{6}$$

where

$$x_{\text{aug}}[k] = \begin{bmatrix} x[k] \\ u[k-1] \end{bmatrix}, \quad \Phi_{\text{aug}} = \begin{bmatrix} \Phi & \Gamma_1(D) \\ 0 & 0 \end{bmatrix}, \quad \Gamma_{\text{aug}} = \begin{bmatrix} \Gamma_0(D) \\ \mathbf{I} \end{bmatrix}.$$

Here, x_{aug}, Φ_{aug}, and Γ_{aug} each have one additional dimension compared to their counterparts in Equation (3). This allows controller design for the delayed system to proceed as in the delay-free discrete case.

A special design paradigm is the Logical Execution Time (LET) paradigm, where the current control input is always applied at the start of the next sampling period, i.e., the sensor-to-actuator delay is set to equal the period $D = T$. This enables a level of separation between control performance and the timing behavior of the underlying implementation, so long as the actual delay does not exceed the sampling period. In this case, Equation (5) simplifies to

$$x[k+1] = \Phi x[k] + \Gamma(D)u[k-1], \tag{7}$$

2.2 Finite Automata

Automata theory provides a formal foundation for modeling and analyzing discrete-state systems, such as categorizing the deadline hit/miss patterns of a controller task. An automaton consists of a set of abstract states, a set of actions or inputs, and a transition relation that specifies how the system evolves from one state to another in response to actions. We review here the basic notions that will be used throughout the paper.

Definition 5 (Finite Automaton). *A finite automaton is a tuple*

$$\mathcal{A} = \langle \Sigma, Q, \delta, s_0, F, \Omega, \lambda \rangle,$$

where

- Σ *is a finite set of input symbols;*
- Q *is a finite set of locations or states;*
- $\delta : Q \times \Sigma \to Q$ *is a transition function, specifying the successor location for a given state and input symbol;*
- $s_0 \in Q$ *is the initial state of the automaton;*
- $F \subseteq Q$ *is a finite set of accepting states;*
- Ω *is an optional finite set of output symbols;*
- $\lambda : Q \times \Sigma \to \Omega$ *is an optional output function that maps each pair of a state and input symbol to an output symbol in the output alphabet* Ω*.*

A *word* (also called a *string*) over the alphabet Σ is a finite sequence $w = a_1 a_2 \cdots a_n$ where each $a_i \in \Sigma$. The empty word is denoted by ε. We write Σ^* for the set of all finite words over Σ, and Σ^+ for the set of all non-empty words.

Given a word $w \in \Sigma^*$, the automaton $\mathcal{A}$ processes w by starting from s_0 and successively applying the transition function according to each symbol of w. Formally, we extend δ to words by defining

$$\delta^*(s, \varepsilon) = s, \quad \delta^*(s, aw) = \delta^*(\delta(s, a), w),$$

for $s \in Q$, $a \in \Sigma$, and $w \in \Sigma^*$. A word $w \in \Sigma^*$ is said to be *accepted* if $\delta^*(s_0, w) \in F$. The set of all accepted words is called the *language* of the automaton:

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(s_0, w) \in F\}.$$

When an output function is present, processing a word w yields not only the final state $\delta^*(s_0, w)$, but also a corresponding sequence of output symbols. This formulation allows automata to serve as transducers, mapping input behaviors to structured output traces.

2.3 Weakly-Hard Constraints and Deadline-Miss Handling Strategies

Safety-critical systems are traditionally modeled as *hard real-time* systems, where every control task must meet its deadline without exception. This assumption has

historically enabled a clean separation between controller design and embedded implementation. However, guaranteeing that *all* deadlines are met is increasingly difficult due to the growing complexity of software stacks and hardware platforms in modern embedded systems.

Weakly-hard constraints [7] provide a more nuanced formalism for characterizing patterns of deadline hits and misses, relaxing the requirement that every job meets its deadline. We consider two types of weakly-hard constraints introduced in [7].

Definition 6 ($\binom{m}{k}$ Weakly-Hard Constraint). *For integers m, k with $0 \leq m \leq k$, a $\binom{m}{k}$ constraint requires that in every window of k consecutive activations of a task, at least m of them must meet their deadlines. Equivalently, at most $(k-m)$ deadlines may be missed in any such window.*

Definition 7 ($\overline{\langle m \rangle}$ Weakly-Hard Constraint). *For a positive integer m, a $\overline{\langle m \rangle}$ constraint requires that a task does not miss more than m deadlines consecutively.*

A deadline hit/miss sequence satisfying a $\binom{m}{k}$ constraint can be represented as a word $w \in \{0,1\}^H$, where a symbol 1 denotes a job completing on time and 0 denotes a miss. A key property of weakly-hard constraints is that they are *regular*: the set of all sequences satisfying a given $\binom{m}{k}$ constraint can be recognized by a finite automaton corresponding to that constraint (Figure 2). We defer a detailed discussion of how deadline misses affect control performance and safety to Section 3. Here, we outline the categories of *deadline-miss handling strategies* used in practice.

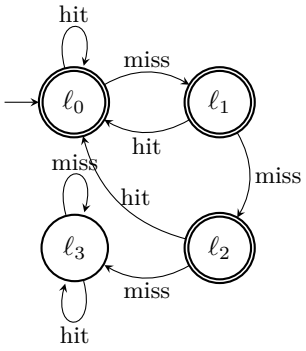


Fig. 2: An example finite automaton capturing the $\overline{\langle 3 \rangle}$ constraint.

Input substitution policies. These determine what control input to apply when a task misses its deadline. A common choice is the `Hold` policy, where the last successfully computed input is re-applied. Another option is the `Zero` policy, where the input is set to zero. Such substitution policies influence the short-term stability and performance of the closed-loop system.

Task overrun policies. These determine how the scheduler responds when a job exceeds its deadline. In the `Skip-Next` policy, the current job is allowed to finish execution, and the next job release is skipped. In the `Kill` policy, the overrun job is terminated immediately and the next job is released as scheduled. These strategies affect the system's computational load and the alignment of control updates with the plant dynamics. In all cases, deadline misses cause deviations in the closed-loop trajectory relative to the nominal case where every deadline is met. Quantifying these deviations is the subject of the next section. A list of notation together with their descriptions and locations in the paper is provided in Table 1.

3 Notions of System-Level Safety

In Section 2.3, we introduced weakly-hard constraints as a way to capture nuanced patterns of deadline hits and misses, moving beyond the traditional assumption that every deadline must always be met. A natural next step is to quantify the impact of specific hit/miss patterns on control safety. In practice, however, deadline misses are not the only source of imperfection: factors such as neural network approximation errors and other implementation artifacts may also cause the system to deviate from its nominal trajectory. The central question is therefore: *under what conditions does the system remain safe despite deviations introduced by imperfect components?*

Classical control theory primarily focuses on system stability, which concerns whether a system returns to equilibrium after disturbances. However, for safety-critical cyber-physical systems, stability alone is insufficient. A system may be stable yet still exhibit dangerous transient behaviors that compromise safety. Unlike binary stability analysis, quantitative safety metrics allow us to categorize different system configurations on a much finer basis than only two classes, viz., whether the system is stable or not.

In the following, we introduce two such measures over a finite horizon H: (i) deviation from an ideal (nominal) trajectory, and (ii) the maximum diameter of reachable sets. The choice of these specific metrics is motivated by the nature of imperfections in autonomous systems. Control strategies are typically designed to maintain the plant around a nominal operating point or trajectory, yielding asymptotic stability. However, asymptotic stability is a liveness property and, by itself, does not guarantee system safety during transients due to external disturbances or inaccurate inputs that push the system away from this equilibrium [2]. **Deviation from nominal trajectories** (Section 3.1) directly quantifies this transient response, capturing dangerous excursions (e.g., swerving out of a lane) that asymptotic stability analysis might miss. Complementing this, the **maximum diameter of reachable sets** (Section 3.2) measures the "spread" of uncertainty introduced by sensing errors. A large diameter indicates that the controller effectively lacks precise state knowledge, which is a safety risk even if the actual trajectory remains valid. Finally, we restrict our analysis to a **finite horizon** because we are specifically interested in the system's recovery behavior following such disturbance events. Since modeling every possible infinite-horizon scenario is intractable, verifying safety over a finite look-ahead window—similar to Model Predictive Control (MPC)—provides a practical and effective measure of system safety in realistic settings.

3.1 Deviation from Ideal Trajectory

The first safety metric quantifies how far the actual system trajectory deviates from an ideal reference behavior [56]. We begin by formalizing the notion of a system trajectory.

Definition 8 (Trajectory). *A trajectory is a function mapping the time interval (for continuous systems) or time instants (for discrete systems) to the state space. Formally, a continuous trajectory has the form* $\tau : [0, H_c] \to \mathbb{R}^p$*, and*

a discrete trajectory has the form $\tau : \{0, 1, \ldots, H\} \to \mathbb{R}^p$*, where* $H_c \in \mathbb{R}^+$ *and* $H \in \mathbb{Z}^+$ *are continuous and discrete time horizon, respectively.*

In absence of deadline misses, the trajectory of a dynamical system can be determined by applying the discrete-time dynamics in Equation (3) and the controller computation Equation (4) repeated, starting from an initial state $x[0]$. Given a sequence of deadline hits and misses, e.g., $w = 010101$, the corresponding trajectory can be calculated by applying either 0 or previous control input depending on the deadline miss handling strategy. When it is clear from context, we will use $\tau(w)$ to denote the trajectory corresponding to w. More broadly, we define the set of trajectories:

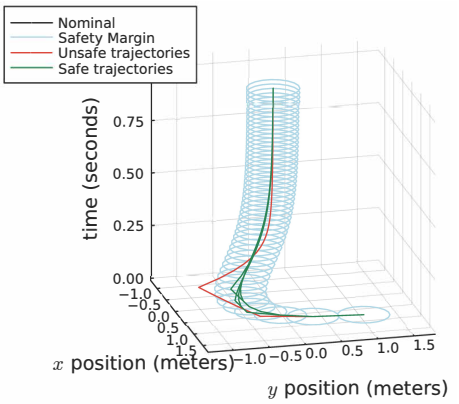


Fig. 3: Safe & unsafe trajectories.

Definition 9 (Set of trajectories). *We use* $\mathcal{T}$ *to indicate a set of trajectories and* $\mathcal{T}(\cdot) = \{\tau \mid \tau \vDash \cdot\}$ *to indicate trajectories that result from certain conditions.*

With a slight overloading of notations, we define $\mathcal{T}\binom{m}{k} = \{\tau(w) \mid w \vDash \binom{m}{k}\}$ to denote all trajectories corresponding to the weakly-hard constraint. A trajectory computed in the absence of external disturbances, timing uncertainties, or other implementation imperfections is referred to as the *nominal trajectory* $\tau^{\texttt{nom}}$. This trajectory represents the intended behavior of the control system under ideal conditions.

Definition 10 (Deviation). *To compare different system evolutions, we define the* deviation *between two trajectories* τ_1 *and* τ_2 *as*

$$\texttt{Dev}(\tau_1, \tau_2) = \max_{k \in [0,H]} d(\tau_1[k], \tau_2[k]) \,, \tag{8}$$

where $d(\cdot, \cdot)$ *is a distance metric on the state space.*

Typical choices include the Euclidean norm, the max norm, or, for set-valued comparisons, the Hausdorff distance. This definition captures the worst-case pointwise separation between two trajectories over the time horizon $[0, H]$.

As an illustration, consider a system subject to deadline misses. Figure 3 shows a number of trajectories as the result of varying deadline hit/miss patterns. The nominal trajectory $\tau^{\texttt{nom}}$, shown as the black line in Figure 3, is the trajectory resulting from no deadline misses. We denote this as $\tau^{\texttt{nom}} = \tau(w^{\texttt{nom}})$, where $w^{\texttt{nom}} = 1^H$ denotes a sequence of H deadline hits. Suppose the hit/miss pattern is represented as a word $w = 110110\ldots$, where a symbol 1 indicates a deadline hit and 0 indicates a miss. The corresponding trajectory $\tau(w)$ is computed from

Equation (3) by applying control inputs from Equation (4) whenever a deadline is met. When a deadline is missed, the system either re-applies the last valid control input (`Hold` strategy) or applies zero input (`Zero` strategy). In both cases, the resulting trajectory deviates from the nominal trajectory $\tau^{\texttt{nom}}$, and the deviation can be quantified by Equation (8).

This formulation also applies in stochastic settings where system behavior is observed through random sampling. Each random sample corresponds to a trajectory that may be compared against the nominal trajectory $\tau^{\texttt{nom}}$ or against other sampled trajectories. Deviation thereby provides a natural metric for quantifying the dispersion of trajectories induced by probabilistic variations.

The deviation metric is especially useful when the ideal system behavior is well understood. In such cases, safety requirements can often be expressed as an upper bound on the allowable deviation from the nominal trajectory. Formally, given a tolerance $\texttt{D}^{\texttt{safe}} > 0$, the system is considered safe if every admissible trajectory $\tau \in \mathcal{T}$ satisfies $\texttt{Dev}(\tau, \tau^{\texttt{nom}}) < \texttt{D}^{\texttt{safe}}$. This condition ensures that, despite imperfections, the system remains within a prescribed neighborhood of its ideal behavior throughout the time horizon.

3.2 Maximum Diameter of Reachable Sets

The second quantitative measure we consider is based on the notion of *reachable sets*. Instead of analyzing a single trajectory starting from an initial state, we study the evolution of a *set of states* over time.

Definition 11 (Diameter of a Set). *Given a set $X \subseteq \mathbb{R}^n$, its* diameter *is defined as*

$$\texttt{Diam}(X) = \max_{x_1, x_2 \in X} d(x_1, x_2),$$

where $d(\cdot, \cdot)$ is a distance metric such as the Euclidean norm.

The diameter measures the maximum pairwise separation between states in X, and thus captures the spread or uncertainty within the set.

To reason about sets of states under dynamics, we employ the set-based reachability analysis of linear systems [44]. For sets $X[k]$ of states and $U[k]$ of inputs, the successor set is computed as

$$X[k+1] = \Phi X[k] \oplus \Gamma U[k], \tag{9}$$

where $\Phi X[k] = \{\Phi x \mid x \in X[k]\}$ and $\Gamma U[k] = \{\Gamma u \mid u \in U[k]\}$. $\oplus$ denotes the *Minkowski sum*, defined for sets $A, B \subseteq \mathbb{R}^n$ as

$$A \oplus B = \{\, a + b \mid a \in A,\, b \in B \,\}.$$

Similarly, the control law extends to sets as

$$U[k] = (-K)X[k] \oplus \{x_r[k]\}, \tag{10}$$

where $-KX[k] = \{-Kx \mid x \in X[k]\}$ and $\{x_r[k]\}$ is the singleton set containing the reference value at time step k.

Practical considerations of calculating reachable sets: Equation (9) represents the standard recursive formulation for set-based reachability analysis of linear systems [44]. While theoretically sound, exact computation of this recursion is often intractable because the geometric complexity of the set $X[k]$ (e.g., the number of vertices or faces) grows with each time step. To make these computations practical for the case studies in this paper, we utilize **Zonotopes** as the underlying set representation. Zonotopes are centrally symmetric polytopes defined by a center $c \in \mathbb{R}^n$ and a matrix of generators $G \in \mathbb{R}^{n \times g}$. They are particularly well-suited for linear control systems because they are closed under linear maps and Minkowski sums—the two primary operations in Equation (9). Specifically, if $X[k]$ is a zonotope (c, G), then the linear map $\Phi X[k]$ is simply $(\Phi c, \Phi G)$.

For our experiments, we implemented the reachability analysis using the **JuliaReach** toolbox [8]. To prevent the number of generators from growing indefinitely, we apply order reduction techniques at each step [45]. This process approximates the exact zonotope $X[k+1]$ with a lower-order zonotope that encloses the original, ensuring a safe over-approximation while maintaining computational efficiency.

In scenarios where estimation error is relative to the state magnitude (as in our neural network experiments), the input set $U[k]$ in Equation (9) becomes state-dependent. We model this by computing an error bound zonotope W_k at each step. Let interval$(X[k])$ denote the axis-aligned bounding box of the current state set. The disturbance set is defined as:

$$W_k = \{-Ke \mid e \in \mathbb{R}^n, |e_i| \leq \epsilon_i \cdot \sup_{x \in X[k]} |x_i|\} \tag{11}$$

where ϵ_i is the relative error rate for the i-th state dimension. This set is added via Minkowski sum to the propagated state dynamics.

Starting from an initial set $X[0]$, recursive application of Equations (9) and (10) yields a sequence of reachable sets $\mathcal{F} = X[1], X[2], \ldots, X[H]$ over the horizon H. This sequence, often referred to as a *flowpipe*, generalizes the notion of a trajectory by capturing all possible evolutions of the system consistent with uncertainties in the state or input.

Definition 12 (Maximum Diameter of Reachable Sets). *We overload the notation slightly and define the Maximum Diameter of Reachable Sets (MDRS) over the time horizon as*

$$\texttt{Diam}(\mathcal{F}) = \max_{k \in [0,H]} \texttt{Diam}(\mathcal{F}[k]).$$

This metric thus measures the worst-case uncertainty of system states at any time within the horizon. A safety requirement $\texttt{D}^{\texttt{safe}}$ may be imposed to ensure that

$$\texttt{Diam}(\mathcal{F}) < \texttt{D}^{\texttt{safe}},$$

thereby limiting the uncertainty of system states and ensuring that the system remains within acceptable safety margins.

As an illustrative example, consider a learning-enabled autonomous system equipped with a neural network estimator for the system state. Suppose the estimator produces an estimated state $\widehat{x} \in \mathbb{R}^p$ that may deviate by up to 20% from the true state $x \in \mathbb{R}^p$ in each dimension. Formally, this can be modeled as

$$\widehat{X}[k] = \{\widehat{x} \mid (1-\alpha)x_i \leq \widehat{x}_i \leq (1+\alpha)x_i\}, \quad \alpha = 0.2,$$

Propagating these sensing errors using Equation (9) and $U[k] = (-K)\widehat{X}[k] \oplus \{x_r[k]\}$ yields a flowpipe of reachable sets that incorporates the effect of estimation error. And the MDRS $\texttt{Diam}(\mathcal{F})$ then provides a bound on how the sensing error manifests as system uncertainty over the time horizon.

3.3 Evaluating Safety

Whether expressed in terms of deviation from the nominal trajectory or the diameter of reachable sets, the ultimate goal of these metrics is to evaluate the safety of a particular system configuration or property. In practice, this evaluation can take many forms. For example, we may wish to study the quantitative safety of a control system operating under a weakly-hard constraint $\binom{m}{k}$, or to analyze the effect of using a specific neural network configuration J within a learning-enabled controller. Both cases require a systematic way to denote and reason about safety properties.

When the context is clear, we use the shorthand notation $\texttt{Dev}(\cdot)$ or $\texttt{Diam}(\cdot)$ to denote the quantitative safety property of a given configuration. For instance, $\texttt{Dev}(\tau)$ denotes the deviation of a trajectory τ from a nominal trajectory $\tau^{\texttt{nom}}$, as introduced in Section 3.1. Similarly, $\texttt{Dev}\binom{m}{k}$ denotes the maximum deviation between any trajectory consistent with the weakly-hard constraint $\binom{m}{k}$ and a nominal trajectory $\tau^{\texttt{nom}}$. In the case of uncertainty induced by a neural network estimator, $\texttt{Diam}(J)$ denotes the Maximum Diameter of Reachable Sets (MDRS) of the system when using network configuration J, as introduced in Section 3.2.

We further distinguish between exact, estimated, and bounded evaluations of safety properties. The notation $\widehat{\texttt{D}}$ denotes an estimated value of the safety metric $\texttt{D}$, typically obtained through simulation, sampling, or statistical analysis. By contrast, $\overline{\texttt{D}}$ denotes a formally derived upper bound on the safety metric, which provides a conservative guarantee even in the presence of uncertainties. These notations allow us to reason consistently about safety across different contexts, while distinguishing between approximate and certified results.

4 How to Check That a Schedule is Safe

Growing complexity in embedded systems has introduced timing uncertainties that may violate the hard real-time assumption that control task deadlines are never missed. Deadline misses can lead to deviation of the control system from its designed behavior, and may lead to safety deviations if such timing uncertainty is not considered in system design and validation. We have introduced weakly-hard constraints (Section 2.3) not only because they provide foundation for further safety analysis (as discussed in Section 3.1), but also because ensuring *at least*

m deadlines are met in a window of k is more practical than ensuring that *all* deadlines are met in a real implementation.

This section addresses the problem of checking the quantitative safety metrics of a control system with potential deadline misses that are consistent with weakly-hard constraints in the form of $\overline{\langle m \rangle}$. The central challenge is that directly computing the maximum deviation is computationally intractable, as it requires exploring all possible runs of the automaton. To overcome this difficulty, we present a set of approximation techniques that upper bound the deviation while remaining computationally feasible.

4.1 Problem Statement

We now formalize the analysis problem. Let $\mathcal{A} = \langle \Sigma, Q, \delta, s_0, F \rangle$ be a finite automaton as defined in Section 2.2 capturing a weakly hard constraint $\overline{\langle m \rangle}$. The *maximum deviation problem* is:

Problem 1 (Maximum Deviation). Given a finite automaton $\mathcal{A}$, an initial set of states, a time horizon H, and a nominal run $\tau^{\mathtt{nom}}$, compute

$$\mathtt{Dev}\overline{\langle m \rangle} = \max_{\tau \in \mathcal{T}(\overline{\langle m \rangle})} \mathtt{Dev}(\tau, \tau^{\mathtt{nom}}), \tag{12}$$

where $\mathcal{T}(\overline{\langle m \rangle})$ is the set of all possible trajectories resulting from deadline miss patterns that satisfy the weakly hard constraint $\overline{\langle m \rangle}$.

Since enumerating all 2^H runs is intractable for large H values, we focus on computing a safe upper bound $\overline{\mathtt{D}} \geq \mathtt{Dev}\overline{\langle m \rangle}$:

Problem 2 (Deviation Bound). Given $\mathcal{A}$ and $\tau^{\mathtt{nom}}$, compute a bound $\overline{\mathtt{D}}$ such that

$$\mathtt{Dev}\overline{\langle m \rangle} \leq \overline{\mathtt{D}} \quad \text{for all } \tau \in \mathcal{T}. \tag{13}$$

4.2 Approximation Techniques

Uncertain Linear Systems The first approach approximates the discrete-time switched linear system [71] defined by the automaton as an uncertain linear system (specifically, a linear difference inclusion) [39]. While the automaton enforces specific transitions between dynamics matrices based on the current state, this method relaxes those constraints, allowing any dynamics matrix from the set of all possible transitions to occur at any time step. Formally, we define the system:

$$x[t+1] = \Lambda x[t],$$

where $\Lambda \subseteq \mathbb{R}^{p \times p}$ represents the set of all possible dynamics matrices. An example of Λ is: $\begin{bmatrix} 1 & \alpha \\ 0 & 2 \end{bmatrix}$, where $\alpha \in [2, 3]$ represents a parameter. Intuitively, this treats the switching behavior as uncertainty, covering all possible dynamics matrices simultaneously to compute a conservative deviation bound. Since the uncertain

dynamics matrix can represent a set of linear dynamics matrices, we leverage it to compute an upper bound on $\texttt{Dev}$. The key idea is to encode all dynamics matrices arising from finite automaton transitions into a single uncertain dynamics matrix. Formally, for a given finite automaton $\mathcal{A}$, we define

$$\Lambda = \big\{\lambda(s,a) \;\big|\; s \in Q, a \in \Sigma\big\}. \tag{14}$$

This construction captures all possible sequences of hits and misses by collecting every transition's dynamics matrix (i.e., every output of λ from Definition 5). By assuming that any dynamics in Λ could occur at each time step, we over-approximate all behaviors of the finite automaton.

The reachable set of a uncertain linear system (ULS) at time t represents the possible states under any permissible sequence of actions. Here, Equation (14) yields an over-approximate reachable set for all runs of length t. Computing the Hausdorff distance between this set and the nominal run (as in Definition 10) then provides the desired upper bound on $\texttt{Dev}$. We denote the one-step reachable set from an initial set $x[0]$ as $\mathit{forward}(\Lambda, x[0])$, so that $x[1] = \Lambda x[0]$ and $x[1] = \mathit{forward}(\Lambda, x[0])$. To represent the uncertain dynamics computationally, we use an interval matrix that over-approximates Λ from Equation (14) as $\tilde{\Lambda}$:

$$\tilde{\Lambda}[i,j] = \big[\min\{\Lambda[i,j]\}, \max\{\Lambda[i,j]\}\big], \tag{15}$$

for all $1 \le i, j \le n$, where n is the system dimension. Since $\tilde{\Lambda} \supseteq \Lambda$, this interval representation safely encompasses all possible dynamics. Algorithm 1 uses this representation to compute an upper bound $\overline{\texttt{D}} \ge \texttt{Dev}$.

Algorithm 1: Computing upper-bound on the deviation as defined in problem 1.

input : A finite automaton $\mathcal{A}$, initial set $x[0]$, nominal run $\tau^{\texttt{nom}}$, time bound H
output : An upper bound $\overline{\texttt{D}} \ge \texttt{Dev}$

1 $\overline{\texttt{D}} \leftarrow 0$;
2 $\tilde{\Lambda} \leftarrow$ Compute using Equation (15); `// Construct interval matrix encoding all hit/miss behaviors.`
3 $x_{nom} \leftarrow \mathit{evol}(\tau_{nom})$; `// Compute the nominal trajectory.`
4 **for** $1 \le t \le H$ **do**
5 $\quad x[t] \leftarrow \mathit{forward}(\tilde{\Lambda}, x[t-1])$; `// Compute reachable set at time` t.
6 $\quad d_t \leftarrow d_H\big(x_{nom}[t], x[t]\big)$; `// Measure deviation from nominal trajectory.`
7 $\quad \overline{\texttt{D}} \leftarrow \max\{\overline{\texttt{D}}, d_t\}$; `// Track maximum deviation.`
8 **return** $\overline{\texttt{D}}$; `// Return the maximum deviation bound.`

Algorithm 1 and its Safety Proof (Sketch) The algorithm begins by constructing the interval matrix $\tilde{\Lambda}$ in Line 3, which encodes all possible hit/miss behaviors. The main loop in Lines 5 to 8 iteratively computes the deviation bound $\overline{\texttt{D}}$. At each time step t: Line 6 computes the one-step reachable set of the ULS, so that $x[t]$ contains all possible system states at time t. Line 7 then measures the

Hausdorff distance between this reachable set and the nominal trajectory. Line 8 updates the running maximum. Finally, Line 9 returns the computed bound $\overline{\mathtt{D}}$.

Correctness follows by induction. Assuming the reachable set $x[t]$ safely contains all true reachable states at time t, the *forward*$(\cdot)$ operator in Line 6 computes an overapproximation of $\bigcup_{A\in\Lambda} Ax[t]$. Thus $x[t+1]$ remains a safe overapproximation, ensuring that the computed deviation bound is valid.

Generalized Recurrence Relations: The uncertain linear system approach treats all dynamics uniformly, ignoring the finite automaton structure. A more precise method exploits the automaton's locations to track which sequences of hits and misses have occurred. Consider an automaton (e.g., Figure 2) where location s_k indicates that k consecutive deadlines have been missed since the last hit.

We introduce notation Ψ_s^t for the reachable set of all trajectories in location s at time t. For instance, $\Psi_{s_k}^t$ captures all states reachable after exactly k consecutive misses at time t. Computing Ψ_s^t requires considering all transitions into location s from every predecessor location.

We now derive generalized recurrence relations for any automaton with at most N consecutive misses, as illustrated in Figure 2. Let Ψ_s^t denote the reachable set in location s at time t. For $t \geq 1$, the recurrence relations are:

$$\Psi_{s_0}^t = \mathit{hull}_{s\in Q}\big(\lambda(s,\mathrm{hit})\cdot\Psi_s^{t-1}\big) \tag{16}$$

$$\Psi_{s_p}^t = \lambda(s_{p-1},\mathrm{miss})\cdot\Psi_{s_{p-1}}^{t-1}, \qquad \text{where } 1\leq p\leq N \tag{17}$$

with initial conditions:

$$\Psi_{s_0}^0 = x[0]; \qquad \Psi_{s_p}^0 = \varnothing, \qquad \text{where } 1\leq p\leq N \tag{18}$$

Equation (16) computes states reachable via a hit from any location, while Equation (17) propagates states forward through consecutive misses.

Algorithm 2 implements these recurrence relations to compute an upper bound on `Dev`.

Algorithm 2 and its Safety Proof (Sketch) The algorithm initializes the recurrence relations in Lines 3 to 5 using Equation (18), placing all initial states in location s_0. The main loop in Lines 6 to 11 computes the deviation bound $\overline{\mathtt{D}}$ iteratively. At each time step t: Lines 7 to 9 compute the reachable sets for all locations using Equations (16) and (17). Line 10 measures the Hausdorff distance between the convex hull of all location-specific reachable sets and the nominal trajectory. Line 11 updates the maximum deviation. Finally, Line 12 returns $\overline{\mathtt{D}}$.

Correctness follows by induction. If the reachable sets are safe at time t, then Lines 7 to 9 compute exact one-step evolutions for each location. Taking the convex hull before measuring distance to the nominal trajectory yields an overapproximation, ensuring $x[t+1]$ remains safe and the deviation bound is valid.

Algorithm 2: Computing upper-bound on the deviation as defined in problem 1.

input : A finite automaton $\mathcal{A}$ encoding maximum number of allowed consecutive misses N, initial set θ, nominal run $\tau^{\texttt{nom}}$, time bound H
output : An upper bound $\overline{\texttt{D}} \geq \texttt{Dev}$
`/* Each location represents a class of hit/miss behaviors. */`

1 $\overline{\texttt{D}} = d_H\big(\texttt{evol}(\tau^{\texttt{nom}})[0], \theta\big)$;
2 $\Psi^0_{s_0} = \theta$; `// Initialize location` s_0 `with the initial set.`
3 **for** $1 \leq k \leq N$ **do**
4 $\Psi^0_{s_k} = \varnothing$; `// Initialize other locations as empty.`
5 **for** $1 \leq t \leq H$ **do**
6 $\Psi^t_{s_0} \leftarrow$ Compute using Equation (16) ; `// Update reachable set for` s_0.
7 **for** $1 \leq k \leq N$ **do**
8 $\Psi^t_{s_k} \leftarrow$ Compute using Equation (17); `// Update reachable sets for` s_k.
9 $d_t \leftarrow d_H\big(\texttt{evol}(\tau^{\texttt{nom}})[t], \mathit{hull}_{0 \leq l \leq N}\{\Psi^t_{s_l}\}\big)$; `// Measure deviation at time` t.
10 $\overline{\texttt{D}} \leftarrow \max\{\overline{\texttt{D}}, d_t\}$; `// Track maximum deviation.`
11 **return** $\overline{\texttt{D}}$; `// Return deviation bound.`

Bounded Runs Both previous methods introduce overapproximation error either globally (ULS) or at each time step (recurrence relations). A third approach achieves greater precision by exhaustively exploring all runs for short horizons, then iteratively applying this exact computation with periodic overapproximations.

Computing the exact maximum deviation Dev requires enumerating all 2^H runs of length H, which is intractable for large H. However, for a short run length $r \ll H$, exhaustive enumeration becomes feasible. The key insight is to compute exact reachable sets for r steps, take convex hulls to overapproximate, then repeat this process to cover the full horizon H.

Two challenges complicate this strategy. First, naively taking a single hull over all final states would lose track of which automaton location each run ended in, potentially creating spurious transitions in subsequent iterations. We address this by grouping runs by their final location and maintaining one hull per location in Q. Second, computing each run independently would introduce massive redundancy, since runs share common prefixes. With $\Sigma = \{\text{hit}, \text{miss}\}$, there are $O(2^r)$ runs of length r, requiring $O(r \cdot 2^r)$ matrix-vector multiplications if computed separately. We improve this to $O(2^r)$ by performing a depth-first traversal of the trie of all runs, maintaining partial results on a stack. This optimization is crucial for practical performance. Algorithm 3 implements this efficient traversal strategy.

The algorithm uses array S as a stack of triples $\langle z, s, a\rangle$ containing the current state, location, and next action to try, with i serving as the stack pointer. Line 6 initializes the stack with the initial state and location. The main loop in Line 7 performs depth-first traversal of the run trie. At each step, one of four cases applies: If we reach a leaf (depth r), Line 9 computes convex hulls and stores them

Algorithm 3: Computing reachable sets for one iteration of the bounded runs method.

```
1  Function BoundedRuns(A, z[0], γ)
       input  : Finite automaton A, set of initial states z[0], run length γ
       output : Mapping from locations to lists of reachable sets over time
2      R ← Mapping from locations to lists of reachable sets;
3      S ← Array of γ named triples ⟨z, s, a⟩;
4      i ← 1;
5      S[i] ← ⟨z[0], s_0, action 1⟩;
6      while i > 0 do
7          if S[i] contains a leaf node then
8              R[S[i].s][:] ← Compute the hulls of all states at all time steps;
9              i ← i − 1;
10         else if the last action, S[i].a, has been tried then
11             i ← i − 1;
12         else if no transition T(S[i].s, S[i].a) then
13             S[i].a ← S[i].a + 1;
14         else
15             S[i + 1] ← ⟨λ(S[i].s, S[i].a)S[i].z, T(S[i].s, S[i].a), action 1⟩;
16             S[i].a ← S[i].a + 1;
17             i ← i + 1;
18     return R;
```

grouped by final location, then backtracks. If all actions from the current node have been explored, Line 12 backtracks. If the current action has no corresponding transition, Line 14 skips to the next action. Otherwise, we extend the current run by one step, update the stack, and descend deeper. When traversal completes, the function returns R, mapping each location to its reachable sets over time.

To cover a full time horizon $H = rJ$, we iteratively apply `BoundedRuns` for J iterations, overapproximating with convex hulls between iterations. Algorithm 4 implements this iterative procedure to compute an upper bound $\overline{\mathtt{D}}$ on the maximum deviation `Dev`.

Algorithm 4 and its Safety Proof (Sketch) The algorithm begins by calling `BoundedRuns` on the initial set in Line 3, populating R with location-indexed reachable sets for the first r steps. Line 5 stores hulled versions in $S[0:r]$. The main loop in Lines 6 to 12 repeats this process $J-1$ times. Each iteration consists of two phases: In Line 7, we call `BoundedRuns` from each location's hull, populating rows of Q with reachable sets indexed by starting location. In Line 10, we take hulls across each column of Q to compute the reachable set for each final location, updating R. Line 12 stores the hulled reachable sets for the next r steps in S. Finally, Line 13 computes and returns the maximum Hausdorff distance between the nominal trajectory and the reachable sets over all rJ steps.

Correctness follows from the combination of exact computation within each `BoundedRuns` call and safe overapproximation via convex hulls between iterations. Since `BoundedRuns` explores all possible runs of length r exactly, and the hulls

Algorithm 4: Bounded runs method for computing maximum deviation from a nominal trajectory.

input : A finite automaton $\mathcal{A}$, set of initial states $z[0]$, nominal run $\tau^{\mathtt{nom}}$, per-tree run length γ, number of iterations J
output : An upper-bound $\overline{\mathtt{D}} \geq \mathtt{Dev}$

1 $S \leftarrow$ list of $rJ + 1$ empty reachable sets;
2 $R \leftarrow \mathtt{BoundedRuns}(\mathcal{A}, z[0], r)$; `// Explore all runs from initial set.`
3 $Q \leftarrow |L| \times |L|$ array of lists of reachable sets over time;
4 $S[0:r] \leftarrow hull_{s \in L}(R[s])$;
5 **for** $2 \leq i \leq J$ **do**
6 **forall** *locations* s *in* $\mathcal{A}$ **do**
7 $\mathcal{A}' \leftarrow \mathcal{A}$ with $s_0 = s$;
8 $Q[s,:] \leftarrow \mathtt{BoundedRuns}(\mathcal{A}', R[s][r], r)$; `// Explore from each location.`
9 **forall** *locations* s *in* $\mathcal{A}$ **do**
10 $R[s] \leftarrow hull_{s \in L}(Q[:,s])$; `// Hull by final location.`
11 $S[ir : ir + r] \leftarrow hull_{s \in L}(R[s])$; `// Store reachable sets for steps` ir `to` $ir + r$`.`
12 **return** $\max_{1 \leq t \leq rJ} \left\{ d_r\left(\mathtt{evol}(\tau^{\mathtt{nom}})[t], S[t]\right) \right\}$;

at iteration boundaries contain all reachable states, the algorithm maintains a safe overapproximation throughout. Therefore, the deviation bound computed in Line 13 is valid.

4.3 Case Studies and Evaluation

We evaluate the three approximation methods on four control system benchmarks.

RC Network The first benchmark is an RC network [34]. The plant model, discretized with a period of 100 ms, is

$$x[t+1] = \begin{bmatrix} 0.5495 & 0.07240 \\ 0.01448 & 0.9332 \end{bmatrix} x[t] + \begin{bmatrix} 0.3781 \\ 0.05234 \end{bmatrix} u[t].$$

Assuming a one-period sensor-to-actuator delay, we design an LQR controller

$$u[t] = \begin{bmatrix} 0.09772 & 0.2504 & 0.07805 \end{bmatrix} \begin{bmatrix} x[t-1] \\ u[t-1] \end{bmatrix}.$$

Electric Steering The second benchmark is an automotive electric steering system based on a permanent magnet synchronous motor [77]. The plant model, discretized with a period of 10 µs, is

$$x[t+1] = \begin{bmatrix} 0.996 & 0.075 \\ -0.052 & 0.996 \end{bmatrix} x[t] + \begin{bmatrix} 0.100 & 0.003 \\ -0.003 & 0.083 \end{bmatrix} u[t].$$

The system is open-loop stable with poles at $0.9957 \pm 0.0626i$. We design a new controller rather than using the one from [77], as the latter appears to destabilize

the system:

$$u[t] = \begin{bmatrix} 0.9067 & 0.07384 & 0 & 0 \\ 0.01041 & 0.9685 & 0 & 0 \end{bmatrix} \begin{bmatrix} x[t-1] \\ u[t-1] \end{bmatrix}.$$

This controller assumes no sensor-to-actuator delay ($K_u = 0$), deliberately stress-testing our methods with a non-optimal design.

Aircraft Pitch The third benchmark is an aircraft pitch control system [127]. The plant model, discretized with period $h = 100\,\text{ms}$, is

$$x[t+1] = \begin{bmatrix} 0.9654 & 5.457 & 0 \\ -0.001338 & 0.9545 & 0 \\ -0.003842 & 5.544 & 1 \end{bmatrix} x[t] + \begin{bmatrix} 0.02842 \\ 0.001969 \\ 0.005641 \end{bmatrix} u[t],$$

where state variable x_3 represents the pitch angle. Assuming a sensor-to-actuator delay of h, we design an LQR controller

$$u[t] = \begin{bmatrix} -0.8551 & 179.2 & 5.999 & 0.3238 \end{bmatrix} \begin{bmatrix} x[t-1] \\ u[t-1] \end{bmatrix}.$$

F1Tenth Car The fourth benchmark is an F1Tenth autonomous racing car [99]. The underlying dynamics follow a bicycle model, which is nonlinear. Since our methods require linear dynamics, we linearize around straight-line motion in the positive x direction and discretize with period 20 ms:

$$x[t+1] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.13 \\ 0 & 0 & 1 \end{bmatrix} x[t] + \begin{bmatrix} 0 \\ 0.02559 \\ 0.3937 \end{bmatrix} u[t] + \begin{bmatrix} 0.13 \\ 0 \\ 0 \end{bmatrix},$$

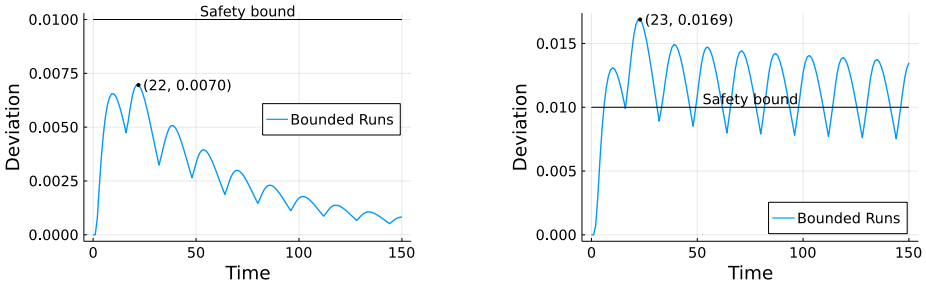
where x_1, x_2, x_3 are the x coordinate, y coordinate, and heading angle, respectively. Since x_1 grows linearly with time in the linearized model and does not affect control, we reduce the model to

$$x[t+1] = \begin{bmatrix} 1 & 0.13 \\ 0 & 1 \end{bmatrix} x[t] + \begin{bmatrix} 0.02559 \\ 0.3937 \end{bmatrix} u[t].$$

We adapt the controller from [95] as

$$u[t] = \begin{bmatrix} 0.2935 & 0.4403 \end{bmatrix} x[t-1].$$

For each benchmark, we evaluate deviation under different deadline miss handling strategies defined in Section 2.3: `Hold & Kill`, `Zero & Kill`, `Hold & Skip-Next`, and `Zero & Skip-Next`. The results in Table 2 highlight that stability analysis alone is insufficient for safety guarantees: all systems remained stable under all strategies, yet deviation bounds varied significantly. The `Hold` strategy consistently outperformed `Zero` across most benchmarks, while `Skip-Next` strategies never achieved the minimum deviation. Algorithmically, the recurrence relation method achieved the best runtime, while bounded runs provided the tightest bounds at the cost of increased computation time.



(a) One consecutive deadline miss (b) Two consecutive deadline misses

Fig. 4: Deviation bounds from Algorithm 4 [56].

4.4 Discussion

Checking Safety Properties We illustrate deviation checking using the F1Tenth Car model and controller. The controller runs every 20 ms, but other tasks may cause occasional deadline misses. We consider a safety threshold of $\mathtt{D}^{\mathtt{safe}} = 0.01$. Following the `Zero&Kill` strategy, a missed job is killed and input 0 applied.

For a weakly-hard constraint of $\overline{\langle 1 \rangle}$, the joint spectral radius bound is $0.944 < 1$, meaning that the system is stable. Figure 4a shows results for $\overline{\mathtt{Dev}\langle 1 \rangle}$ calculated using bounded runs iteration (Algorithm 4, $r = 16$). The deviation fluctuates periodically, with a maximum value of 0.0070 at $t = 22$. Since $\overline{\mathtt{Dev}\langle 1 \rangle} < \mathtt{D}^{\mathtt{safe}}$, the safety property holds under one-miss constraints.

For a weakly-hard constraint of $\overline{\langle 2 \rangle}$—allowing one additional consecutive deadline miss—the joint spectral radius rises to $0.959 < 1$, meaning that stability remains. However, bounded runs iteration now yields deviation $\overline{\mathtt{Dev}\langle 2 \rangle} = 0.0169$ at $t = 23$ (Figure 4b), exceeding the safety threshold $\mathtt{D}^{\mathtt{safe}} = 0.01$. The system must then be redesigned or scheduled differently, though our bounds are conservative and may overestimate true deviation.

This example shows that stability does not guarantee safety: timing uncertainty can yield unsafe deviations. Our methods provide systematic deviation bounds, guiding design under weakly-hard constraints.

Scalability: To further illustrate the value of our methods, we evaluate their scalability across a variety of situations. We first compare runtimes of the algorithms across different plant models and miss strategies, then examine how runtime grows with longer horizons and more behaviors in the finite automaton. Finally, we analyze the bounded runs iteration method in detail, focusing on how its per-tree run length parameter affects runtime and tightness of the bound.

Scalability of methods: We conducted experiments across the plant models introduced in Section 4.3, fixing the initial state as $x_1 = 10$, $x_2 = 10$, and the rest zero. Each run used the $\overline{\langle 3 \rangle}$ constraint and a horizon $H = 150$. The ULS method (Algorithm 1) converged only for the RC Network under `Zero&Kill`, diverging elsewhere but always within 36.5 s. The recurrence relation method (Algorithm 2) was more robust, producing results for the RC Network under

all strategies, yet diverged for other models. Only the bounded runs iteration method (Algorithm 4) consistently produced bounds, thanks to its tunable run length parameter γ. Some configurations required such a large γ that analysis exceeded the one-hour limit, but in all other cases a finite bound was found, with the chosen γ reported in parentheses.

Runtime growth with time horizon H: To study runtime scaling, we ran each algorithm on the RC Network with up to three consecutive misses over horizons of 100, 300, and 1000 steps (Table 3). Both Algorithm 2 and Algorithm 4 scaled linearly with horizon length, as expected. By contrast, Algorithm 1 exhibited exponential growth due to repeated calls to the *forward* function and failed to complete 1000-step runs under `Skip-Next` because of floating-point overflow from divergent bounds. For practical use, the other two algorithms are thus more attractive from a runtime perspective.

Number of allowable behaviors: The number of behaviors allowed by the finite automaton grows with weaker constraints, e.g., allowing more consecutive misses. Since Algorithm 2 and Algorithm 4 enumerate these behaviors explicitly, their runtime increases with this parameter, as confirmed in Table 4 for the RC Network under `Hold&Skip-Next` with $\overline{\langle 2 \rangle}$, $\overline{\langle 4 \rangle}$, $\overline{\langle 8 \rangle}$, and $\overline{\langle 16 \rangle}$. By contrast, Algorithm 1 is unaffected, as it overapproximates the entire automaton. Both explicit methods scaled similarly in practice.

Varying run length γ: The bounded runs iteration method introduces a tunable parameter γ, the number of steps between bounding box overapproximations of reachable sets. Since its subroutine Algorithm 3 grows exponentially in γ, larger values trade higher runtime for tighter bounds. In practice, the smallest feasible γ is often preferable: this strategy was used for the Electric Steering, Aircraft Pitch, and F1Tenth models, with minimal overhead from a linear search for the minimum. Conversely, there is no benefit in choosing γ larger than the time at which the maximum deviation $\overline{\mathtt{D}}$ occurs, as seen in the RC Network. In some cases, such as Electric Steering and Aircraft Pitch, larger γ was required to prevent divergence, but beyond that threshold no further benefit was gained. Finally, for the F1Tenth Car, a bound of 6.01 at time 27 exceeded $r = 20$, suggesting that larger γ could improve tightness—yet at significant computational cost. Designers must therefore balance precision and runtime, accepting conservative bounds when execution time is already high and safety requirements are met.

4.5 Summary

We introduced a framework for checking the quantitative safety metric (such as deviation `Dev`) of weakly-hard constraints $\overline{\langle m \rangle}$. By representing weakly-hard constraints with finite automata with output and using it to compute deviation metric from Section 3, we framed the maximum deviation problem and the deviation upper bound problem, then presented three approximation methods to solve them. These methods allow us to derive quantitative safety metrics for control systems under timing uncertainties, beyond the binary classification offered by stability analysis that may not give the full picture. In the next section, we build on these foundations to address the synthesis of safe schedules.

5 How to Synthesize a Safe Schedule

In Section 4, we developed techniques to verify whether a given control task is safe in the presence of timing uncertainties. We now turn to the complementary problem: how to *synthesize* a schedule that is guaranteed to be safe, in a constrained resource setting where deadline misses are unavoidable. We utilize the method of checking the quantitative safety metrics of particular weakly-hard constraints from Section 4 to synthesize safe schedules.

5.1 Problem Formulation

Consider a collection of n dynamical systems modeled as discrete-time state-space system as in Equation (3), and correspondingly n controller tasks $C_1, \ldots, C_n$ implemented as periodic real-time tasks on a shared processor. For now, we also assume that the tasks share the same period and at most $j < n$ tasks can execute in any given period due to resource constraint.

The nominal trajectory $\tau_i^{\mathtt{nom}}$ for each task is defined as the trajectory resulting from no deadline misses, i.e., $\tau^{\mathtt{nom}} = \tau(w^{\mathtt{nom}}), w^{\mathtt{nom}} = 111\ldots11$. And each dynamical system has a safety margin $\mathtt{D}_i^{\mathtt{safe}}$ around their nominal behavior requiring that $\mathtt{Dev}(\tau, \tau^{\mathtt{nom}}) \leq \mathtt{D}_i^{\mathtt{safe}}\ \forall \tau \in \mathcal{T}$, where $\mathcal{T}$ represents all possible trajectories of the system. Following the definition in Section 2.3, we are interested in first finding weakly-hard constraints of the form $\binom{m}{k}$ that satisfy the safety requirement, i.e., $\mathtt{Dev}\binom{m}{k} \leq \mathtt{D}_i^{\mathtt{safe}}$.

Problem 3 (Constraint Synthesis). Given a dynamical system with the initial state $x[0]$, a time horizon H, and the allowed maximum deviation $\mathtt{D}^{\mathtt{safe}}$ from the nominal behavior, find a set of weakly-hard constraints of the form $\binom{m}{k}$ such that the plant behavior is safe under each of these constraints, i.e., $\mathtt{Dev}\binom{m}{k} \leq \mathtt{D}^{\mathtt{safe}}$.

Given that H may be arbitrarily large, considering all possible weakly-hard constraints becomes computationally intractable. We therefore introduce a maximum window size k_{max} where $k_{max} \ll H$, and restrict our synthesis to constraints of the form $\binom{m}{k}$ satisfying $m \leq k$ and $k \leq k_{max}$. By solving problem 3, we obtain a set of weakly-hard constraints for each plant. We then address the following scheduling problem.

Problem 4 (Schedule Synthesis). Given a set of n controllers $\{C_i\}$, each with a set of weakly-hard constraints, and an implementation platform where at most $j < n$ controllers can be scheduled in each time slot, determine if a schedule exists where all the controllers can be scheduled without violating their safety constraints over the time horizon H. Furthermore, synthesize a schedule if one exists.

Problem 5 (Safe Schedule Synthesis). Given a set of n controllers $\{C_i\}_{i=1}^n$, each with a safety margin $\mathtt{D}_i^{\mathtt{safe}}$, determine whether there exists a schedule S assigning jobs to time slots such that for every controller C_i, the resulting hit/miss sequence w_i satisfies $\mathtt{Dev}(\tau(w_i), \tau_i^{\mathtt{nom}}) \leq \mathtt{D}_i^{\mathtt{safe}}$. If such a schedule exists, construct one.

5.2 Constraint Synthesis

The first step is to derive, for each controller, the set of weakly-hard constraints that guarantee safety. Formally, given integers m, k and a controller C, let $\mathcal{T}\binom{m}{k}$ denote the set of trajectories resulting from all hit/miss sequences in $L\binom{m}{k}$, the regular language defined by the $\binom{m}{k}$ constraint (Section 2.3). Define

$$\mathtt{D}\binom{m}{k} = \max_{\tau \in \mathcal{T}\binom{m}{k}} \mathtt{Dev}(\tau, \tau^{\mathtt{nom}}),$$

where $\mathtt{Dev}$ is the trajectory distance metric defined in Section 3. We say that C is safe under $\binom{m}{k}$ if $\mathtt{D}\binom{m}{k} \leq \mathtt{D}^{\mathtt{safe}}$.

Exact computation of $\mathtt{D}\binom{m}{k}$ is intractable, as it requires enumerating exponentially many hit/miss patterns. Instead, we use over-approximation algorithms introduced in Section 4.2. This yields an upper bound $\overline{\mathtt{D}}\binom{m}{k} \geq \mathtt{D}\binom{m}{k}$; if $\overline{\mathtt{D}}\binom{m}{k} \leq \mathtt{D}^{\mathtt{safe}}$, then safety is guaranteed.

Algorithm 5: Constraint synthesis for a controller C

Input: Controller C, initial state $z[0]$, horizon H, deviation bound $\mathtt{D}^{\mathtt{safe}}$, maximum window size $k_{\max}$
Output: Set of safe weakly-hard constraints

```
result ← ∅;
for k ← 2 to k_max do
    for m ← 1 to k do
        D̄ ← DeviationUB(m, k, C, z[0]);
        if D̄ ≤ D^safe then
            result ← result ∪ {(m k)};
            break;
return result;
```

The output is a minimal set of constraints $\{(m_k, k)\}$ such that if a task meets at least m_k deadlines in any k consecutive releases, its trajectory remains safe.

5.3 Automata-Based Schedule Synthesis

Having synthesized a safe constraint set for each controller, we now construct schedules that jointly satisfy them under processor capacity j. Recall from Section 2.2 that each constraint $\binom{m}{k}$ corresponds to a regular language $L\binom{m}{k}$. We construct for each controller C_i a *controller automaton* A_i that accepts the union of all $L\binom{m}{k}$ constraints synthesized for C_i. Figure 5 shows an example automaton that corresponds to the $\binom{2}{3}$ constraint.

The *scheduler automaton* A_S is then defined as the synchronous product of the A_i, restricted so that at most j jobs are scheduled per slot. An accepting run of A_S corresponds exactly to a feasible schedule. Existence of a safe schedule thus reduces to an automata emptiness check, and an explicit schedule may be obtained from any accepting run.

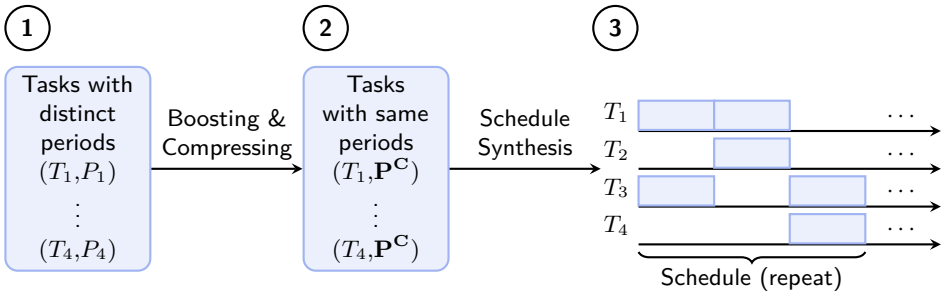


Fig. 6: Overview of the period boosting/compressing workflow.

5.4 Period Boosting and Compressing

The preceding construction assumes tasks share a common period. In practice, controllers are designed with distinct sampling periods $\{P_i\}$. To enable joint scheduling, we align them by *period boosting* (increasing P_i) or *period compressing* (decreasing P_i) [141]. This yields a common slot size P_C, within which at most j jobs may be scheduled. An overview of the procedure is shown in Figure 6. Choosing P_C involves a trade-off: shorter slots improve control responsiveness but increase contention; longer slots reduce contention but risk violating safety margins. For each candidate P_C, we discretize the plant models accordingly and re-evaluate safe constraints. In some cases, recomputing feedback gains for the new period is necessary to preserve stability; in others, original gains suffice.

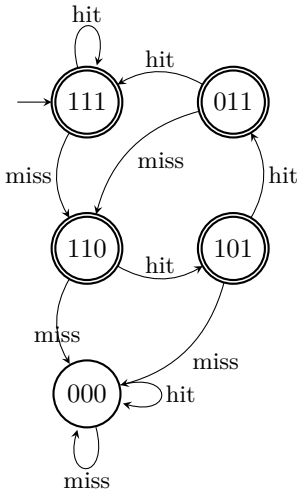


Fig. 5: Example automaton for the $\binom{2}{3}$ constraint; non-accepting states are consolidated.

5.5 Case Studies and Evaluation

We implemented the above algorithms in `Julia` and evaluated them on five benchmark controllers from the automotive domain: RC network, F1Tenth car, DC motor, suspension system, and cruise control. For each, we specified WCETs, nominal periods, and safety margins. We then:

1. Synthesized weakly-hard constraints using Algorithm 5;
2. Constructed controller automata A_i and the scheduler automaton A_S;
3. Checked existence of safe schedules under different P_C values;
4. Compared results with and without recomputing controller gains.

Table 5 summarizes the outcomes. Notably, safe schedules were synthesized even in settings where the utilization $U = \sum_i C_i/P_i$ exceeded 1, highlighting that utilization tests are insufficient in the weakly-hard regime.

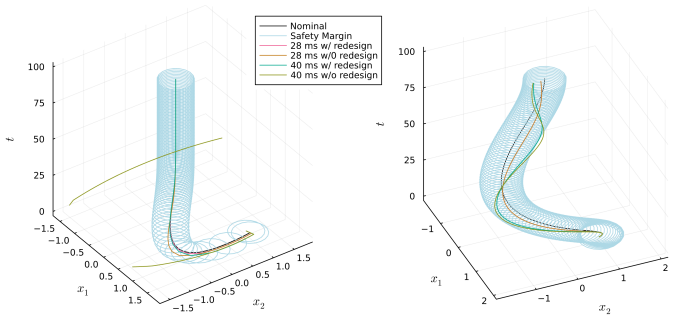


Fig. 7: Example trajectories for F1Tenth and Cruise Control models under synthesized schedules [141].

5.6 Discussion

The results highlight two key insights. First, weakly-hard constraints expand the space of feasible schedules beyond what utilization or deadline-driven analyses would allow. Second, period boosting and compressing expose non-intuitive trade-offs: while shorter periods usually enhance control quality, they can harm schedulability at the system level, and conversely, modestly longer periods can yield feasible joint schedules. These observations underscore the importance of safety-driven, rather than deadline-driven, scheduling.

5.7 Summary

We presented two complementary approaches to safe schedule synthesis. Constraint-based synthesis derives admissible deadline miss patterns and constructs schedules via automata. Period boosting and compressing harmonize task periods to enable synthesis across heterogeneous controllers. Both approaches rely fundamentally on the safety notion of bounded deviation from nominal trajectories (Section 3), linking system-level safety to task-level timing. This chapter thus establishes the methodological foundation for the probabilistic extensions explored in Section 6.

6 Statistical Hypothesis Testing for Schedule Synthesis

In Section 4, we established the safety of weakly-hard schedules by deterministically verifying all admissible hit/miss patterns up to a certain number of periods, and used over-approximation techniques to mitigate the exponential growth of computation time. While sound, this approach suffers from two key drawbacks. First, the resulting deviation bounds are often overly conservative, excluding many feasible schedules. Second, even with over-approximation, the number of trajectories to be explored still grows exponentially with the length of the horizon considered before bounding boxes are applied, making this parameter difficult to tune: values that are too small cause excessive over-approximation, while values that are too large render the computation intractable.

To overcome these limitations, we adopt a Statistical Hypothesis Testing (SHT) approach to estimate deviation upper bounds with high statistical confidence. Instead of exhaustively enumerating all possible trajectories, we sample representative executions consistent with a given weakly-hard constraint and apply statistical tests to infer, with high probability, an upper bound on the deviation. This yields a tractable and less conservative characterization of safe constraints. Finally, we integrate this statistical procedure with the schedule synthesis technique introduced in Section 5.3, generating a concrete schedule that is subsequently verified through deterministic analysis. In this way, the use of SHT improves efficiency and reduces conservatism, while the final synthesized schedule remains deterministically safe.

6.1 Statistical Hypothesis Testing Framework

We begin by recalling the deviation metric defined in Section 3.1. For a system with nominal trajectory $\tau^{\texttt{nom}} = \{\tau^{\texttt{nom}}[0], \ldots, \tau^{\texttt{nom}}[H]\}$, the deviation of an alternative trajectory $\tau = \{x[0], \ldots, x[H]\}$ is

$$\texttt{Dev}(\tau,\ \tau_{\text{nom}}) = \max_{0 \le k \le H} d(x[k],\ \tau^{\texttt{nom}}[k]), \tag{19}$$

where $d(\cdot, \cdot)$ is a metric on $\mathbb{R}^p$. Safety requires that $\texttt{Dev}(\tau, \tau_{\text{nom}}) \le d_{\text{safe}}$ for all trajectories τ induced by the schedule.

Given a weakly-hard constraint $\binom{m}{k}$, let $\mathcal{T}\binom{m}{k}$ denote the set of all trajectories consistent with that constraint. The maximum deviation associated with this constraint is

$$\texttt{D}\tbinom{m}{k} = \max_{\tau \in \mathcal{T}\binom{m}{k}} \texttt{Dev}(\tau,\ \tau_{\text{nom}}). \tag{20}$$

Checking this exactly is intractable for large H. Instead, the SHT framework estimates an upper bound $\overline{\texttt{D}}\binom{m}{k}$ such that

$$\Pr\left[\texttt{D}\tbinom{m}{k} \le \overline{\texttt{D}}\tbinom{m}{k}\right] \ge c, \tag{21}$$

for a user-specified confidence level $c \in (0, 1)$.

The SHT procedure is illustrated in Figure 8 and involves three interacting modules:

1. **Hypothesizer:** generates an initial guess $\overline{\texttt{D}}_0$ by sampling a small set of trajectories consistent with $\binom{m}{k}$ and computing their deviations.
2. **Verifier:** tests the null hypothesis

$$H_0 : \Pr\left[\texttt{Dev}(\tau, \tau_{\text{nom}}) \le \overline{\texttt{D}}\right] < c,$$
$$H_1 : \Pr\left[\texttt{Dev}(\tau, \tau_{\text{nom}}) \le \overline{\texttt{D}}\right] \ge c,$$

   using a finite sample $X = \{\tau_1, \ldots, \tau_K\}$ of trajectories. The sample size K is derived from the desired Bayes factor and confidence level.
3. **Refiner:** if the verifier rejects H_1, it provides a counterexample trajectory τ' with $\texttt{Dev}(\tau', \tau_{\text{nom}}) > \overline{\texttt{D}}$. The hypothesizer updates $\overline{\texttt{D}}$ to this new deviation and repeats.

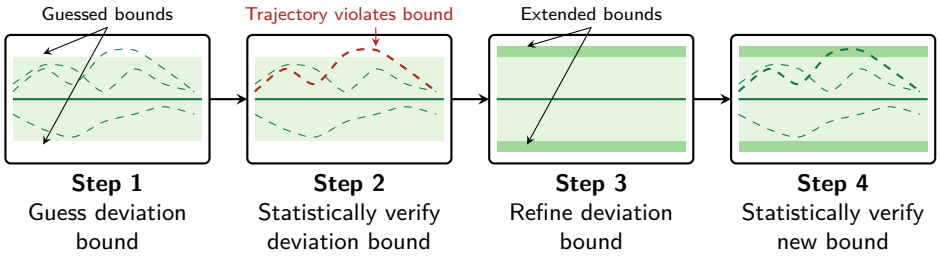


Fig. 9: Iterative deviation bound computation process.

This loop terminates when the verifier accepts H_1, yielding a bound $\overline{\mathtt{D}}\binom{m}{k}$ with confidence c.

6.2 Illustrative Example

We illustrate the method on a discrete-time control system modeled as in Section 2.1. Consider the weakly-hard constraint $(1, 3)$, meaning that in any three consecutive activations, at least one job must meet its deadline. Suppose the initial state is $x[0] = [1, 1]^T$ and the horizon is $H = 5$.

1. The hypothesizer samples two hit/miss sequences satisfying $(1, 3)$, e.g. $w = 10110$ and $w = 11001$, and computes the corresponding deviations. The maximum deviation 0.0482 is taken as the initial guess $\overline{\mathtt{D}}_0$.
2. The verifier tests H_1 with sample size K, and finds a counterexample trajectory with deviation $0.3157 > \overline{\mathtt{D}}_0$.
3. The refiner updates the guess to $\overline{\mathtt{D}}_1 = 0.3157$ and resubmits it. This time, all sampled trajectories satisfy the deviation bound, and H_1 is accepted.

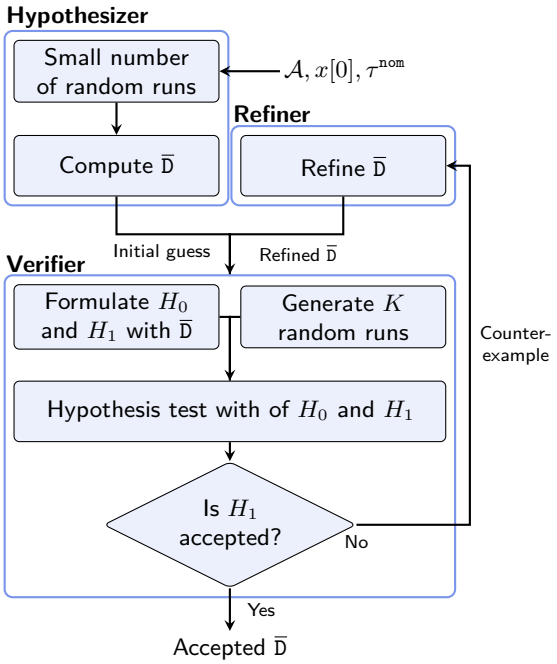


Fig. 8: Overview of the statistical hypothesis testing framework for schedule synthesis.

Thus the SHT method returns $\overline{\mathtt{D}}(1, 3) = 0.3157$ with confidence c. In this example, the bound coincides with the true maximum deviation that would also be obtained through exhaustive enumeration, but at far lower computational cost.

6.3 Integration with Schedule Synthesis

The SHT procedure integrates with the automata-based synthesis workflow introduced in Section 5.3. For each task T_i with safety margin $\mathtt{D}_i^{\mathtt{safe}}$:

1. Enumerate candidate weakly-hard constraints $\binom{m}{k}$ up to a maximum window size k_{max}.
2. Apply the SHT procedure to compute $\overline{\mathtt{D}}_i\binom{m}{k}$ for each constraint.
3. Collect the set of constraints deemed safe:

$$\mathcal{C}_i = \{\binom{m}{k} : \overline{\mathtt{D}}_i\binom{m}{k} \leq \mathtt{D}_i^{\mathtt{safe}}\}.$$

4. Construct a task automaton A_i encoding all hit/miss patterns satisfying at least one constraint in $\mathcal{C}_i$, following Section 2.2.

The scheduler automaton is then built as the synchronous product of the task automata, subject to the resource constraint that at most J tasks can execute in each slot. An accepting run of this automaton corresponds to a candidate schedule.

Finally, each synthesized schedule is subject to a deterministic *sanity check*: the exact trajectory τ_i induced for each task is computed, and its deviation from the nominal trajectory is verified against $\mathtt{D}_i^{\mathtt{safe}}$. This ensures that, despite relying on probabilistic guarantees at the constraint-checking stage, the final synthesized schedule is deterministically safe. This mirrors the philosophy of Section 4, but applied only to a small number of candidate schedules rather than the entire exponential space.

6.4 Deterministic Verification

The statistical hypothesis testing (SHT) procedure described above provides only a *probabilistic* guarantee that a weakly-hard constraint $\binom{m}{k}$ is safe for a given controller, i.e., that the maximum deviation $\mathtt{D}\binom{m}{k}$ does not exceed the prescribed safety margin d_{safe} with high confidence c. At first glance, this may appear to compromise the goal of obtaining deterministic safety guarantees for synthesized schedules. However, because of the end result of schedule synthesis in Section 6.3 are concrete schedules, we can still obtain deterministic safety guarantees even though we employed statistical hypothesis testing.

The key point is that weakly-hard constraints are used *only as an intermediate representation* in the synthesis process. Once a candidate schedule has been generated from these constraints, its safety can be assessed deterministically. Indeed, a concrete schedule $w = \{w[0], \ldots, w[H]\}$ induces a unique trajectory $\tau^w = \{x[0], \ldots, x[H]\}$ for each controller, as defined in Sections 2.1 and 2.3. The deviation of this trajectory from the nominal trajectory τ_{nom} is then computed exactly via

$$\mathtt{Dev}(w) = \max_{0 \leq k \leq H} d(x[k],\ \tau^{\mathtt{nom}}[k]), \tag{22}$$

where $d(\cdot, \cdot)$ is the state-space distance metric introduced in Section 3.1. The schedule w is verified to be *deterministically safe* if and only if

$$\mathtt{Dev}(w) \leq D^{\text{safe}}.$$

This two-phase process yields the best of both worlds:

- **Efficiency and tightness:** SHT is used to identify sets of weakly-hard constraints that are *likely* to be safe, drastically reducing the search space compared to exhaustive deterministic checking.
- **Deterministic guarantee:** Each synthesized schedule is subjected to exact deviation computation, ensuring that the final result—the concrete schedule—is safe with certainty.

Formally, let $\mathcal{C}_i$ denote the collection of safe weakly-hard constraints for task T_i identified via SHT, and let $\mathcal{S}$ denote the set of candidate schedules generated from the product automaton construction in Section 6. The deterministic verification phase selects

$$\mathcal{S}_{\text{safe}} = \{w \in \mathcal{S} \mid \texttt{Dev}(w) \leq \texttt{D}_i^{\texttt{safe}} \text{ for all tasks } T_i\}.$$

Thus, while the intermediate step of constraint classification relies on statistical confidence, the end product of the synthesis process is a deterministically certified schedule.

This approach highlights an important advantage of the statistical hypothesis testing approach for schedule synthesis: *statistics guides the search, but determinism certifies the result.* By using SHT to prune infeasible constraints early, we avoid the exponential blow-up and over-approximation of purely deterministic methods (Section 4), yet we retain the same level of assurance in the final synthesized schedule.

6.5 Case Studies and Evaluation

We evaluate the statistical hypothesis testing approach on five automotive control systems sharing a computational platform, addressing two research questions:

1. Can the SHT-based approach synthesize safe schedules when deterministic methods fail?
2. How does the computational efficiency compare to existing deterministic methods?

Benchmarks We evaluate on five automotive control benchmarks. All systems are discretized with period $P = 20\,\text{ms}$, and controllers are designed using LQR assuming a one-period sensor-to-actuator delay.

RC Network (RC) The first benchmark is a resistor-capacitor network [34]. The continuous-time plant model is

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

F1Tenth Car (F1) The second benchmark is an F1Tenth autonomous racing car [99]. We linearize the bicycle model around straight-line motion, yielding

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

The next three benchmarks are automotive subsystems from [105].

DC Motor (DC) The third benchmark is a DC motor speed controller [127]:

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

Car Suspension (CS) The fourth benchmark is a suspension system [119]:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -8 & -4 & 8 & 4 \\ 0 & 0 & 0 & 1 \\ 80 & 40 & -160 & -60 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 80 \\ 20 \\ -1120 \end{bmatrix} u(t).$$

Cruise Control (CC) The fifth benchmark is a cruise control system [100]:

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

Experimental Setup We compare the SHT-based approach against the deterministic method from [140]. The control objective for all five systems is to stabilize from an offset initial state x_0 to the origin. Initial states and safety margins are specified in Table 6. All five controller tasks execute on a shared processor, with a resource constraint of $j = 2$, meaning only two tasks can execute per period $P = 20\,\text{ms}$. This constraint models realistic automotive scheduling scenarios similar to AUTOSAR Adaptive [140].

All experiments use Julia 1.8 with the following parameters: maximum weakly-hard window size $k_{max} = 6$, Bayes factor threshold $B = 4.15 \times 10^5$, time horizon $H = 100$, confidence level $c = 0.99$, and Euclidean distance for deviation measurement.

These parameter values enable direct comparison with the deterministic baseline [140]. While modest ($k_{max} = 6$, $n = 5$), they cover practically relevant scenarios—window size 6 captures extreme cases like 1 hit in every 6 consecutive invocations. Constraint synthesis dominates computational cost and scales linearly with n, so larger n would yield proportional runtime increases without fundamentally changing the comparison.

RQ1: Synthesis Effectiveness: We evaluate whether the SHT-based approach can synthesize safe schedules when the deterministic method fails. We apply both methods to the benchmarks, with the deterministic method using iteration parameter $n = 15$. **Result**: the SHT-based method successfully synthesizes a safe schedule (Figure 10), while the deterministic method fails.

Tables 7 and 8 provide detailed constraint synthesis and schedule verification results.

Table 7 shows deviation upper bounds $\widehat{\mathtt{D}}\binom{m}{k}$ computed by the SHT-based method for each weakly-hard constraint $\binom{m}{k}$. Constraints satisfying $\widehat{\mathtt{D}}\binom{m}{k} < \mathtt{D}^{\mathtt{safe}}$ are deemed safe. Highlighting indicates: ■ safe by both methods ($\widehat{\mathtt{D}}, \overline{\mathtt{D}} < \mathtt{D}^{\mathtt{safe}}$); ■

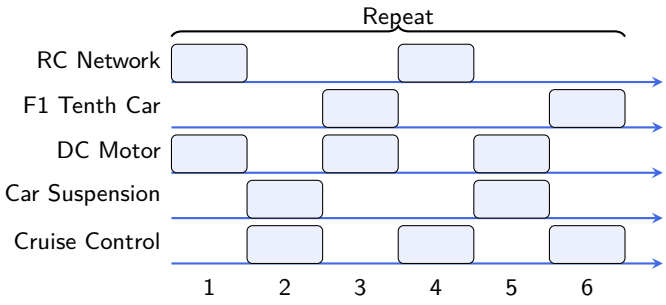


Fig. 10: The *deterministic* schedule synthesized for the benchmarks in Section 6.5 [142].

safe only by SHT method ($\widehat{\mathtt{D}} \leq \mathtt{D}^{\mathtt{safe}} \leq \tilde{d}$). By construction, any constraint safe under the deterministic method is also safe under the SHT method. The methods perform comparably for RC Network and DC Motor, but the SHT method produces significantly tighter bounds for F1 Tenth Car, Car Suspension, and Cruise Control, with Car Suspension and Cruise Control gaining many additional safe constraints.

Table 8 shows exact deviations for each benchmark under the synthesized schedule (Figure 10), computed via deterministic verification (Section 6.4). Each system's schedule matches a safe weakly-hard constraint; for example, RC Network's schedule corresponds to $\left(\frac{1}{3}\right)$. For two benchmarks (Car Suspension and Cruise Control), the deterministic method's estimates $\overline{\mathtt{D}}$ exceed the safety margin $\mathtt{D}^{\mathtt{safe}}$, while the SHT method's estimates remain within bounds and closer to actual deviations. This aligns with Table 7, confirming that the deterministic method over-approximates severely for these systems. The synthesized schedule passed verification on the first attempt, as expected given the high confidence level $c = 0.99$. These results demonstrate that the SHT-based approach successfully produces *deterministically* safe schedules even when existing deterministic methods fail.

RQ2: Computational Efficiency: To assess computational efficiency, we compare runtime against the deterministic method with two configurations: $n = 15$ (used in RQ1) and $n = 18$ (the minimum n for which the deterministic method succeeds). Higher n yields tighter bounds but exponentially longer runtime [56]. Runtime results appear in Table 9.

The SHT-based method achieves 30× to 600× speedup per benchmark for constraint synthesis, and overall speedup of 55× (vs. $n = 15$) and 394× (vs. $n = 18$). Schedule synthesis and verification contribute negligible overhead. The SHT method exhibits consistent per-benchmark runtimes, while the deterministic method shows high variance (Car Suspension takes significantly longer). Critically, the deterministic method requires trial-and-error tuning of n, itself a time-consuming process, whereas the SHT method eliminates this parameter search.

In summary, the SHT-based approach is orders of magnitude faster while avoiding manual parameter tuning required by deterministic methods.

The use of SHT offers several advantages. First, sampling-based verification yields tighter bounds, with upper bounds $\overline{\mathsf{D}}\binom{m}{k}$ that are typically much closer to the true deviation than conservative reachable-set methods of Section 4. Second, the approach is more scalable since the computational cost grows linearly both with the number of samples and the time horizon H, rather than exponentially with H. Finally, SHT enhances feasibility by allowing schedules excluded by deterministic over-approximation to be recovered, thereby broadening the design space. However, the approach also has certain limitations. The probabilistic nature of SHT means that constraint safety is guaranteed only with confidence c, though the final deterministic verification step mitigates this risk. Additionally, parameter tuning presents challenges, as sample size K and Bayes factor thresholds affect both accuracy and computational effort, and must be chosen carefully.

6.6 Summary

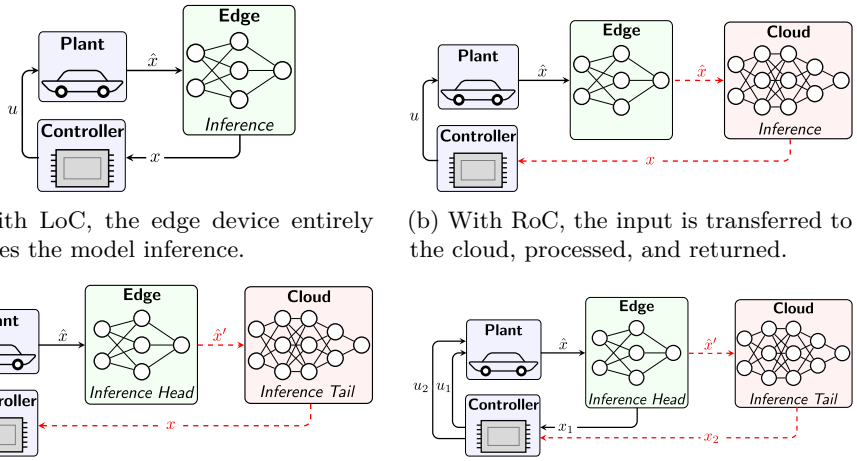
In summary, statistical hypothesis testing provides a principled, scalable, and less conservative alternative to the deterministic safety checking methods of Section 4. By evaluating safety in terms of the deviation metric defined in Section 3.1, it aligns with our quantitative system-level safety framework. When integrated with automata-based schedule synthesis (Section 2.2, Section 2.3), SHT enables the construction of schedules that are deterministically safe yet efficiently discovered. This combination of probabilistic reasoning and deterministic validation forms the backbone of our statistical approach to safe schedule synthesis.

7 Safety-Driven Edge-Cloud Partitioning

In the previous sections, we considered implementation imperfections in the form of timing uncertainties, such as the inability to meet 100% of deadlines. We now turn to another major source of imperfection: inaccuracies introduced by neural network estimators. Although advances in deep learning have dramatically improved estimation accuracy, these gains often come with larger models and higher computational demands. On resource-constrained autonomous platforms—limited by space, energy, and cost—it is rarely feasible to deploy the most accurate models solely on edge devices. Cloud-based deployment removes these constraints and enables more accurate inference but introduces significant sensor-to-actuator delays due to communication and computation overhead.

Split Computing (SC) mitigates these drawbacks by partitioning the DNN into an edge "head" that produces fast but less reliable estimates, and a cloud "tail" that yields more accurate but delayed results. This raises fundamental design questions: how should control strategies combine these heterogeneous inferences, and how should computational resources and DNN sizes be allocated across edge and cloud?

We address these questions by providing the mathematical formulation of a hybrid controller, which combines the delayed inputs from both the edge and the cloud, taking into account the latency between sensing the state, and the time at which this control input is produced by the DNNs at the edge and cloud.



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

(b) With RoC, the input is transferred to the cloud, processed, and returned.

(c) With SC, computation is split; compression reduces bandwidth.

(d) With SC and early-exit, edge and cloud share work with intermediate results.

Fig. 11: Different learning-enabled CPS architectures.

Our results show that purely edge- or cloud-based solutions lead to significant deviations from the ideal behavior and increase control effort, whereas SC reduces both. However, performance is highly sensitive to the edge–cloud resource split, motivating systematic exploration of this design space and the development of tailored control strategy templates.

7.1 Split Computing for Edge-Cloud Platforms

Over the past decade, DNNs have achieved remarkable accuracy and are increasingly deployed on edge devices (Local-only Computing (LoC)) [15]. However, resource constraints limit edge deployment, often requiring compression techniques that reduce accuracy. Alternatively, cloud-based Remote-only Computing (RoC) offloads all computations to a server, achieving high accuracy but incurring latency and bandwidth costs.

SC offers a compromise by partitioning the DNN between edge and cloud, reducing delay and transmission overhead. Early SC methods split models at fixed layers; recent approaches optimize latency and communication. Any architecture using a DNN, denoted $\mathcal{M}(\cdot)$, produces output y from input x. We compare LoC, RoC, and SC, and discuss SC in Cyber-Physical Systems (CPSs), where both performance and safety are critical.

– Local-only Computing (LoC): Here, the edge device executes $\mathcal{M}(x)$ entirely (Figure 11a), minimizing latency but limiting model size. Lightweight variants $\bar{\mathcal{M}}(x)$ (e.g., MobileNetV3 [58]) and compression methods such as pruning, quantization [70], or knowledge distillation [53] improve efficiency, albeit with potential accuracy loss.

- Remote-only Computing (RoC): In RoC, x is transmitted to the cloud for processing (Figure 11b). This yields high accuracy but incurs significant communication delay and bandwidth usage.
- Split Computing (SC): In SC, the DNN is divided into a local "head" and remote "tail" (Figure 11c). Data may be compressed with auto-encoders [85], where the encoder $\mathcal{F}(x) = z_l$ runs locally and the decoder $\mathcal{G}(z_l) = \bar{x}$ runs on the cloud, with distortion $d(x, \bar{x})$. Early studies [63] suggested splitting after initial layers to balance latency and energy. Later methods exploit quantization [68], sparsity [13], lossy/lossless compression [16, 29, 86].

Split-point selection evolved from layer-based heuristics [116] to neuron-based methods [12,31], which preserve influential neurons before partitioning. Extensions incorporate Multi-Task Learning (MTL) [14], enabling simultaneous tasks and improved generalization. In LoC (Figure 11a), limited edge resources restrict DNN size, increasing estimation error and reachable set size, thus reducing safety. In SC (Figure 11c), larger cloud models improve accuracy but add sensor–actuator delay and risk of packet loss, again affecting safety. Early-exit variants (Figure 11d) compute a quick local control u_1 from estimate x_1, while cloud inference refines with u_2, reducing latency. Importantly, the most accurate DNN is not necessarily the safest in SC due to delay. Thus, careful split-point selection is essential for control design. We illustrate SC for pedestrian distance estimation in autonomous driving. The split DNN processes camera frames at 30 fps:

1. Image acquisition: frames are resized, cropped, and normalized.
2. Preprocessing: YOLO [60] detects pedestrians and bounding boxes, passed to the head network.
3. Head computation: bounding box features are transformed into intermediate representations, enabling fast but uncertain estimates, then transmitted.
4. Tail computation: the tail refines features for precise distance regression, outputting pedestrian–vehicle distances for navigation and control.

7.2 Hybrid Controller Design for Sub-Sample Cloud Delays

We design a controller for SC architectures. Sensors capture the environment; a DNN estimates state variables, which inform control inputs. Examples include pedestrian distance estimation or adaptive cruise control. The key challenge is balancing inference accuracy against delay. We propose deploying a small DNN_E on the edge and a larger DNN_C on the cloud. DNN_E provides fast but uncertain estimates; DNN_C refines accuracy after communication delay. The controller applies early control using DNN_E outputs and updates when DNN_C results arrive.

System setup: Two DNNs run in parallel: DNN_E locally and DNN_C remotely. Within each sampling period h, three control inputs are applied (Figure 12): (i) $u_C[k-1]$ (previous cloud output) until DNN_E finishes, (ii) $u_E[k]$ (current edge output) until DNN_C finishes, (iii) $u_C[k]$ (current cloud output) thereafter.
Delays are D_E for DNN_E and D_C for DNN_C, both assumed $< h$.

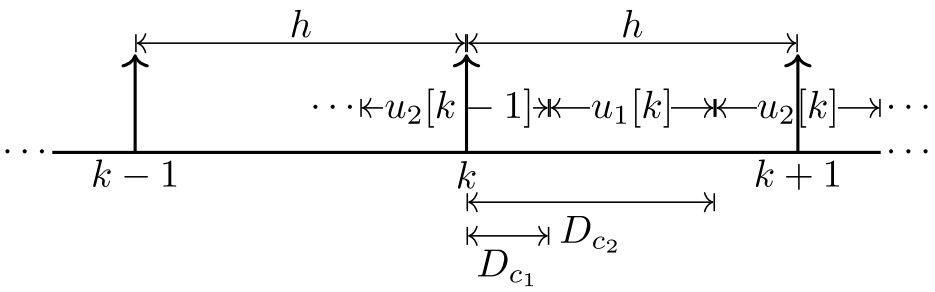


Fig. 12: Control inputs (i) $u_2[k-1]$, (ii) $u_1[k]$, and (iii) $u_2[k]$ [149].

State-space model Let $x[k] \in \mathbb{R}^n$ be the plant state, $u[k] \in \mathbb{R}^m$ the control input. The discrete-time system is:

$$x[k+1] = Ax[k] + Bu[k]. \tag{23}$$

With split delays, $x[k+1]$ becomes:

$$x[k+1] = Ax[k] + \Gamma_1(D_E, D_C)u_E[k] + \Gamma_2(D_C)u_C[k] + \Gamma_3(D_E, D_C)u_C[k-1],$$

with

$$\Gamma_1(D_E, D_C) = \int_0^{D_C - D_E} e^{As} B\, ds,$$
$$\Gamma_2(D_C) = \int_0^{h-D_C} e^{As} B\, ds,$$
$$\Gamma_3(D_E, D_C) = \int_{h-D_C}^{h-D_E+D_C} e^{As} B\, ds.$$

Define augmented state and control:

$$z[k] = \begin{bmatrix} x[k] \\ u_C[k-1] \end{bmatrix}, \qquad u[k] = \begin{bmatrix} u_E[k] \\ u_C[k] \end{bmatrix}.$$

Then,

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

with

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

Controller Design We design K via Linear–Quadratic Regulator (LQR), minimizing

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

The Riccati equation yields P, and

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

Finally, estimation uncertainty is modeled by scaling the $k_{x_{ij}}$ entries of K with uncertainty factors λ_{1_i} for DNN_E and λ_{2_i} for DNN_C , simulating state estimation errors while leaving $k_{u_{ij}}$ unaffected.

7.3 Hybrid Controller Design for Multi-Sample Cloud Delays

In the previous section, we discussed the controller design strategy for edge-cloud platforms. In this section, we explain the different scenarios involved in edge-cloud computing and formulate them based on when the control inputs from the edge u_E and the cloud u_C arrive within a control period. As mentioned earlier, the control inputs u_E and u_C are derived from DNN_E and DNN_C respectively. We will start with describing the two base cases: edge-only computing and cloud-only computing, and then go on to describe various edge-cloud scenarios.

Edge-Only Control: In the edge-only computing scenario, all computations are executed on the edge device, leading to faster but less accurate control due to limited computational resources. The control input is assumed to be available within a single sampling period, with $u_E[k-1]$ applied until the updated control input $u_E[k]$ is received after a delay d_E as shown in Figure 13a. The system dynamics with sensor-to-actuator delay are expressed as

$$x[k+1] = A \cdot x[k] + \Gamma_1 \cdot u_E[k-1] + \Gamma_2 \cdot u_E[k] \tag{24}$$

$x\,[k]$ is the system state vector and the matrices Γ_1 and Γ_2 are given by

$$\Gamma_1 = \int_{h-d_E}^{h} e^{As} B \cdot ds\ , \qquad \Gamma_2 = \int_{0}^{h-d_E} e^{As} B \cdot ds\ ,$$

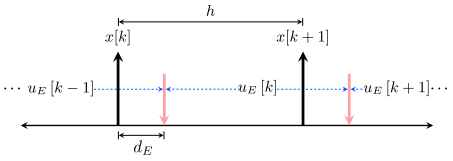
which capture the effect of the control inputs over the sampling interval. To recover a standard discrete-time state-space form, an augmented state vector

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

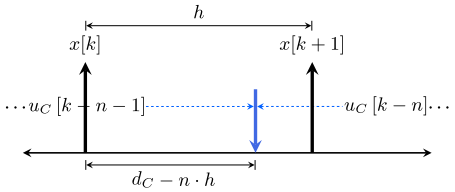
is introduced, yielding the system model

$$z[k+1] = \Phi \cdot z[k] + \Gamma \cdot u[k]\ , \quad \text{with} \quad \Phi = \begin{bmatrix} A & \Gamma_1 \\ 0 & 0 \end{bmatrix}, \quad \Gamma = \begin{bmatrix} \Gamma_2 \\ I \end{bmatrix}. \tag{25}$$

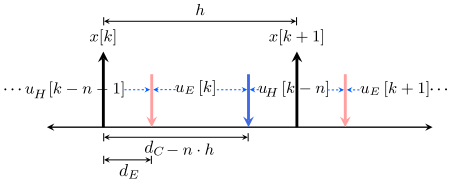
The gain matrix K is then computed via a standard LQR design, leading to the control input $u_E[k] = Kz[k]$.



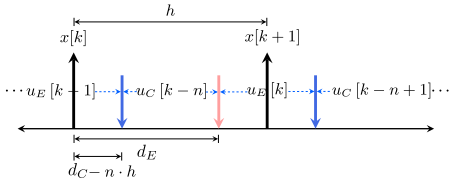
(a) Closed loop feedback is calculated solely based on the estimation from edge network.



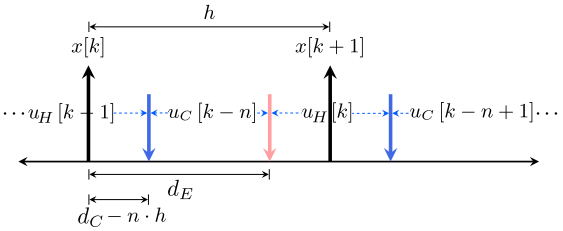
(b) Closed loop feedback is calculated solely based on the estimation from cloud network.



(c) u_C arrives after u_E and the new actuation value is the weighted sum of u_C and u_E.



(d) u_C is after u_E and overrides it.



(e) u_E arrives after u_C and the new actuation value is the weighted sum of u_C and u_E.

Fig. 13: Different scenarios arising as a result of multi-sample cloud delays [35].

Cloud-Only Control: In the cloud-only control scenario, sensed data is transmitted to the cloud for state estimation, and the resulting control input incurs a delay of $d_C \in [nh, (n+1)h]$ as shown in Figure 13b. Consequently, the control input $u_C[k-n]$ applied at the k^{th} sampling period is based on the state measured n periods earlier. The system dynamics are expressed as:

$$x[k+1] = A \cdot x[k] + \Gamma_1 \cdot u_C[k-n-1] + \Gamma_2 \cdot u_C[k-n], \tag{26}$$

where

$$\Gamma_1 = \int_{(n+1)h-d_C}^{h} e^{As} B \, ds, \qquad \Gamma_2 = \int_{0}^{(n+1)h-d_C} e^{As} B \, ds. \tag{27}$$

Defining the augmented state vector

$$z[k] = \begin{bmatrix} x[k] \\ \vdots \\ x[k-n] \\ u_C[k-n-1] \end{bmatrix}, \quad u[k] = u_C[k-n], \tag{28}$$

and the system model becomes

$$z[k+1] = \Phi \cdot z[k] + \Gamma \cdot u[k], \text{ where, } \Phi = \begin{bmatrix} A & 0 & \cdots & 0 & 0 & \Gamma_1 \\ I & 0 & \cdots & 0 & 0 & 0 \\ 0 & I & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & I & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix}, \Gamma = \begin{bmatrix} \Gamma_2 \\ 0 \\ \vdots \\ 0 \\ I \end{bmatrix}. \quad (29)$$

The control input is computed via the LQR design, yielding the gain matrix $K \in \mathbb{R}^{q \times ((n+1)\cdot p+q)}$

$$u[k] = Kz[k], \quad K = \begin{bmatrix} 0 \cdots 0 \; K_C \; 0 \end{bmatrix}. \quad (30)$$

Edge-Cloud Hybrid Control: In the hybrid control strategy, sensed data is simultaneously transmitted to both the edge and the cloud DNNs for state estimation. During each sampling interval $[k \cdot h, (k+1) \cdot h]$, two control inputs are generated: u_E from the edge, based on $\hat{x}_E[k]$, and u_C from the cloud, based on $\hat{x}_C[k-n]$. If $d_E < d_C - n \cdot h$, the edge input arrives first; otherwise, the cloud input arrives earlier. The hybrid controller determines the actuation values applied within $[k \cdot h, (k+1) \cdot h]$ according to the arrival order of these inputs.

A natural choice is to apply whichever input arrives first, as continuing with older inputs would degrade performance. For example, if u_E arrives at d_E before u_C, then u_E is applied until u_C arrives at $d_C - n \cdot h$, after which u_C is used until $(k+1) \cdot h + d_E$. At that time, the next set of actuation values, based on $x[k+1]$, becomes available, and prior inputs are discarded.

The general hybrid strategy is expressed as:

$$u_H[k] = \alpha \cdot u_E[k] + (1-\alpha) \cdot u_C[k-n], \quad (31)$$

where $\alpha \in [0, 1]$ determines the relative contribution of edge and cloud inputs. The cases $\alpha = 0$ and $\alpha = 1$ correspond to one input fully overriding the other.

In our work, α is selected empirically to optimize performance. Determining the optimal α analytically remains future work. For linear systems, such an analysis should be feasible by characterizing the uncertainties associated with the inferences from DNN_E and DNN_C.

Combined-input follows Edge-input: We first analyze the case where the edge control input arrives before the cloud input, i.e., $d_E < d_C - n \cdot h$. In this setting, u_E is applied during the interval $[k \cdot h + d_E,\ k \cdot h + (d_C - n \cdot h)]$ as shown in Figure 13c. Once u_C becomes available, a weighted combination of u_E and u_C is applied during $[k \cdot h + (d_C - n \cdot h),\ (k+1) \cdot h + d_E]$. The system dynamics are given by

$$x[k+1] = A \cdot x[k] + \Gamma_1 \cdot u_H[k-1] + \Gamma_2 \cdot u_E[k] + \Gamma_3 \cdot u_H[k], \text{where,}$$

$$\Gamma_1 = \int_{(n+1)h-d_C}^{(n+1)h-d_C+d_E} e^{As} B\, ds, \Gamma_2 = \int_0^{d_C-d_E-nh} e^{As} B\, ds, \Gamma_3 = \int_0^{(n+1)h-d_C} e^{As} B\, ds.$$

To express the system in standard state-space form, we define

$$z[k] = \begin{bmatrix} x[k] \\ \vdots \\ x[k-n] \\ u_E[k-1] \\ u_C[k-n-1] \end{bmatrix}, \quad u[k] = \begin{bmatrix} u_E[k] \\ u_C[k-n] \end{bmatrix},$$

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

$$\Phi = \begin{bmatrix} A & 0 \cdots 0 & \Gamma_1\alpha & \Gamma_1(1-\alpha) \\ I & 0 \cdots 0 & 0 & 0 \\ 0 & I \cdots 0 & 0 & 0 \\ \vdots & \vdots \ddots \vdots & \vdots & \vdots \\ 0 & 0 \cdots I & 0 & 0 \\ 0 & 0 \cdots 0 & 0 & 0 \end{bmatrix}, \quad \Gamma = \begin{bmatrix} \Gamma_2 + \Gamma_3\alpha & \Gamma_3(1-\alpha) \\ 0 & 0 \\ \vdots & \vdots \\ I & 0 \\ 0 & I \end{bmatrix}.$$

The corresponding LQR gain $K \in \mathbb{R}^{2q \times ((n+1)p+2q)}$ is then computed, with the components corresponding to $x[k], x[k+1], \cdots, x[k-n]$, $u_E[k-1]$ and $u_C[k-n-1]$ in $z[k]$:

$$K = \begin{bmatrix} K_E & 0 & \cdots & 0 & 0 \\ 0 & \cdots & K_C & 0 & 0 \end{bmatrix}.$$

As a special case, when $\alpha = 0$ (Figure 13d), the hybrid controller reduces to the setting where the cloud input simply overrides the edge input after arrival. In this case, u_E is applied during $[k \cdot h + d_E,\ k \cdot h + (d_C - n \cdot h)]$, and is replaced by u_C for $[k \cdot h + (d_C - n \cdot h),\ (k+1) \cdot h + d_E]$. The dynamics then simplify to

$$x[k+1] = A \cdot x[k] + \Gamma_1 \cdot u_C[k-n-1] + \Gamma_2 \cdot u_E[k] + \Gamma_3 \cdot u_C[k-n],$$

with corresponding augmented state representation $z[k+1] = \Phi \cdot z[k] + \Gamma \cdot u[k]$, where,

$$z[k] = \begin{bmatrix} x[k] \\ \vdots \\ x[k-n] \\ u_C[k-n-1] \end{bmatrix}, \quad u[k] = \begin{bmatrix} u_C[k-n] \\ u_E[k] \end{bmatrix}.$$

The associated gain matrix $K \in \mathbb{R}^{2q \times ((n+1)p+q)}$, with $u[k] = Kz[k]$, in this case has the form:

$$K = \begin{bmatrix} 0 & \cdots & 0 & K_C & 0 \\ K_E & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

Combined-input follows Cloud-input: We next consider the case where the cloud control input arrives before the edge control input, i.e., $d_C - n \cdot h < d_E$. In this setting, u_C is applied during the interval $[k \cdot h + (d_C - n \cdot h),\, k \cdot h + d_E]$. Once u_E becomes available, a weighted combination of u_E and u_C is applied during $[k \cdot h + d_E,\, (k+1) \cdot h + (d_C - n \cdot h)]$ as illustrated in Figure 13e.

The system dynamics take the form

$$x[k+1] = A \cdot x[k] + \Gamma_1 \cdot u_H[k-1] + \Gamma_2 \cdot u_C[k-n] + \Gamma_3 \cdot u_H[k], \text{where,}$$

$$\Gamma_1 = \int_{h-d_E}^{d_C - d_E - (n-1)h} e^{As} B\, ds, \quad \Gamma_2 = \int_0^{d_E - d_C + nh} e^{As} B\, ds, \quad \Gamma_3 = \int_0^{h-d_E} e^{As} B\, ds.$$

To cast the dynamics into standard state-space form, we define

$$z[k] = \begin{bmatrix} x[k] \\ \vdots \\ x[k-n] \\ u_E[k-1] \\ u_C[k-n-1] \end{bmatrix}, \quad u[k] = \begin{bmatrix} u_E[k] \\ u_C[k-n] \end{bmatrix},$$

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

$$\Phi = \begin{bmatrix} A & 0 \cdots 0 & \Gamma_1\alpha & \Gamma_1(1-\alpha) \\ I & 0 \cdots 0 & 0 & 0 \\ 0 & I \cdots 0 & 0 & 0 \\ \vdots & \vdots \ddots \vdots & \vdots & \vdots \\ 0 & 0 \cdots I & 0 & 0 \\ 0 & 0 \cdots 0 & 0 & 0 \end{bmatrix}, \quad \Gamma = \begin{bmatrix} \Gamma_3\alpha & \Gamma_2 + \Gamma_3(1-\alpha) \\ 0 & 0 \\ \vdots & \vdots \\ I & 0 \\ 0 & I \end{bmatrix}.$$

The gain matrix $K \in \mathbb{R}^{2q \times ((n+1)p+2q)}$ is computed from this formulation. It can be simplified into components corresponding to $x[k], x[k+1], \cdots, x[k-n]$, $u_E[k-1]$ and $u_C[k-n-1]$ in $z[k]$ as follows:

$$K = \begin{bmatrix} K_E & 0 & \cdots & 0 & 0 \\ 0 & \cdots & K_C & 0 & 0 \end{bmatrix}.$$

In this section we analyzed four distinct cases, depending on whether actuation is based on edge-only, cloud-only, or hybrid inference, and on the order in which u_E and u_C arrive within each sampling period. For each case, the primary contribution was the derivation of a state-space representation of the closed-loop system. Once this representation was established, standard LQR methods were applied to compute the corresponding control gains.

7.4 Case Studies and Evaluation

We evaluate the control strategies proposed in Section 7.3 using a case study of an F1Tenth [99] racing car model. The following subsections describe the objectives of our experiments, the experimental setup, the methodology, and the corresponding results. We also compare different combinations of edge-based and cloud-based inferences to determine which strategies result in the best performance under varying inference delays and uncertainty levels associated with DNN_E and DNN_C.

Objectives Our goal is to assess how combining control inputs from both DNN_E and DNN_C affects the safety and performance of the closed-loop system (plant + controller). The evaluation proceeds in two parts:

(A) A real-world case study using two state-of-the-art DNNs deployed on hardware platforms representing edge and cloud computing.

(B) A broader evaluation of controller performance under varying inference delays and uncertainties. This explores when hybrid control is advantageous and when edge-only or cloud-only strategies are preferable.

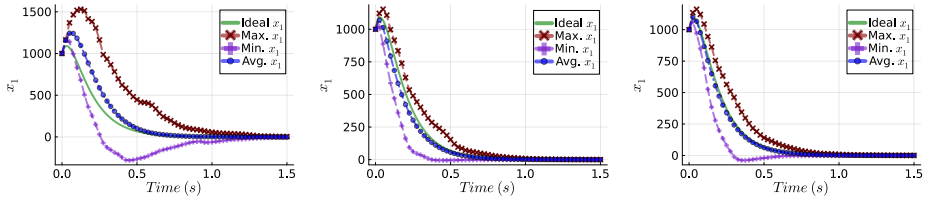
Setup: For experiment (A), the edge device is considered to be an NVIDIA Jetson Nano [97], capable of 472 GFLOP/s. The cloud platform is considered to be a GeForce RTX 5060 [96], offering a compute speed of 614 TFLOP/s. We considered `MobileNetv3_small` (DNN_E) for the edge and `EfficientNet_B6` (DNN_C) for the cloud [143]. The edge DNN is smaller and faster but incurs higher inference error of 42.53% and latency $d_E = 0.02$ s. The cloud DNN is slower with a latency of $d_C = 0.1$ s but more accurate with an error of 21.08%. Data transmission between edge and cloud adds 0.1 s round-trip delay, considering a 5G link [32].

The plant model is the state-space representation of an F1Tenth [99] racing car with two states: velocity (x_1) and yaw angle (x_2). The inputs are acceleration and steering. The controller's task is to bring the vehicle to rest from an initial velocity. Estimation uncertainties are modeled as Gaussian: $\mathcal{N}(0, \sigma_E)$ for DNN_E and $\mathcal{N}(0, \sigma_C)$ for DNN_C.

Methodology: The continuous-time plant trajectory with ground truth state knowledge is taken as the *ideal trajectory*. Discrete-time trajectories using uncertain DNN-based state estimates form the *non-ideal trajectories*.

Given $u[k] = Kz[k]$, we partition K into K_x (applied to $x[k]$) and K_u (applied to the remaining components), i.e., $K = \begin{bmatrix} K_x & K_u \end{bmatrix}$. To simulate uncertainty, we scale each state component by a random factor Λ_i, to give $\hat{x}[k] = \Lambda x[k]$, with $\Lambda = [\Lambda_1, \Lambda_2, \ldots, \Lambda_n]^T$. This modifies the effective feedback matrix as

$$K_{\text{estimation}} = \begin{bmatrix} K_x \Lambda & K_u \end{bmatrix}.$$



(a) Edge-only control (b) Cloud-only control (c) Edge-Cloud control

Fig. 14: System dynamics with combinations of edge-cloud perception [35].

Although Λ is unknown in practice, it can be empirically estimated by evaluating DNNs against labeled datasets. For simulation, the closed-loop dynamics are expressed as

$$z[k+1] = \Phi z[k] + \Lambda K z[k]. \tag{32}$$

We measure deviation from the ideal trajectory using Root Mean Square Error (RMSE):

$$\text{RMSE}(\hat{x}, \hat{x}_I) = \sqrt{\frac{1}{n}\sum_{i=0}^{n}\left(\frac{\hat{x}_i - \hat{x}_{I_i}}{\hat{x}_{I_i}}\right)^2},$$

where $\hat{x}$ and $\hat{x}_I$ denote non-ideal and ideal state values, respectively, and n is the number of sampling steps.

All strategies from Section 7.3 (edge-only, cloud-only and hybrid) were simulated. For each run, uncertainties were drawn from the respective Gaussian distribution of the DNN at sensing time, perturbing the state before computation of the control input. Each trajectory was run for 1.5 s, with 500 runs per strategy. We focus on x_1 in the results as we only consider x_1 as the observable state. The hybrid parameter α was tuned empirically to minimize RMSE, and its optimal value varied with inference delays and uncertainties.

Results: For experiment (A), Table 10 reports the average RMSE values for the different control strategies used and their standard deviations (SD). Figure 14 shows representative trajectories. A lower RMSE indicates closer tracking of the ideal trajectory, while lower SD implies more consistent performance.

The hybrid controller achieved the best performance, with average RMSE 27.38 and SD 13.20, compared to 113.57 for the edge-only controller and 41.04 for the cloud-only controller. Figure 14c shows smaller deviations of the mean non-ideal trajectory (dotted blue) from the ideal trajectory (green) with the edge-cloud controller, with narrower spread compared to edge-only (Figure 14a) and cloud-only (Figure 14b).

For experiment (B), we varied inference delays and uncertainties. Table 11 summarizes the optimal strategy under different setups. For example, when $d_E = 0.005$, $d_C = 0.02$, $\sigma_E = 0.3$, and $\sigma_C = 0.2$, cloud-only control achieved an RMSE of 21.36, outperforming hybrid with an RMSE of 24.23. However, when d_C increased to 0.04, hybrid control outperformed cloud-only control with an RMSE of 24.85 over 42.66.

Final Remarks: The best control strategy is dependent on the setup. With small cloud delays (≤ 9 sampling periods), cloud-only control is preferable. For larger delays (> 10 sampling periods), hybrid control yields lower RMSE. The inference uncertainties (σ_E and σ_C) further influence this choice. Thus, system designers can use this methodology both to select controllers and to size DNNs for edge and cloud deployments to achieve desired safety-performance trade-offs.

8 Safety-Driven GPU Partitioning

In Section 7, we examined strategies for optimizing a single controller through Split Computing (SC), where high-accuracy but high-latency cloud inference complements low-accuracy, low-latency edge inference. We now turn to the setting where cloud offloading is infeasible and all inference must be performed locally at the edge. Designers must therefore balance estimation accuracy against the available computational resources.

This challenge becomes even more pronounced when multiple DNNs are deployed on the same platform, each estimating different components of the system state. The impact of estimation errors is not uniform: certain state variables may influence system-level safety metrics (such as trajectory deviation or reachable set diameter) much more strongly than others. In such cases, allocating more resources to the DNNs responsible for these critical variables can lead to substantial safety gains, whereas treating all estimators equally may result in inefficient use of limited resources. Most existing work, however, optimizes DNNs in isolation, focusing on model-level accuracy rather than system-level safety.

In this section, we address this gap by formulating safety-driven GPU partitioning techniques. Our approach [143] explicitly accounts for the varying sensitivity of system safety to different state components. Neural networks associated with safety-critical variables are assigned larger partitions and more accurate models, while those associated with less critical variables can operate with smaller ones. System safety is then evaluated not by accuracy in isolation, but by quantitative measures introduced in Section 3—specifically, the Maximum Diameter of Reachable Sets (MDRS).

8.1 Problem Statement

This section introduces the optimal neural architecture sizing problem, the assumptions made in this study, and the methods proposed to approach the problem.

Assumptions. We make the following assumptions:

1. The plant is modeled as a p-dimensional linear system with feedback control, described by the state-space model $x[k+1] = Ax[k] + Bu[k]$, and a linear controller K with control input $u[k] = K\hat{x}[k]$.
2. The state estimate $\hat{x}[k]$ is obtained using DNNs (NNs), with each component $\hat{x}_i$ estimated by a separate NN.
3. A pool of n NNs with different cost–uncertainty trade-offs is available, and any of these NNs can be used to estimate any of the p system state components, with multiple components possibly assigned to the same NN.

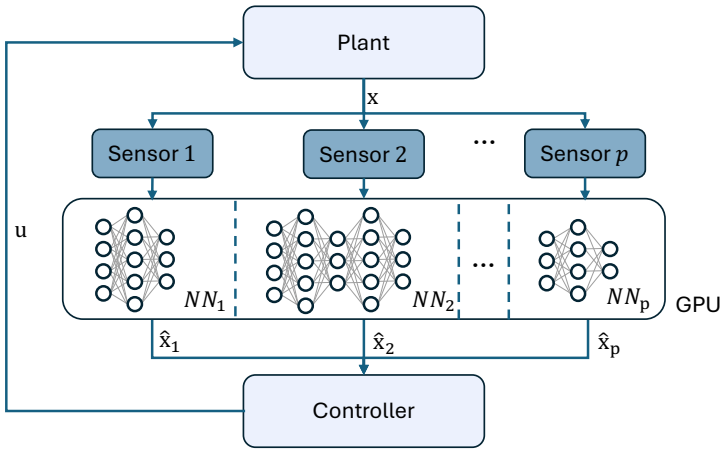


Fig. 15: Graphical illustration of neural architecture sizing problem

Assumption 3 is made for clarity of problem formulation. In practice, different state components may require different sets of candidate NNs (e.g., visual vs. radar-based estimation). The methods remain valid in this more general setting. A visual representation of the problem is provided in Figure 15.

Formal Problem. We now define the optimal neural architecture sizing problem. Let each DNN NN_j be characterized by: a cost c_j, representing computational or resource usage, and an estimation uncertainty ϵ_j, capturing the bound on its error.

The system designer must assign one NN to each of the p state components while ensuring that the overall cost does not exceed a given budget C. System performance is quantified by the Maximum Diameter of Reachable Sets (MDRS) `Diam` as defined in Section 3.2.

Problem 6. Given:

1. A p-dimensional linear system defined by (A, B), feedback gain K, initial set $X[0]$, and horizon H,
2. n DNNs with costs $c_1, \ldots, c_n$ and uncertainties $\epsilon_1, \ldots, \epsilon_n$,

find an assignment $J = \{NN_1, \ldots, NN_p\}$, where $NN_i \in \{1, \ldots, n\}$, such that:

$$\min_J \texttt{Diam}(J) \tag{33}$$

$$\text{s.t.} \sum_{i=1}^{p} c_i \leq C. \tag{34}$$

8.2 Diameter Calculation with Perception Errors

State estimation error is expressed in

$$\hat{x}_i[k] \in [x_i[k] - e_i,\ x_i[k] + e_i], \tag{35}$$

where $e = (e_1, \ldots, e_p) \in (\mathbb{R}^+)^p$ is the vector of error bounds.

This uncertainty can be expressed using set operations. Specifically, let E denote the uncertainty bounding set, which we represent as a Zonotope. Then the estimated state can be written as

$$\hat{x}[k] \in x[k] \oplus E, \tag{36}$$

where $\oplus$ is the Minkowski sum operator.

Substituting this uncertain state estimate into the feedback control law Equation (4), the system dynamics evolve as

$$\begin{aligned} x[k+1] &\in Ax[k] \oplus BK\hat{x}[k] \\ &= Ax[k] \oplus BK(x[k] \oplus E) \\ &= (A + BK)x[k] \oplus BKE. \end{aligned} \tag{37}$$

Thus, the reachable sets for the uncertain closed-loop system can be computed by iterating Equation (37).

8.3 Dynamic Programming-Based Heuristic

The first heuristic for assigning GPU resources among DNN estimators is a dynamic programming (DP)-based strategy. While the term "dynamic programming" is used, it is important to emphasize that this approach does not guarantee optimality. The reason is that the closed-loop behavior of control systems cannot be predicted solely from the uncertainty of the NNs used for state estimation.

We begin by assuming that the candidate DNN NN_i are sorted in increasing order of cost. The algorithm initializes by assigning the lowest-cost NN to each state variable, i.e.,

$$J \leftarrow [1, 1, \ldots, 1],$$

and running a reachability analysis to determine the resulting maximum diameter E of the reachable set.

Subsequent iterations expand the allocation space systematically. At each step, for every feasible allocation vector J, we generate new allocations by incrementing one element of J. For example, from $[2, 3, 3]$ the algorithm generates $[3, 3, 3]$, $[2, 4, 3]$, and $[2, 3, 4]$. For each new allocation, the maximum reachable set diameter $\texttt{Diam}(J)$ is computed.

To control the search complexity, we prune dominated solutions. Specifically, a solution J_1 is said to be dominated by another solution J_2 if J_1 has both higher cost and a larger reachable set diameter:

$$\text{cost}(J_1) > \text{cost}(J_2) \wedge \texttt{Diam}(J_1) > \texttt{Diam}(J_2). \tag{38}$$

All dominated solutions are removed before proceeding to the next iteration. This aggressive pruning ensures that only potentially useful allocations are retained.

The heuristic continues until either the maximum-cost allocation

$$J = [n, n, \ldots, n]$$

Algorithm 6: Dynamic programming heuristic

Data: A, B, K, X_0, H, $c_1, \ldots, c_n$, $\epsilon_1, \ldots, \epsilon_n$
Result: List of $\langle J, \texttt{Diam}(J), \text{cost}(J) \rangle$

$J \leftarrow$ vector of p ones;
Add J to current solutions and tried solutions;
repeat
  foreach $J \in$ *current solutions* **do**
    Add $\langle J, \texttt{Diam}(J), \text{cost}(J) \rangle$ to results;
    foreach $i \in 1 \ldots p$ **do**
      $J' \leftarrow J$ incremented at position i;
      if $J' \in$ *tried solutions* **then**
        continue
      Add J' to next solutions and tried solutions;
  foreach $J \in$ *next solutions* **do**
    Prune J if dominated (Equation (38))
  current solutions $\leftarrow$ next solutions;
  next solutions $\leftarrow \emptyset$;
until *current solutions* $= \emptyset$;
return *results*

has been evaluated or all candidate solutions are pruned in a given iteration. The procedure is summarized in Algorithm 6.

While this DP-based heuristic systematically explores the allocation space and prunes suboptimal solutions, it still requires evaluating a non-negligible number of assignments. This makes the method relatively slow for high-dimensional systems. To address this, the next section introduces a faster heuristic that trades off optimality for reduced computational burden.

8.4 Greedy Fast-Iterative Heuristic

The dynamic programming heuristic of Section 8.3 systematically explores the allocation space, but may still require evaluating many non-optimal assignments. To further reduce computational complexity, we introduce a greedy fast-iterative heuristic that quickly estimates the cost–performance trade-off by exploring only a small portion of the solution space.

Instead of incrementing resources across all state components in a structured manner, this heuristic increases resources for one state component at a time, following a predetermined priority order. For example, if the priority order is $[1, 3, 2]$, then the sequence of explored allocations is

$$[1,1,1] \;\rightarrow\; [2,1,1] \;\rightarrow\; [2,1,2] \;\rightarrow\; [2,2,2] \;\rightarrow \cdots,$$

until the most expensive option is reached for all state variables. In this way, the resources allocated across state components remain nearly uniform, and the explored solutions tend to approximate the optimal trade-off surface.

Algorithm 7: Greedy fast-iterative heuristic

Data: A, B, K, X_1, H, $c_1, \cdots, c_n$, $\epsilon_1, \cdots, \epsilon_n$, *order*
Result: List of $\langle J, \texttt{Diam}(J), \text{cost}(J)\rangle$

```
J ← vector of p ones;
repeat
    Add ⟨J, Diam(J), cost(J)⟩ to results;
    Increment the next element of J following order;
until J > vector of p n's;
return results
```

While the priority *order* can be arbitrary, we adopt a sensitivity-based ordering to guide resource allocation. Specifically, state components with higher sensitivity values (see Section 8.5) are assigned higher priority. This ensures that limited computational resources are allocated first to the most critical state variables.

The greedy heuristic offers several practical advantages:

1. It explores a much smaller portion of the solution space compared to the dynamic programming heuristic.
2. Because solution costs increase monotonically, the algorithm can be easily adapted to terminate at the closest allocation under a budget constraint, avoiding unnecessary reachability computations.
3. The method is embarrassingly parallel, as each reachability computation is independent and can be distributed across available processing resources.

The pseudocode for the heuristic is shown in Algorithm 7. Its computational complexity is $O(np)$, polynomial in both the number of states p and the number of DNNs n, making it well suited for high-dimensional systems.

8.5 Sensitivity Analysis-Based Heuristic

A key idea for reducing the search space in resource assignment is to measure how estimation errors in each state variable affect the closed-loop dynamics. To this end, we define the *sensitivity* of a state component x_i as the multiplicative effect of its estimation error on the system trajectory. Intuitively, states with higher sensitivity values have a stronger influence on the reachable set, while states with lower values have a weaker influence.

Given sensitivity values for all state components, the heuristic proceeds as follows: for a given budget C, and the uncertainty–cost trade-offs (ϵ_i, c_i) of the available DNNs, we (i) select candidate assignments of networks to states that minimize the weighted uncertainty (sensitivity $\times$ error bound), and (ii) perform only a single reachability analysis to evaluate the corresponding diameter of the reachable set. This greatly reduces computational effort compared to methods that require evaluating all candidate assignments.

From System Dynamics to Sensitivity Values: The approach builds on prior work by Ghosh et al. [40], who studied the effect of perturbations in the system dynamics matrix A on reachable sets. Uncertainties at different entries of

A can cause different amounts of stretching of the reachable set. This stretching is closely related to changes in the maximum singular value of A: perturbations that induce a larger increase in the maximum singular value lead to greater expansion of the reachable set.

Formally, introducing a perturbation ϵ at index $[i, j]$ modifies A to $A + \Delta$, and the sensitivity of this entry can be quantified as the proportional increase in the largest singular value of $A + \Delta$.

We adapt this idea to handle uncertainties in *state estimation* rather than dynamics. The controller uses $\hat{x} = (I + \Lambda)x$, where $\Lambda = \text{diag}(\delta_1, \ldots, \delta_p)$ captures estimation errors. Substituting into the closed-loop model gives us

$$x[t + 1] = (A + BK\Lambda)x[t],$$

which can be viewed as a perturbed system with dynamics $\Phi' = \Phi + \Delta$, where $\Phi = A+BK$ and Δ depends on the estimation uncertainty in each component. By activating one uncertainty δ_i at a time, we can compute how much the maximum singular value increases. This provides the sensitivity score S_i for state x_i.

Sensitivity-Guided Allocation: Once sensitivity scores $\{S_i\}_{i=1}^{p}$ are obtained, the resource allocation problem is formulated as an integer program:

$$\min_{NN_1,\cdots,NN_p} \quad \sum_{i=1}^{p} \epsilon_i \cdot S_i \tag{39a}$$

$$\text{subject to} \quad \sum_{i=1}^{p} c_i \leq C \tag{39b}$$

$$NN_i \in \{1, \cdots, n\} \; \forall i. \tag{39c}$$

The objective seeks to minimize the weighted sum of uncertainties, while assigning more accurate (and typically more expensive) networks to high-sensitivity states. The budget constraint ensures that the overall resource consumption does not exceed C.

In this approach, the sensitivity values are computed only once in polynomial time and reused across all budget levels. Also, only a single reachability analysis is required for the chosen allocation, dramatically reducing computational overhead compared to dynamic programming or greedy search.

Extension to Nonlinear Systems: For nonlinear closed-loop dynamics $\dot{x} = f(x, g(\hat{x}))$, where $\hat{x} = x+\delta x$, we first expand the dynamics around the equilibrium and retain only first-order terms in δx. This yields

$$\dot{x} = \tilde{f}(x) + h(x)\delta x,$$

where $\tilde{f}$ represents the nominal dynamics and $h(x)\delta x$ captures the contribution of estimation errors. Linearizing around the operating point leads to

$$\dot{x} = (A + B\delta x)x,$$

where A and B are Jacobians of $\tilde{f}$ and h, respectively.

Sensitivity can then be quantified by examining how the largest singular value of $A + B\delta x_i$ changes when perturbing each state component individually. States that induce the largest increases are considered the most sensitive, and should be prioritized for accurate estimation.

8.6 Case Studies and Evaluation of Proposed Heuristics

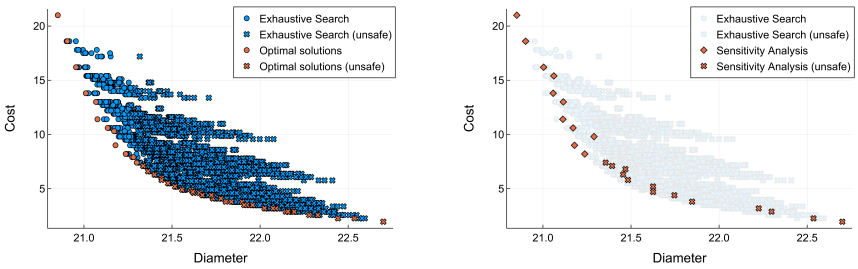
We assessed the effectiveness of the three proposed heuristics using two case studies:

1. A synthetic setup where the cost and uncertainty values are derived from the EfficientNet family [125].
2. An application-oriented setup using the Dist-YOLO model [128] for object detection and distance estimation with multiple backbone architectures.
3. A simulated autonomous racing environment, `F1TENTH Gym` [99], where

In both studies, we employed a five-dimensional numerical state-space model from the JuliaReach toolbox [8]. All heuristics were implemented in Julia and benchmarked against exhaustive search. Experiments were run on an Apple M2 CPU at 3.5 GHz in single-threaded mode.

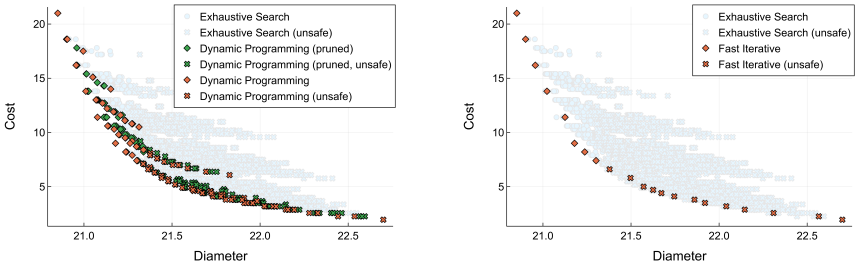
Case Study I: EfficientNet-Derived Costs and Uncertainty: As an initial demonstration, we used the accuracy–cost tradeoff of EfficientNet variants in our setting. EfficientNet offers eight configurations (B0–B7) as shown in Table 12. We measured uncertainty as $(\text{Top-5 accuracy})^{-1} - 1$ and computational cost as FLOPs. We carried out two experiments: one using the first five EfficientNet configurations, and another including all eight. For each experiment, we performed an exhaustive search across all network allocations and compared the results to those obtained by the heuristics. Metrics included the number of explored solutions, execution time, and whether the resulting system trajectories violated a safety constraint ($x_3 \leq -25$). Unsafe allocations are highlighted in Figure 16 as crosses. To ensure a fair comparison, we also evaluated the sensitivity-based heuristic across 40 equally spaced budget levels, since it only produces one solution per budget value.

We compared the three heuristics in terms of Pareto coverage, efficiency, and scalability. The dynamic programming approach explored the largest number of solutions and recovered nearly the entire Pareto front, though at higher computational cost. The fast iterative heuristic identified fewer Pareto-optimal points, but maintained strong proximity to the front while achieving substantial speedup. The sensitivity analysis method captured a moderate number of Pareto-optimal solutions, though generally less effective than the fast iterative heuristic, but it is the fastest by far when a budget is imposed. Scalability analysis further shows that exhaustive search becomes infeasible even for moderate problem sizes, while all three heuristics scale effectively, with sensitivity analysis requiring only a single point evaluation per budget. This showed that the dynamic programming and greedy heuristics are well-suited for exploring tradeoffs when no strict budget is specified. The former provides denser coverage of optimal solutions, while the



(a) Exhaustive search

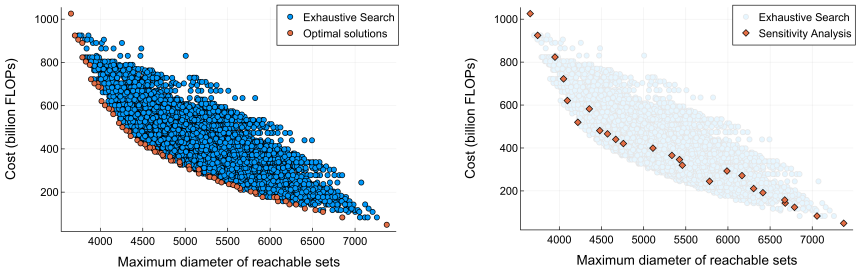
(b) Sensitivity-based heuristics



(c) Dynamic programming heuristic

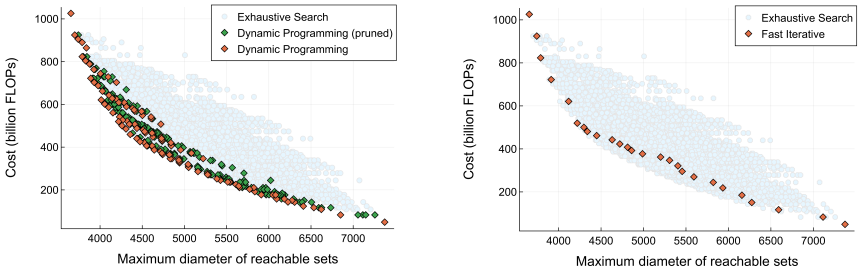
(d) Fast iterative heuristic

Fig. 16: Results of the exhaustive search and the three allocation heuristics, using cost and uncertainty values derived from Table 12 [143].



(a) Exhaustive search

(b) Sensitivity-based heuristics



(c) Dynamic programming heuristic

(d) Fast iterative heuristic

Fig. 17: Results of the exhaustive search and the three allocation heuristics, using cost and uncertainty values from Table 13 [143].

latter sacrifices coverage for efficiency. The sensitivity-based heuristic, though less accurate in capturing the Pareto surface, is the most practical when a target budget is fixed in advance.

Case Study II: Dist-YOLO with Multiple Backbones: To validate our findings in a more realistic perception setting, we extended the evaluation to the Dist-YOLO framework [128], trained on the KITTI 3D Object Detection Evaluation 2017 dataset [66]. We trained six backbone configurations, including MobileNet [59], EfficientNet [125], and Xception [30] We measured absolute relative error (ARE) in distance estimation as the uncertainty metric ϵ, and FLOPs as the cost metric c as shown in Table 13.

Compared to the EfficientNet case, uncertainty levels were higher, leading to larger reachable set diameters and in some cases diverging trajectories. This made it less meaningful to label specific outcomes as "safe" or "unsafe." The overall Pareto surface shape still remained similar to the previous case study.

As shown in Figure 17, the dynamic programming heuristic again recovered most of the Pareto-optimal allocations. The greedy heuristic performed acceptably, producing solutions close to but not consistently on the Pareto front. This discrepancy suggests that the sensitivity heuristic may be less effective when network backbones are not optimized for balanced accuracy–efficiency scaling.

Takeaway: Across both case studies, the proposed heuristics demonstrate substantial reductions in computation time relative to exhaustive search, while still capturing the key cost–performance tradeoffs. Dynamic programming offers the best Pareto coverage, greedy achieves the best efficiency, and sensitivity analysis provides a constant-time solution when only budget feasibility is required.

Case Study III: F1TENTH Gym: We evaluated [150] our sensitivity-based resource allocation strategy on `F1TENTH Gym` [99], a simulated autonomous racing platform. This platform provides reproducible vehicle dynamics suitable for multi-agent experiments. We use it to evaluate closed-loop safety.

Vehicle Dynamics. The underlying dynamics are modeled as a five-dimensional nonlinear system with states (x, y, δ, v, ψ) and control inputs (u_δ, a). Here, x and y denote the Cartesian position, δ the steering angle, v the velocity, and ψ the orientation of the vehicle. The inputs u_δ and a correspond to steering adjustment and acceleration, respectively.

State Estimation via Neural Networks. The vehicle is equipped with a 2700-ray LiDAR scan spanning $[-135°, 135°]$ at $0.1°$ resolution. From this input, we extract three states critical to system behavior: track width, lateral displacement, and orientation. To estimate these, we design three specialized convolutional DNNs (`DNN_wid`, `DNN_dis`, `DNN_ang`), each processing subsampled 1080-dimensional LiDAR inputs. Architectures range from lightweight to large-capacity networks (24–596 kB), summarized in Table 14).

Sensitivity Analysis. To prioritize resource allocation, we linearize the nonlinear feedback system and analyze changes in the maximum singular value under perturbations to individual state components. This analysis indicates that width

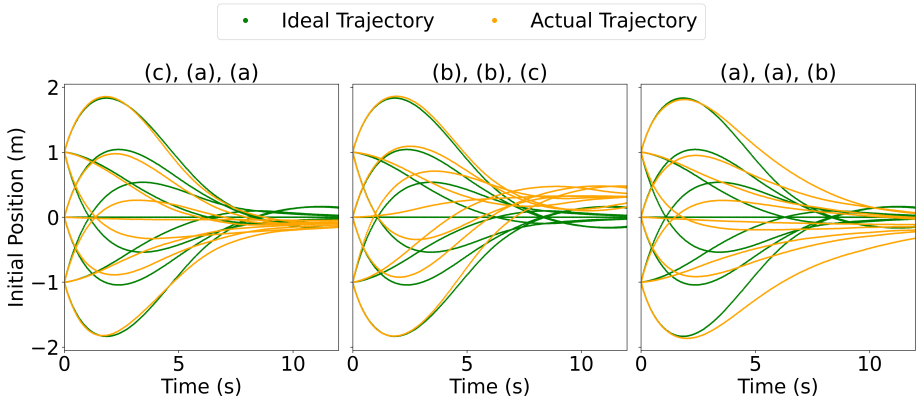


Fig. 18: Comparison of various DNN configurations over trajectories starting from 9 different initial positions with the corresponding ideal trajectory. The DNN configuration is provided with the image [150].

and lateral displacement are more influential (sensitivities 5.62 and 5.62) compared to orientation (5.28). Thus, inaccuracies in width or displacement estimates propagate more severely into the system trajectory. Sensitivity values provide an ordering among states rather than exact error magnitudes, guiding which DNNs should be allocated more capacity.

Evaluation Setup. Experiments were conducted on a 10-meter track with varying width (1–8 meters). Vehicles were initialized at seven lateral offsets (fractions of track width) and three orientations ($-\pi/3$, 0, $\pi/3$), yielding 21 initial conditions. Each configuration was simulated for 12 seconds under a controller tasked with stabilizing the vehicle at the track center. We measured: 1. *Average Deviation*, the mean Euclidean distance between actual and ideal trajectories, and 2. *Average Maximum Deviation*, the mean of maximum trajectory deviations.

Findings. Results, summarized in Table 15, validate the sensitivity-based allocation principle. Configurations that devote larger models to width and displacement (e.g., **(c)(c)(a)**) achieve the lowest deviations, while those prioritizing orientation (e.g., **(a)(c)(c)**) perform significantly worse. Furthermore, correct allocation of limited resources outperforms poorly allocated larger budgets; for instance, **(c)(a)(a)** surpasses **(a)(c)(c)** despite lower total cost. These patterns are also illustrated in trajectory comparisons (Figure 18), where the deviation from the ideal path aligns with the observed metrics.

Takeaway. The F1TENTH experiments demonstrate that sensitivity analysis provides a principled basis for allocating DNN resources. Assigning higher-capacity networks to the most sensitive state components yields substantial safety gains, while naïve allocation can degrade performance even with higher budgets.

In Section 9, we show how the bounds derived here can be interpreted in an environment-aware manner using digital-twin-based safety analysis.

9 Environment-Aware System-Level Safety via Digital Twins: The SOTERIA Framework

The previous sections of this paper develop a system-level approach to safety assurance in autonomous systems built from imperfect components. Rather than insisting on idealized assumptions—such as perfect timing or exact state estimation—the paper introduces quantitative safety metrics, including deviation from a nominal trajectory $Dev(\cdot)$ and the maximum diameter of reachable sets $Diam(\cdot)$, evaluated over a finite horizon and compared against a safety threshold D_{safe}. These metrics characterize how imperfections in timing and learning propagate through control systems.

A key remaining question is how to interpret these bounds in the context of real-world operation. In particular, the acceptability of a given deviation or uncertainty bound often depends on the physical environment in which the system operates. This section introduces *SOTERIA*, a formal digital-twin-enabled framework that addresses this question by explicitly incorporating environment modeling into system-level safety verification.

9.1 Motivation: From Bounds to Environment-Aware Safety Decisions

Throughout Sections 3–8, safety is expressed in terms of inequalities of the form:

$$Dev(\tau, \tau_{\text{nom}}) < D^{\text{safe}} \quad \text{or} \quad Diam(F) < D^{\text{safe}},$$

where τ_{nom} denotes the ideal, imperfection-free trajectory and F is the flowpipe of reachable states induced by timing or estimation uncertainty. These conditions provide necessary quantitative guarantees, but they do not specify how D^{safe} should be chosen.

In practice, the same bound on $Dev(\cdot)$ or $Diam(\cdot)$ may correspond to safe or unsafe behavior depending on external conditions such as geometry, obstacles, or operating speed. A control system experiencing a fixed end-to-end latency or estimator error may be safe in one environment yet unsafe in another, even when internal computational behavior remains unchanged. This observation motivates making the environment an explicit part of the safety model rather than treating it implicitly or pessimistically [136–138].

9.2 Overview of the SOTERIA Framework

SOTERIA [136] was developed to verify the safety of latency-aware cyber-physical systems whose worst-case timing behavior may violate classical hard real-time assumptions, yet still be safe in practice. The central idea is to separate timing analysis from physical safety verification, while linking them through a *digital twin*. Formally, SOTERIA models a system as a composition

$$M := \{M_W, M_S, M_C, M_P\},$$

where M_W is the workload model, M_S the scheduler model, M_C the controller, and M_P the physical dynamics. In addition, SOTERIA introduces an explicit *environment model* $\mathcal{E}$, capturing the operational context in which the physical system evolves.

Scheduler and workload models are used to symbolically compute worst-case or bounded end-to-end latencies for control task chains. These latency bounds play a role analogous to the timing uncertainty, weakly-hard constraints, or delayed inference patterns analyzed earlier in this paper. Rather than enforcing strict deadline satisfaction, SOTERIA injects these bounds into a digital twin of the controller, physics, and environment, and checks whether a safety property $\mathcal{P}$ holds: $M \parallel \mathcal{E} \models \mathcal{P}$.

9.3 Alignment with Quantitative Safety Metrics

Although originally formulated in terms of feasibility, the safety checks in SOTERIA can be interpreted directly using the quantitative metrics developed in previous sections. Over a finite horizon H, the digital twin induces a set of admissible trajectories or a flowpipe F parameterized by timing or estimation imperfections.

Safety can then be assessed by evaluating quantities such as the maximum trajectory deviation $Dev(\tau, \tau_{\text{nom}})$ or the maximum reachable-set diameter $Diam(F)$, and comparing them against an environment-dependent safety threshold $D^{\text{safe}}(\mathcal{E})$. From this perspective, SOTERIA provides a principled way to interpret the bounds derived earlier, not in the abstract, but relative to a specific operating environment.

9.4 Complementarity with Imperfect Components

A key insight of SOTERIA is that strict timing feasibility is not always necessary for safety. A system may violate classical schedulability constraints or experience bounded inference delays, yet still satisfy

$$Dev(\cdot) < D^{\text{safe}}(\mathcal{E}) \quad \text{or} \quad Diam(\cdot) < D^{\text{safe}}(\mathcal{E})$$

in a given environment $\mathcal{E}$. Conversely, the same system may be unsafe in a more demanding environment $\mathcal{E}'$.

This environment-aware perspective complements the paper's central theme. Weakly-hard timing models, learning imperfections, and resource allocation strategies quantify how imperfections propagate through computation and control; SOTERIA determines when those propagated effects actually matter for safety.

9.5 Perspective and Outlook

In the context of this section, SOTERIA serves as a decision layer that sits atop system-level safety analysis. The techniques developed in earlier sections derive meaningful bounds on deviation and uncertainty; SOTERIA interprets those bounds with respect to physics and environment. Together, these approaches suggest a unified workflow for safety assurance of autonomous systems with imperfect components: (a) quantify imperfections using system-level metrics;

(b) propagate their effects through control and physical dynamics; and (c) evaluate safety relative to an environment-specific threshold $D^{\text{safe}}(\mathcal{E})$.

By explicitly modeling the environment, SOTERIA extends system-level safety reasoning beyond worst-case abstractions and toward realistic, context-aware verification of autonomous systems.

10 Related Work

This paper builds on and connects multiple research threads in cyber-physical systems (CPS), spanning resource-aware design, timing and scheduling, safety and verification, learning-enabled control, and edge–cloud architectures. Our contribution differs from much of the prior work by adopting a unified, *system-level* notion of safety that explicitly accounts for imperfect components and quantifies their combined impact on closed-loop behavior.

Resource-aware CPS design and control/architecture co-design. A large body of work has studied resource-aware CPS design, where control performance must be maintained despite constraints on computation, communication, and memory bandwidth [22,79]. This includes models that explicitly capture processor availability [28], network communication limits [130], and shared memory or cache effects on control tasks [26,27]. A prominent methodology in this space is *control/architecture co-design*, which jointly optimizes control parameters (e.g., sampling periods or gains) and architectural parameters (e.g., task schedules or platform configurations) [3,47,48,107]. Toolchains supporting such co-design have been reported in [111,133]. These approaches typically formulate an optimization problem that resolves partial specifications of controllers and implementation platforms to ensure compatibility and performance [50,104,119]. Extensions of this idea have been applied to systems experiencing aging effects, such as processor slowdown or battery degradation, where controller parameters are adapted to preserve performance [23,24]. In automotive networks such as FlexRay, co-design has been used to jointly determine controller parameters and time-/event-triggered communication schedules [73,105,131], supported by timing analysis and schedule synthesis techniques [78,84,118,123].

While this literature seeks compatibility between controllers and architectures, it often assumes *qualitative* correctness guarantees (e.g., deadlines are always met). In contrast, the work reported in this paper accepts that such guarantees may be violated in practice and instead reasons about whether the resulting deviations remain within system-level safety bounds, as formalized in Section 3.

CPS design methods and model-based design. Control/architecture co-design is closely related to broader CPS design methodologies and model-based design frameworks [25,49,108]. Prior work has investigated modeling state dependencies within CPS applications [10], integrating multiple control applications on shared architectures [90], and managing the evolution of automotive control software [103]. Other studies address design under uncertainty, such as anticipating future functional extensions [94,114] or constructing communication schedules that can accommodate unknown future traffic without disrupting existing timing guarantees [120]. These methods primarily aim at architectural robustness and

extensibility. Our approaches discussed in this paper complements them by focusing on how architectural and timing uncertainties propagate through control dynamics and affect quantitative safety metrics, which are then evaluated using the techniques in Section 4 and exploited for synthesis in Section 5.

CPS safety, reliability, and security. Safety and reliability have long been central concerns in CPS research, including formal verification of safety and security properties [38, 93, 98]. Traditional approaches rely on testing [126] or formal verification techniques [11, 65] that often assume idealized component behavior. Extensions address reliability under hardware aging or unreliable platforms [24, 51, 83], as well as fault diagnosis and recovery [134]. Safety has also been studied in the context of platform architecture design [76, 124] and automotive ECUs and in-vehicle networks [46, 72, 106, 121].

Security-related research has examined CPS vulnerabilities, runtime monitoring, and verification of security properties [69,92,112,135,148]. While these efforts provide important guarantees, they typically focus on individual components or subsystems. In contrast, our work here explicitly integrates safety analysis across timing, control, and learning components, and evaluates safety at the system level using quantitative metrics (Section 3) rather than binary pass/fail criteria.

Timing analysis, scheduling, and CPS implementation. A substantial literature addresses timing analysis of embedded software and architectures [54, 82, 87, 101, 139], timing analysis of control software in particular [4,62,147], including WCET estimation [6, 61, 89] and real-time scheduling algorithms [18, 19, 74, 75, 122, 129]. Energy-aware scheduling has also been studied [88], along with protocol-specific analyses for FlexRay and service-oriented architectures [33, 91, 113, 115]. Other work more tightly couples timing analysis with control design [20, 37, 102, 110].

Research on timing isolation and mixed-criticality CPS has explored how safety-critical control software can be protected from interference by less critical tasks [80, 81, 109]. These approaches typically aim to enforce strict timing guarantees. In contrast, our work relaxes this assumption and instead checks and synthesizes schedules that may include deadline misses, provided they satisfy system-level safety constraints (Sections 4–5).

Control performance under timing violations. More recent work explicitly studies control performance under deadline misses and timing uncertainty. Rather than requiring all deadlines to be met, this line of research characterizes which deadline hit/miss patterns preserve safety or stability [43, 52, 56]. Verification techniques have been developed to check whether a given schedule satisfies such patterns [41, 64, 145], and synthesis approaches aim to construct schedules that intentionally allow deadline misses to improve resource utilization [140–142]. Statistical methods have been introduced to accelerate both verification and synthesis [42]. This body of work directly informs our approach. We extended and elaborated these ideas by embedding them within a unified safety framework that supports both deterministic and statistical reasoning (Section 6) and integrated them with learning-enabled components and architectural decisions.

Safety-driven design of learning-enabled CPS. Autonomous CPS increasingly rely on deep neural networks (DNNs) for perception and estimation [9,55,67,117]. Prior

work has studied efficient and timing-predictable vision pipelines [1, 5, 36] as well as formal methods to bound DNN inference uncertainty [132, 144]. More recent research has begun to move beyond component-level correctness by explicitly linking DNN errors and timing violations to system-level safety [17,21,57,143,146, 150]. We elaborated advances this direction by integrating learning imperfections with control and scheduling decisions, and by evaluating their combined effect using quantitative safety metrics (Sections 7 and 8).

Edge–cloud CPS and hybrid architectures. Edge–cloud CPS architectures have been explored to balance latency, bandwidth, and accuracy in DNN-based inference pipelines. Early systems demonstrated collaborative inference between edge and cloud [63], followed by extensive work on compression, pruning, quantization, and sparsity to reduce communication overhead [13, 16, 29, 68, 70, 85, 86]. Frameworks such as CDE, I-SPLIT, and Split-Et-Impera improve split-point selection and interpretability [12, 31, 116], while multi-task extensions address shared edge platforms [14]. Existing control-oriented studies often assume continuous cloud availability or treat latency as an external parameter [56, 149]. In contrast, we explicitly modeled hybrid edge–cloud control strategies under intermittent cloud invocation and analyzes their safety impact using system-level metrics (Section 7). This perspective is further extended to resource allocation among multiple DNNs in Section 8 and to environment-aware interpretation of safety bounds via digital twins in Section 9.

11 Concluding Remarks

Modern autonomous systems are increasingly defined by complex software. This paradigm has unlocked new capabilities but has also widened the gap between the idealized system design and its practical implementation. Traditional verification methods, which often focus on verifying that each component behaves ideally—tasks never miss deadlines, hardware never fails—are becoming untenable in the face of this complexity.

This paper has argued for and demonstrated a fundamental shift in perspective: from **qualitative, component-level requirements** to **quantitative, system-level properties**. Rather than requiring, "the task must never miss its deadline," we instead determine whether the cumulative effect of deadline misses causes the system to deviate beyond a specified safety boundary. This perspective accepts imperfection as a reality of complex systems and provides a formal basis for reasoning about its consequences.

The technical contributions of this paper provide a cohesive framework for realizing this vision. We began by introducing how **quantitative safety metrics**: a system's deviation from its nominal behavior, and the spread in a system's reachable states over time (Section 3). We then developed techniques for **verifying safety under timing uncertainty**, specifically occasional deadline misses that still satisfy weakly-hard constraints (Section 4). Building on this analytical foundation, we showed how to **synthesize schedules that provably guarantee safety** even when resource limitations make deadline misses unavoidable (Sections 5 and 6). In addition, we addressed imperfections in Learning-Enabled

Systems (LES) by developing strategies for **split computing** and scenarios when **multiple neural estimators share the same computational resource** (Sections 7 and 8). Finally, Section 9 presented *SOTERIA*, which complements the preceding analysis by **addressing system-level safety in the presence of imperfect runtime orchestration and component interactions**. While earlier sections quantify how timing and learning imperfections propagate through control systems, SORIERA demonstrates **how such bounds can be managed and enforced at the architectural level**. Combined with the environment-aware digital-twin-based safety analysis, this provides a cohesive view of safety assurance that spans computation, control, orchestration, and operating context.

This work also opens several promising avenues for future research. First, many of our methods are currently designed for linear systems only. Extending these methods to nonlinear dynamics can be valuable. Second, while we addressed timing and estimation imperfections separately, how to analyze the combined effects of multiple, simultaneous sources of imperfection remains an open challenge. Finally, extending our design-time synthesis techniques to enable dynamic, runtime decision-making can make our methods much more adaptable.

Acknowledgments: This work is partially supported by the NSF Grant 2038960 and a Dieter Schwarz Courageous Research Grant from Germany. Chakraborty is also a Fellow of the TUM IAS in Germany.

References

1. Amert, T., et al.: Timing-predictable vision processing for autonomous systems. In: Design, Automation and Test in Europe Conference (DATE) (2021)
2. Ames, A.D., et al.: Control barrier functions: Theory and applications. In: 2019 18th European control conference (ECC). pp. 3420–3431. IEEE (2019)
3. Annaswamy, A.M., et al.: Arbitrated network control systems: A co-design of control and platform for cyber-physical systems. In: Workshop on Cyber-Physical Systems (CPSW@CISS) (2013)
4. Balszun, M., et al.: Effectively utilizing elastic resources in networked control systems. In: International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2017)
5. Balszun, M., et al.: Predictable vision for autonomous systems. In: IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC) (2020)
6. Becker, M., et al.: Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors: A comeback of model checking. International Journal on Software Tools for Technology Transfer (STTT) **21**(5), 515–543 (2019)
7. Bernat, G., et al.: Weakly hard real-time systems. IEEE Transactions on Computers **50**(4), 308–321 (Apr 2001)
8. Bogomolov, S., et al.: JuliaReach: a toolbox for set-based reachability. In: ACM International Conference on Hybrid Systems: Computation and Control (HSCC) (2019)
9. Bordoloi, U.D., et al.: Autonomy-driven emerging directions in software-defined vehicles. In: Design, Automation and Test in Europe Conference (DATE) (2023)

10. Bouillard, A., et al.: Lightweight modeling of complex state dependencies in stream processing systems. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2009)
11. Broy, M., et al.: Cross-layer analysis, testing and verification of automotive control software. In: International Conference on Embedded Software (EMSOFT) (2011)
12. Capogrosso, L., 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) (2023)
13. Capogrosso, L., et al.: Enhancing split computing and early exit applications through predefined sparsity. In: Forum on Spec. & Design Lang. (FDL) (2024)
14. Capogrosso, L., et al.: MTL-Split: Multi-Task Learning for Edge Devices using Split Computing. In: Design Automation Conference (DAC) (2024)
15. Capogrosso, L., et al.: LO-SC: Local-only split computing for accurate deep learning on edge devices. In: Intl. Conference on VLSI Design (VLSID) (2025)
16. Carra, D., Neglia, G.: DNN Split Computing: Quantization and Run-Length Coding are Enough. In: Global Communications Conference (GLOBECOM) (2023)
17. Chakraborty, S., Schneider, K.: Designing imperfect cyber-physical systems. In: Forum on Specification and Design Languages (FDL) (2025)
18. Chakraborty, S., Thiele, L.: A new task model for streaming applications and its schedulability analysis. In: Design, Auto. & Test in Europe Conf. (DATE) (2005)
19. Chakraborty, S., et al.: On the complexity of scheduling conditional real-time code. In: Workshop on Algorithms and Data Structures (WADS) (2001)
20. Chakraborty, S., et al.: Timing and schedulability analysis for distributed automotive control applications. In: Int. Conf. Embedded Soft. (EMSOFT) (2011)
21. Chakraborty, S., et al.: Cross-layer interactions in CPS for performance and certification. In: Design, Auto. & Test in Europe Conf. (DATE) (2019)
22. Chang, W., Chakraborty, S.: Resource-aware automotive control systems design: A cyber-physical systems approach. Foundations and Trends in Electronic Design Automation **10**(4), 249–369 (2016)
23. Chang, W., et al.: Battery- and aging-aware embedded control systems for electric vehicles. In: IEEE Real-Time Systems Symposium (RTSS) (2014)
24. Chang, W., et al.: Reliable CPS design for mitigating semiconductor and battery aging in electric vehicles. In: Cyber-Physical Systems, Networks, and Applications (CPSNA) (2015)
25. Chang, W., et al.: Model-based design of resource-efficient automotive control software. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (2016)
26. Chang, W., et al.: Memory-aware embedded control systems design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **36**(4), 586–599 (2017)
27. Chang, W., et al.: Cache-aware task scheduling for maximizing control performance. In: Design, Automation and Test in Europe Conference (DATE) (2018)
28. Chang, W., et al.: OS-aware automotive controller design using non-uniform sampling. ACM Transactions on Cyber-Physical Systems **2**(4), 26:1–26:22 (2018)
29. Choi, H., et al.: Deep Feature Compression for Collaborative Object Detection. In: 25th International Conference on Image Processing (ICIP) (2018)
30. Chollet, F.: Xception: Deep Learning with Depthwise Separable Convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017)
31. Cunico, F., et al.: I-SPLIT: Deep Network Interpretability for Split Computing. In: International Conference on Pattern Recognition (ICPR) (2022)

32. Ericsson: 5G Experience is Determined by Speed, not Latency. https://www.ericsson.com/en/blog/2022/8/who-cares-about-latency-in-5g, Accessed: 2025-05-18
33. Fraccaroli, E., et al.: Timing predictability for SOME/IP-based service-oriented automotive in-vehicle networks. In: Design, Automation and Test in Europe Conference (DATE) (2023)
34. Gabel, R.A., Roberts, R.A.: Signals and Linear Systems. John Wiley & Sons (Jan 1991)
35. Ganguli, P., et al.: Trading Delays with Uncertainty: Controller Design for DNN-based Perception Processing on Edge-Cloud Platforms. In: 33rd International Conference on Real-Time Networks and Systems (RTNS) (2025)
36. Geier, M., et al.: GigE vision data acquisition for visual servoing using SG/DMA proxying. In: IEEE International Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia) (2016)
37. Geier, M., et al.: Debugging fpga-accelerated real-time systems. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2020)
38. Georgakos, G., et al.: Reliability challenges for electric vehicles: From devices to architecture and systems software. In: Design Automation Conf. (DAC) (2013)
39. Ghosh, B., Duggirala, P.S.: Robust Reachable Set: Accounting for Uncertainties in Linear Dynamical Systems. ACM Trans. Embed. Comput. Syst. **18**(5s), 97:1–97:22 (Oct 2019)
40. Ghosh, B., Duggirala, P.S.: Robustness of Safety for Linear Dynamical Systems: Symbolic and Numerical Approaches (Sep 2021). https://doi.org/10.48550/arXiv.2109.07632
41. Ghosh, B., et al.: Statistical hypothesis testing of controller implementations under timing uncertainties. In: International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2022)
42. Ghosh, B., et al.: Statistical verification of autonomous system controllers under timing uncertainties. Real-Time Systems **60**(1), 108–149 (2024)
43. Ghosh, S.K., et al.: Design and validation of fault-tolerant embedded controllers. In: Design, Automation and Test in Europe Conference (DATE) (2018)
44. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Hybrid Systems: Computation and Control (HSCC). Springer (2005)
45. Girard, A., et al.: Compute-efficient reachability analysis for linear systems with time-varying delays. In: International Symposium on Automated Technology for Verification and Analysis (ATVA) (2006)
46. Glaß, M., et al.: Designing heterogeneous ECU networks via compact architecture encoding and hybrid timing analysis. In: Design Auto. Conf. (DAC) (2009)
47. Goswami, D., et al.: Co-design of cyber-physical systems via controllers with flexible delay constraints. In: Asia and South Pacific Design Automation Conference (ASP-DAC) (2011)
48. Goswami, D., et al.: Re-engineering cyber-physical control applications for hybrid communication protocols. In: Design, Automation and Test in Europe Conference (DATE) (2011)
49. Goswami, D., et al.: Model-based development and verification of control software for electric vehicles. In: Design Automation Conference (DAC) (2013)
50. Goswami, D., et al.: Multirate controller design for resource- and schedule-constrained automotive ecus. In: Design, Automation and Test in Europe Conference (DATE) (2013)
51. Goswami, D., et al.: Fault-tolerant embedded control systems for unreliable hardware. In: IEEE International Symposium on Integrated Circuits (ISIC) (2014)

52. Goswami, D., et al.: Relaxing signal delay constraints in distributed embedded controllers. IEEE Tran. on Control Systems Technology **22**(6), 2337–2345 (2014)
53. Gou, J., et al.: Knowledge Distillation: A Survey. International Journal of Computer Vision **129**(6), 1789–1819 (2021)
54. Hagiescu, A., et al.: Performance analysis of flexray-based ecu networks. In: Design Automation Conference (DAC) (2007)
55. Hobbs, C., et al.: Perception computing-aware controller synthesis for autonomous systems. In: Design, Automation and Test in Europe Conference (DATE) (2021)
56. Hobbs, C., et al.: Safety analysis of embedded controllers under implementation platform timing uncertainties. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(11), 4016–4027 (2022)
57. Hobbs, C., et al.: Quantitative safety-driven co-synthesis of cyber-physical system implementations. In: International Conference on Cyber-Physical Systems (ICCPS) (2024)
58. Howard, A., et al.: Searching for MobileNetV3. In: International Conference on Computer Vision (ICCV) (2019)
59. Howard, A.G., et al.: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications (Apr 2017). https://doi.org/10.48550/arXiv.1704.04861
60. Jocher, G., Chaurasia, A., Qiu, J.: Ultralytics YOLO. https://github.com/ultralytics/ultralytics
61. Ju, L., et al.: Performance debugging of Esterel specifications. In: Intl. Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2008)
62. Ju, L., et al.: Timing analysis of Esterel programs on general-purpose multiprocessors. In: Design Automation Conference (DAC) (2010)
63. Kang, Y., et al.: Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. SIGPLAN Notices **52**(4) (2017)
64. Kauer, M., et al.: Formal verification of distributed controllers using time-stamped event count automata. In: Asia and South Pacific Design Automation Conference (ASP-DAC) (2013)
65. Kauer, M., et al.: Fault-tolerant control synthesis and verification of distributed embedded systems. In: Design, Auto. and Test in Europe Conf. (DATE) (2014)
66. The KITTI Vision Benchmark Suite. https://www.cvlibs.net/datasets/kitti/, Accessed: 2024-06-24
67. Kruber, F., et al.: Vehicle position estimation with aerial imagery from unmanned aerial vehicles. In: IEEE Intelligent Vehicles Symposium (IV) (2020)
68. Li, G., 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)
69. Liang, H., et al.: Security-driven codesign with weakly-hard constraints for real-time embedded systems. In: IEEE International Conference on Computer Design (ICCD) (2019)
70. Liang, T., et al.: Pruning and Quantization for Deep Neural Network Acceleration: A Survey. Neurocomputing **461**, 370–403 (2021)
71. Liberzon, D.: Switching in systems and control, vol. 190. Springer (2003)
72. Lukasiewycz, M., Chakraborty, S.: Concurrent architecture and schedule optimization of time-triggered automotive systems. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2012)
73. Lukasiewycz, M., et al.: FlexRay switch scheduling - a networking concept for electric vehicles. In: Design, Automation and Test in Europe Conf. (DATE) (2011)

74. Lukasiewycz, M., et al.: Modular scheduling of distributed heterogeneous time-triggered automotive systems. In: Asia and South Pacific Design Automation Conference (ASP-DAC) (2012)
75. Lukasiewycz, M., et al.: Priority assignment for event-triggered systems using mathematical programming. In: Design, Automation and Test in Europe Conference (DATE) (2013)
76. Lukasiewycz, M., et al.: System architecture and software design for electric vehicles. In: Design Automation Conference (DAC) (2013)
77. Maggio, M., et al.: Control-System Stability Under Consecutive Deadline Misses Constraints. LIPIcs, Volume 165, ECRTS 2020 **165**, 21:1–21:24 (2020). https://doi.org/10.4230/LIPICS.ECRTS.2020.21
78. Majumdar, D., et al.: Reconfigurable communication middleware for flexray-based distributed embedded systems. In: International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2015)
79. Maldonado, L., et al.: Exploiting system dynamics for resource-efficient automotive cps design. In: Design, Automation and Test in Europe Conference (DATE) (2019)
80. Masrur, A., et al.: VM-based real-time services for automotive control applications. In: International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2010)
81. Masrur, A., et al.: Designing VM schedulers for embedded real-time applications. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2011)
82. Masrur, A., et al.: Schedulability analysis of distributed cyber-physical applications on mixed time-/event-triggered bus architectures with retransmissions. In: International Symposium on Industrial Embedded Systems (SIES) (2011)
83. Masrur, A., et al.: Schedulability analysis for processors with aging-aware autonomic frequency scaling. In: International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2012)
84. Masrur, A., et al.: Timing analysis of cyber-physical applications for hybrid communication protocols. In: Design, Automation and Test in Europe Conference (DATE) (2012)
85. Matsubara, Y., 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)
86. Matsubara, Y., 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 (2022)
87. Maxiaguine, A., et al.: Rate analysis for streaming applications with on-chip buffer constraints. In: Asia and South Pacific Design Autom. Conf. (ASP-DAC) (2004)
88. Maxiaguine, A., et al.: DVS for buffer-constrained architectures with predictable QoS-energy tradeoffs. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2005)
89. Metta, R., et al.: TIC: A scalable model checking based approach to wcet estimation. In: Languages, Compilers, Tools and Theory for Embedded Systems (LCTES) (2016)
90. Minaeva, A., et al.: Control performance optimization for application integration on automotive architectures. IEEE Transactions on Computers **70**(7), 1059–1073 (2021)
91. Mundhenk, P., et al.: Policy-based message scheduling using FlexRay. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2014)

92. Mundhenk, P., et al.: Lightweight authentication for secure automotive networks. In: Design, Automation and Test in Europe Conference (DATE) (2015)
93. Mundhenk, P., et al.: Security analysis of automotive architectures using probabilistic model checking. In: Design Automation Conference (DAC) (2015)
94. Mundhenk, P., et al.: Dynamic platforms for uncertainty management in future automotive E/E architectures: Invited. In: Design Automation Conference (DAC) (2017)
95. Murphy, K.: Analysis of Robotic Vehicle Steering and Controller Delay. In: 5th International Symposium on Robotics and Manufacturing ISRAM '94 (Aug 1994)
96. NVIDIA: NVIDIA GeForce RTX 5060. https://www.nvidia.com/en-us/geforce/graphics-cards/50-series/rtx-5060-family/, Accessed: 2025-05-18
97. NVIDIA: NVIDIA Jetson Nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/, Accessed: 2025-05-18
98. Oetjens, J.H., et al.: Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges. In: Design Automation Conference (DAC) (2014)
99. O'Kelly, M., et al.: F1TENTH: An Open-source Evaluation Environment for Continuous Control and Reinforcement Learning. In: Proceedings of Machine Learning Research (PMLR). vol. 123, pp. 77–89. PMLR (2020)
100. Osman, K., et al.: Modelling and controller design for a cruise control system. In: 5th International Colloquium on Signal Processing & Its Applications (2009)
101. Phan, L.T.X., et al.: Timing analysis of mixed time/event-triggered multi-mode systems. In: IEEE Real-Time Systems Symposium (RTSS) (2009)
102. Phan, L.T.X., et al.: Modeling buffers with data refresh semantics in automotive architectures. In: International Conference on Embedded Software (EMSOFT) (2010)
103. Ramesh, S., et al.: Specification, verification and design of evolving automotive software: Invited. In: Design Automation Conference (DAC) (2017)
104. Roy, D., et al.: Automated synthesis of cyber-physical systems from joint controller/architecture specifications. In: Forum on Specification and Design Languages (FDL) (2016)
105. Roy, D., et al.: Multi-objective co-optimization of FlexRay-based distributed control systems. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2016)
106. Roy, D., et al.: Hybrid automotive in-vehicle networks. In: ACM/IEEE International Symposium on Networks-on-Chip (NOCS) (2017)
107. Roy, D., et al.: Semantics-preserving cosynthesis of cyber-physical systems. Proceedings of the IEEE **106**(1), 171–200 (2018)
108. Roy, D., et al.: Waterfall is too slow, let's go agile: Multi-domain coupling for synthesizing automotive cyber-physical systems. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (2018)
109. Roy, D., et al.: Goodspread: Criticality-aware static scheduling of cps with multi-qos resources. In: IEEE Real-Time Systems Symposium (RTSS) (2020)
110. Roy, D., et al.: Timing debugging for cyber-physical systems. In: Design, Automation and Test in Europe Conference (DATE) (2021)
111. Roy, D., et al.: Tool integration for automated synthesis of distributed embedded controllers. ACM Transactions on Cyber-Physical Systems **6**(1), 3:1–3:31 (2022)
112. Sagstetter, F., et al.: Security challenges in automotive hardware/software architecture design. In: Design, Automation and Test in Europe Conference (DATE) (2013)

113. Sagstetter, F., et al.: Schedule integration framework for time-triggered automotive architectures. In: Design Automation Conference (DAC) (2014)
114. Sagstetter, F., et al.: Multischedule synthesis for variant management in automotive time-triggered systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **35**(4), 637–650 (2016)
115. Sagstetter, F., et al.: Generalized asynchronous time-triggered scheduling for flexray. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **36**(2), 214–226 (2017)
116. Sbai, M., et al.: Cut, Distil and Encode (CDE): Split Cloud-Edge Deep Inference. In: International Conference on Sensing, Communication, and Networking (SECON) (2021)
117. Scharfenberger, C., et al.: Robust image processing for an omnidirectional camera-based smart car door. ACM Transactions on Embedded Computing Systems **11**(4), 87:1–87:28 (2012)
118. Schneider, R., et al.: Optimized schedule synthesis under real-time constraints for the dynamic segment of flexray. In: IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC) (2010)
119. Schneider, R., et al.: Constraint-driven synthesis and tool-support for flexray-based automotive control systems. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2011)
120. Schneider, R., et al.: On the quantification of sustainability and extensibility of flexray schedules. In: Design Automation Conference (DAC) (2011)
121. Schneider, R., et al.: Compositional analysis of switched Ethernet topologies. In: Design, Automation and Test in Europe Conference (DATE) (2013)
122. Schneider, R., et al.: Multi-layered scheduling of mixed-criticality cyber-physical systems. Journal of Systems Architecture **59**(10-D), 1215–1230 (2013)
123. Schneider, R., et al.: Quantifying notions of extensibility in flexray schedule synthesis. ACM Transactions on Design Automation of Electronic Systems **19**(4), 32:1–32:37 (2014)
124. Shreejith, S., et al.: VEGa: A high performance vehicular ethernet gateway on hybrid fpga. IEEE Transactions on Computers **66**(10), 1790–1803 (2017)
125. Tan, M., Le, Q.V.: EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (Sep 2020). https://doi.org/10.48550/arXiv.1905.11946
126. Tibba, G., et al.: Testing automotive embedded systems under x-in-the-loop setups. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (2016)
127. Tilbury, D., Messner, B.: Control Tutorials for MATLAB and Simulink
128. Vajgl, M., et al.: Dist-YOLO: Fast Object Detection with Distance Estimation. Applied Sciences **12**(3), 1354 (Jan 2022). https://doi.org/10.3390/app12031354
129. Voit, H., et al.: Optimizing hierarchical schedules for improved control performance. In: International Symposium on Industrial Embedded Systems (SIES) (2010)
130. Voit, H., et al.: Adaptive switching controllers for systems with hybrid communication protocols. In: American Control Conference (ACC) (2012)
131. Voit, H., et al.: Adaptive switching controllers for tracking with hybrid communication protocols. In: IEEE Conference on Decision and Control (CDC) (2012)
132. Wang, Z., et al.: Bounding perception neural network uncertainty for safe control of autonomous systems. In: Design, Automation and Test in Europe Conference (DATE) (2021)
133. Waszecki, P., et al.: How to engineer tool-chains for automotive E/E architectures? SIGBED Review **10**(4), 6–15 (2013)

134. Waszecki, P., et al.: Decentralized diagnosis of permanent faults in automotive E/E architectures. In: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS) (2015)
135. Waszecki, P., et al.: Automotive electrical and electronic architecture security via distributed in-vehicle traffic monitoring. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **36**(11), 1790–1803 (2017)
136. Wilson, K., Aothers: Soteria: A formal digital-twin-enabled framework for safety-assurance of latency-aware cyber-physical systems. In: 28th ACM International Conference on Hybrid Systems: Computation and Control (HSCC) (2025)
137. Wilson, K., et al.: Physics-aware mixed-criticality systems design via end-to-end verification of cps. In: 22nd ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE). IEEE (2024)
138. Wilson, K., et al.: Physics-informed mixed-criticality scheduling for f1tenth cars with preemptable ROS 2 executors. In: IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS) (2025)
139. Xie, G., et al.: A real-time CAN-CAN gateway with tight latency analysis and targeted priority assignment. In: IEEE Real-Time Systems Symp. (RTSS) (2020)
140. Xu, S., et al.: Safety-aware flexible schedule synthesis for cyber-physical systems using weakly-hard constraints. In: Asia and South Pacific Design Automation Conference (ASP-DAC) (2023)
141. Xu, S., et al.: Safety-aware implementation of control tasks via scheduling with period boosting and compressing. In: International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2023)
142. Xu, S., et al.: Statistical approach to efficient and deterministic schedule synthesis for cyber-physical systems. In: International Symposium on Automated Technology for Verification and Analysis (ATVA) (2023)
143. Xu, S., et al.: GPU partitioning & neural architecture sizing for safety-driven sensing in autonomous systems. In: International Conference on Autonomous Agents (ICAA) (2024)
144. Xu, S., et al.: Poster abstract: Neural architecture sizing for autonomous systems. In: International Conference on Cyber-Physical Systems (ICCPS) (2024)
145. Yeolekar, A., Metta, R., Hobbs, C., Chakraborty, S.: Checking scheduling-induced violations of control safety properties. In: International Symposium on Automated Technology for Verification and Analysis (ATVA) (2022)
146. Yeolekar, A., et al.: Repairing control safety violations via scheduler patch synthesis. In: International Conference on Cyber-Physical Systems (ICCPS) (2025)
147. Zhang, L., et al.: Timing challenges in automotive software architectures. In: International Conference on Software Engineering Companion (ICSE Companion) (2014)
148. Zhao, Q., et al.: CAN bus intrusion detection based on auxiliary classifier GAN and out-of-distribution detection. ACM Transactions on Embedded Computing Systems **21**(4), 45:1–45:30 (2022)
149. Zhu, T., et al.: Controllers for edge-cloud cyber-physical systems. In: International Conference on Communication Systems and Networks (COMSNETS) (2025)
150. Zhu, T., et al.: Safety-driven DNN sizing for vehicular CPS. IEEE Embedded Systems Letters (2025). https://doi.org/https://doi.org/10.1109/LES.2025.3595839

Table 1: Summary of Notation

Control Models		
$x(t)$, $x[k]$	Plant state (continuous/discrete time), with $x(\cdot) \in \mathbb{R}^p$	Sec. 2.1
$u(t)$, $u[k]$	Control input, with $u(\cdot) \in \mathbb{R}^q$	Sec. 2.1
$y(t)$	Measured output, with $y(\cdot) \in \mathbb{R}^r$	Sec. 2.1
A, B, C, D	Continuous-time LTI system matrices in $\dot{x}(t) = Ax(t) + Bu(t)$, $y(t) = Cx(t) + Du(t)$	Sec. 2.1
T	Sampling period for discretization	Sec. 2.1
D	Sensor-to-actuator delay within a sampling period	Sec. 2.1
Φ, Γ	Discrete-time dynamics matrices in $x[k+1] = \Phi x[k] + \Gamma u[k]$	Sec. 2.1
$\Gamma_0(D)$, $\Gamma_1(D)$	Input matrices capturing the effect of delayed actuation	Sec. 2.1
K	Feedback gain matrix used to compute $u[k]$	Sec. 2.1
M_W, M_S, M_C, M_P	The workload model, the scheduler model, the controller, and the physical dynamics	Sec. 9.2
Automata and Weakly-Hard Constraints		
$\mathcal{A} = \langle \Sigma, Q, \delta, s_0, F \rangle$	Finite automaton encoding admissible hit/miss patterns	Sec. 2.2
Σ	Alphabet of scheduler outcomes/actions	Sec. 2.2
Q	Set of automaton states	Sec. 2.2
δ	Transition function (extended to words as δ^*)	Sec. 2.2
w	Word over Σ; represents a hit/miss sequence over the horizon	Sec. 2.3
H	Time horizon (word length / number of periods)	Sec. 2.3
$\binom{m}{k} = \binom{m}{k}$	Weakly-hard constraint: at least m hits in every window of length k	Sec. 2.3
$\overline{\langle m \rangle} = \overline{\langle m \rangle}$	Weakly-hard constraint: at most m consecutive misses	Sec. 2.3
`Hold`, `Zero`	Input substitution policies: reuse last input / apply zero input	Sec. 2.3
`Kill`, `Skip-Next`	Task overrun policies: terminate overrun / skip next job release	Sec. 2.3
Safety Metrics and Synthesis		
τ	System trajectory over a finite horizon	Sec. 3
τ^{nom}	Nominal (ideal) trajectory with no imperfections	Sec. 3
$\mathcal{T}$	Set of trajectories (e.g., induced by a weakly-hard constraint)	Sec. 3
$\texttt{Dev}(\tau, \tau^{\text{nom}})$	Deviation of τ from τ^{nom} (trajectory distance metric)	Sec. 3.1
$\mathcal{F}$	Flowpipe: sequence of reachable sets over the horizon	Sec. 3.2
$\texttt{Diam}(\mathcal{F})$	Maximum diameter of reachable sets in a flowpipe	Sec. 3.2
$\texttt{D}^{\text{safe}}$	Safety requirement (maximum allowed deviation/diameter)	Sec. 3.3
$\overline{\texttt{D}}$, $\widehat{\texttt{D}}$	Sound upper bound / statistical estimate of a safety metric	Sec. 3.3
n, j	Number of controllers and per-slot execution capacity	Sec. 5
S	Schedule assigning controller jobs to time slots	Sec. 5
c	Confidence level used in statistical hypothesis testing	Sec. 6
DNN_E, DNN_C	Edge and cloud neural estimators in split computing	Sec. 7
D_E, D_C	Edge and cloud inference delays	Sec. 7
u_E, u_C	Control inputs computed from edge/cloud estimates	Sec. 7
J	Assignment of neural networks to state components (DNN configuration)	Sec. 8.1
$\texttt{D}^{\text{safe}}(\mathcal{E})$	Environment ($\mathcal{E}$)-dependent Safety requirement	Sec. 9.3

Table 2: Number of Points Where Each Strategy Minimizes Deviation [56]

Miss Strategy	RC Net Steering	Aircraft	F1Tenth
Hold & Kill	**91**	95	**100**	**100**
Zero & Kill	67	**100**	0	0
Hold & Skip-Next	0	0	0	0
Zero & Skip-Next	0	0	0	0

Table 3: Running Time Over Varying Time Horizon (100, 300, 1000 Steps), At Most Three Consecutive Misses [56]

Miss Strategy	Algorithm 1	Algorithm 2	Algorithm 4
Hold & Kill	1.9, 15.7, 172	0.36, 1.0, 3.6	0.096, 0.31, 1.1
Zero & Kill	2.0, 16.4, 178	0.35, 1.0, 3.5	0.101, 0.34, 1.1
Hold & Skip-Next	7.8, 72.0, —	0.66, 1.9, 6.8	0.47, 1.4, 4.3
Zero & Skip-Next	8.1, 74.6, —	0.65, 1.9, 6.5	0.47, 1.4, 4.6

Table 4: Running Time Over Varying Misses for 150 Time Steps, Hold&Skip-Next [56]

Algorithm	$\overline{\langle 2 \rangle}$	$\overline{\langle 4 \rangle}$	$\overline{\langle 8 \rangle}$	$\overline{\langle 16 \rangle}$
Algorithm 2	0.72 s	1.20 s	2.12 s	3.92 s
Algorithm 4	0.43 s	0.87 s	1.7 s	3.4 s

Table 5: Summary of schedule synthesis results for benchmark controllers [141]

Common Period P_C	Gains	Safe Schedule Exists?	Notes
15 ms	original	no	only 1 task per slot
28 ms	original	yes	feasible without redesign
40 ms	recomputed	yes	redesign required

Table 6: Initial states and safety margins for the five benchmarks

	Initial State	Safety Margin
RC Network	$[1\ 1]^T$	0.07
F1 Tenth Car	$[1\ 1]^T$	0.56
DC Motor	$[100\ 100]^T$	0.1
Car Suspension	$[100\ 100\ 100\ 100]^T$	0.8
Cruise Control	$[1\ 1\ 1]^T$	0.06

Table 7: Deviation upper bounds $\widehat{\mathtt{D}}$ for each system and weakly hard constraint, computed by the SHT-based method [142]

	Window Size (k)	Minimum Hits (m) 1	2	3	4	5	6
RC Network $\mathtt{D}^{\mathtt{safe}} = 0.07$	1	0.0	–	–	–	–	–
	2	0.036	0.0	–	–	–	–
	3	0.0656	0.036	0.0	–	–	–
	4	0.0899	0.0656	0.036	0.0	–	–
	5	0.11	0.0899	0.0656	0.036	0.0	–
	6	0.126	0.11	0.0899	0.0656	0.036	0.0
F1 Tenth Car $\mathtt{D}^{\mathtt{safe}} = 0.56$	1	0.0	–	–	–	–	–
	2	0.179	0.0	–	–	–	–
	3	0.364	0.179	0.0	–	–	–
	4	0.557	0.364	0.179	0.0	–	–
	5	0.75	0.557	0.364	0.179	0.0	–
	6	0.949	0.75	0.557	0.364	0.179	0.0
DC Motor $\mathtt{D}^{\mathtt{safe}} = 0.1$	1	0.0	–	–	–	–	–
	2	0.0546	0.0	–	–	–	–
	3	0.107	0.0546	0.0	–	–	–
	4	0.156	0.107	0.0546	0.0	–	–
	5	0.204	0.157	0.107	0.0546	0.0	–
	6	0.248	0.204	0.157	0.107	0.0546	0.0
Car Suspension $\mathtt{D}^{\mathtt{safe}} = 0.8$	1	0.0	–	–	–	–	–
	2	0.16	0.0	–	–	–	–
	3	0.34	0.16	0.0	–	–	–
	4	0.53	0.34	0.159	0.0	–	–
	5	0.729	0.529	0.339	0.159	0.0	–
	6	0.908	0.717	0.526	0.338	0.158	0.0
Cruise Control $\mathtt{D}^{\mathtt{safe}} = 0.06$	1	0.0	–	–	–	–	–
	2	0.0138	0.0	–	–	–	–
	3	0.0298	0.0115	0.0	–	–	–
	4	0.0368	0.0212	0.0117	0.0	–	–
	5	0.0455	0.0323	0.0194	0.0115	0.0	–
	6	0.0584	0.0423	0.0262	0.0201	0.0116	0.0

Table 8: Exact deviation values for each system when scheduled according to Figure 10 [142]

	Safety Margin	Closest Constraint	Estimated $\widehat{\mathtt{D}}\left(\frac{m}{k}\right)$	Estimated $\overline{\mathtt{D}}\left(\frac{m}{k}\right)$	Actual Deviation
RC Network	0.07	$\left(\frac{1}{3}\right)$	0.0656	0.0656	0.0092
F1 Tenth Car	0.56	$\left(\frac{1}{3}\right)$	0.364	0.364	0.303
DC Motor	0.1	$\left(\frac{1}{2}\right)$	0.0546	0.0546	0.157
Car Suspension	0.8	$\left(\frac{1}{3}\right)$	0.34	1.04	0.13
Cruise Control	0.06	$\left(\frac{1}{2}\right)$	0.0138	0.0712	0.00578

Table 9: Execution times for the SHT-based and deterministic methods

	SHT ($c = 0.99$)	DET ($n = 15$)	DET ($n = 18$)
RC Network	2.17 s	115.35 s	818.92 s
F1 Tenth Car	2.09 s	115.96 s	817.70 s
DC Motor	2.78 s	112.95 s	820.18 s
Car Suspension	3.10 s	292.35 s	2071.46 s
Cruise Control	3.33 s	111.65 s	799.89 s
Schedule Synthesis	0.017 s	0.106 s	0.012 s
Schedule Verification	0.008 s	–	–
Total	13.50 s	748.37 s	5328.16 s
Schedulable?	Yes	No	Yes

Table 10: Performance of different control strategies (distance from ideal trajectory) [35]

Scenario	Avg. of RMSE	SD of RMSE
(a) Edge-Only control	113.57	44.44
(b) Cloud-Only control	41.04	18.74
(c) Edge-Cloud Hybrid control	**27.38**	**13.20**

Table 11: Performance comparisons of the different setups [35]

Latency of DNN_E, d_E (in s)	Latency of DNN_C, d_C (in s)	SD of DNN_E, σ_E	SD of DNN_C, σ_C	n	Best performing controller
0.005	0.01 – 0.03	0.3	0.2	0 – 1	Cloud-Only
0.005	0.04 – 0.50	0.3	0.2	2 – 25	Hybrid
0.01	0.01 – 0.05	0.7	0.3	0 – 2	Cloud-Only
0.01	0.06 – 0.50	0.7	0.3	3 – 25	Hybrid
0.01	0.01 – 0.19	1.5	1.2	0 – 9	Cloud-Only
0.01	0.20 – 0.50	1.5	1.2	10 – 25	Hybrid
0.02	0.02 – 0.62	0.5	0.2	0 – 25	Hybrid

Table 12: Computation cost and accuracy tradeoff for EfficientNet configurations on ImageNet

Model	FLOPs	Accuracy
EfficientNet_B0	0.39G	77.1%
EfficientNet_B1	0.70G	79.1%
EfficientNet_B2	1.0G	80.1%
EfficientNet_B3	1.8G	81.6%
EfficientNet_B4	4.2G	82.9%
EfficientNet_B5	9.9G	83.6%
EfficientNet_B6	19G	84.0%
EfficientNet_B7	37G	84.3%

Table 13: Computation cost and Absolute Relative Error (ARE) tradeoff for Dist-YOLO backbones on KITTI.

Backbone	FLOPs	ARE
MobileNetv3_small	9.833G	42.53%
MobileNetv2	43.714G	34.23%
EfficientNet_B2	69.371G	30.61%
EfficientNet_B3	84.574G	27.23%
Xception	103.935G	24.34%
EfficientNet_B6	205.171G	21.08%

Table 14: Mean Squared Error (MSE) of DNNs for estimating width, distance from right wall, and orientation respectively by size [150]

DNN Size	**DNN_wid MSE**	**DNN_dis MSE**	**DNN_ori MSE**
(a) (24 kB)	0.8125	0.2323	0.0930
(b) (66 kB)	0.2270	0.0603	0.0760
(c) (192 kB)	0.0247	0.0059	0.0205
(d) (326 kB)	0.0057	–	–
(e) (596 kB)	0.0029	–	–

Table 15: Summary of Network Combinations and Performance [150]

DNN_wid	**DNN_dis**	**DNN_ang**	**Cost**	**Avg. Dev.**	**Max Dev.**
Group 1: Cost = 408kB					
(c)	(c)	(a)	408	0.0518	0.0864
(a)	(c)	(c)	408	0.0686	0.1107
(c)	(a)	(c)	408	0.0736	0.1113
Group 2: Cost = 324kB					
(c)	(b)	(b)	324	0.0582	0.0998
(b)	(c)	(b)	324	0.0839	0.1241
(b)	(b)	(c)	324	0.0863	0.1168
Group 3: Cost = 240kB					
(c)	(a)	(a)	240	0.0625	0.0981
(a)	(c)	(a)	240	0.0716	0.1097
(a)	(a)	(c)	240	0.0746	0.1113
Group 4: Cost = 114kB					
(b)	(a)	(a)	114	0.0531	0.0906
(a)	(b)	(a)	114	0.0919	0.1435
(a)	(a)	(b)	114	0.1494	0.2118