

Main Sources of Variability and Non-Determinism in AD Software: Taxonomy and Prospects to Handle Them

Miguel Alcon^{1,2}, Axel Brando¹, Enrico Mezzetti¹,

Jaume Abella¹, Francisco J. Cazorla¹

¹Barcelona Supercomputing Center

²Universitat Politecnica de Catalunya

Abstract

Safety standards in domains like automotive and avionics seek for deterministic execution (lack of jittery behavior) as a stepping stone to build a certification argument on the correct timing behavior of the system. However, the use of artificial-intelligence (AI) software in safety-critical systems carries several built-in and derivative sources of non-determinism that are at odds with safety standard determinism requirements. In this work we analyze the main sources of non-determinism of autonomous driving (AD) software, as highly representative and compelling example of the use of AI software, deep neural networks (DNN) in particular, in critical embedded systems. Paradoxically, DNN-based software in its inference phase – once the NN structure and weights have been fixed – turns out to consist mainly in matrix multiplications, which are inherently quite time deterministic. Our work focuses on sources of variability and non-determinism in AD software, covering algorithmic elements of AD software, low-level software and hardware computing platform, and data-flow constraints among AD modules. As final contribution of our work, which mainly focuses on problem identification, we develop some prospects on the information and metrics needed to better understand and control the unpredictability and non-determinism of AD software.

I. INTRODUCTION

The hardware and software deployed in safety-critical systems, in industries like automotive and avionics, have been historically very conservative, with simple control software running on top of single-core processors (a.k.a. electronic control units) and controlling few input signals and

actuators. In such scenario, the main interaction among applications, and hence the contention among them, occurs at the network level (e.g. CAN). However, in recent years, there have been an increase in the number and sophistication of software-delivered functionalities as one of the main competitive factors of embedded products. In the automotive domain, emerging solutions range from Advanced Driver Assistance Systems (ADAS) functionalities to low-level autonomous driving (AD) vehicles (e.g. L3), with full (L5) AD as future prospect. This trend has required a paradigm shift in both the software and hardware solutions adopted in time critical embedded systems:

- At the software level, artificial intelligence (AI) software in general, and deep learning (DL) techniques in particular, are at the very heart of the realization of advanced software functions such as computer vision (e.g., object detection and tracking), path planning, driver-monitoring systems, and gesture-activated AI assistants. This is so because DL techniques have largely demonstrated their effectiveness in managing complex and heterogeneous problems, outperforming other type of control algorithms [1], [2]. Especially for decision-making functions, DL solutions are a cornerstone in the development of future advanced (fully) autonomous systems. As such, DL is considered crucial for providing functionally-critical features such as perception (e.g. obstacle detection), path/trajectory planning, vehicle tracking in state-of-the-art AD vehicles.
- At the hardware level, also boosted by the increasing performance requirements of DL software, the adoption of MultiProcessor system on Chip (MPSoCs) is on the rise even in critical systems. MPSoCs are equipped with a score of powerful computing elements like CPUs, GPUs, TPUs, and embedded FPGAs that can implement specific accelerators. Besides, modern MPSoCs encompass complex and distributed interconnects, in addition to several types of on-SoC and off-SoC memories [3], [4].

Both, DL software and MPSoCs, bring significant challenges to the overall Validation and Verification (V&V) strategy in critical systems as defined in domain specific documents like ISO 26262 and DO-178C. In this work, we focus on the software timing aspects of certification. The impact of the entailed paradigm shift is so deep that new standards are being considered and defined along with the first deployments of these software and hardware technologies, e.g. SOTIF [5] for capturing automotive-specific challenges from autonomous systems, and A(M)C 20-193 [6] in avionics for the safe use of software on MPSoCs.

The main difference between old simple control software running on simple micro-controllers and new AI software running on multicores is that the latter introduce potential sources of non-determinism. In this line, safety standards call for solutions to handle those sources of non-determinism.

For instance, ISO 26262 in automotive [7] requires showing “freedom from interference”, whereas CAST-32A – now A(M)C 20-193 – [8] for avionics promotes the identification and mitigation of interference channels. The common goal in both cases, also shared with other domains, is seeking for execution time predictability and determinism, which actually requires analyzing and understanding software execution time variability.

In this line, as a first step, several works already present and characterize timing variability in AD frameworks [9], [10], [11], [12], [13]. Unlike these works, whose main focus is on the *outcome* of non-determinism, i.e. analyzing the execution time distributions, in this contribution we move the center of gravity to the main *sources* of variability and non-determinism in AD frameworks. We perform our analysis at the three main layers in the computing stack:

- At the algorithm level, in which we identify the main AI models used by the most performance intensive modules of Apollo and the respective sources of variability and non-determinism;
- At the software architecture level, where we focus on the characterizing traits of AD frameworks to observe how timing variability may affect data-flow dependencies between input data and software modules, with the result of magnifying the effect of the variability stemming from the other layers in the execution stack.
- At the platform level, which is an area deeply covered in the real-time literature, we summarize the sources of variability at MPSoC and run-time software levels;

In our taxonomy, we identify *original* sources of execution time variability and *derivative* sources. In the former group, we find (i) data inputs, which is further classified into current input and history of inputs; (ii) randomization in the AI models used by Apollo; and (iii) the underneath variability of the hardware and software platform. In the latter group we include (i) the activation modes and (ii) data-flow dependencies at the software architecture level.

The rest of this contribution is structured as follows. After introducing Apollo in Section II, we proceed analyzing the original sources of execution time variability, those at the algorithm (Section III) and platform (Section IV) levels. Then, in Section V, we analyze the derivative sources of variability that mainly happen at software data-flow level. In the same section, we also

develop on cascade variability effects, occurring when a small variation in an original source of variability is magnified (resonance effect) by other derivative sources. Section VI supports our reasoning with specific examples from the Apollo on CyberRT framework. In Section VII we consider the implications of timing variability and non-determinism in the final system behavior and identify the types of information and metrics that are instrumental in understanding the impact of variability. Section VIII presents the related work covering different aspects and sources of non-determinism and variability. Finally, Section IX summarizes the main conclusions of this work.

II. APOLLO

A. Introduction to Apollo

In this work we consider Apollo [14] as a representative AD software. Apollo is an industrial-quality open-source AD software framework developed and distributed by Baidu together with over 120 industrial partners, the majority of which are car manufacturers and top-tier AI companies. Indeed, Apollo is already present on the road as it has been deployed on a variety of prototype vehicles, including self-driving robotaxis [15]. The framework supports cutting-edge hardware such as Velodyne and other suppliers' latest LiDARs and cameras, as well as GPU acceleration.

Apollo is structured as a set of highly-coupled and interconnected *modules*. A *module* is a piece of software in charge of a specific functionality of the AD framework, from perceiving the surroundings of the vehicle to steering or braking. Modules receive data from *sensors* or other modules, process it, and send the result to following module(s) in the *execution chain* according to a well-define data-flow semantics. In Figure 1 we present the main software architecture and modules in Apollo and how they are chained.

Following the natural flow of Apollo's execution, from receiving inputs from sensors to controlling the vehicle, we next describe the main functionality of each module. *HD Map* acts as a library and operates as a query engine support that offers ad-hoc organized information about the roads to *Planning*, *Prediction*, and *Perception*. *Perception* detects obstacles (e.g., pedestrians, vehicles, or barriers), traffic signs, and drivable regions in the vicinity of the autonomous vehicle. It combines the output of many types of sensors, including LiDAR, radar, and cameras, to increase its accuracy. It is typically the most complex and performance intensive module in AD systems. *Localization* is another module that utilizes sensors. In particular, it combines

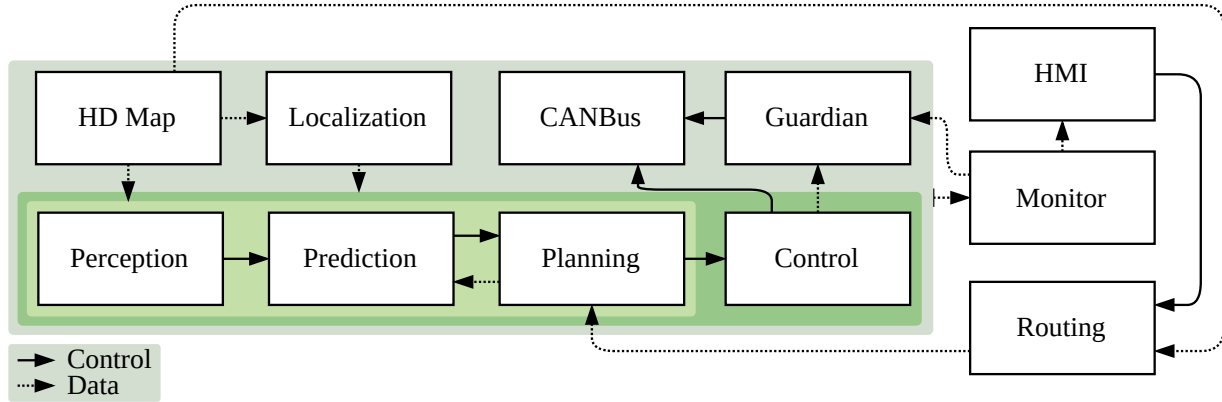


Fig. 1: Apollo's interface view.

various sources of information, such as GPS, LiDAR, and IMU (inertial measurement units), to estimate where the autonomous vehicle is located. *Prediction* anticipates the speed and future motion trajectories of obstacles once they have been detected by *Perception* and the vehicle has been located after *Localization*. *Routing* calculates the vehicle's current route. *Planning* uses the obstacles trajectories (from *Prediction*), and vehicle's current state (velocity, steering, etc.), route (from *Routing*) and location (from *Localization*) to create a safe and comfortable spatio-temporal trajectory for the autonomous vehicle to follow. *Control* generates control commands such as acceleration, braking, and steering to carry out the scheduled spatio-temporal trajectory. *CanBus* implements these control commands by acting as the control interface that sends commands to the vehicle's hardware, and communicates chassis data to the software system.

The Perception [16] module is composed of other (sub)-modules that carry out a particular functionality. Figure 2 shows Perception's architecture. *Traffic Light* receives messages from the camera(s), detects whether there is a traffic light in front of the vehicle and if so, recognizes the active light. The rest of the (sub)-modules receive messages from the sensor with their name, detect the obstacles in range, and classify them. *Fusion* is fed with the detection of each sensor and fuses the information on the same obstacles gathered by the different sensors. It also classifies each object depending on whether it is a vehicle, bicycle, or pedestrian. Finally, *Perception* outputs the detected obstacles with the heading, velocity and classification information, and the detected traffic light with the light's color.

While *Traffic Light* can receive messages from multiple sensors, *Camera*, *LiDAR* and *Radar* modules can be instantiated multiple times to process data from multiple sensors. For instance,

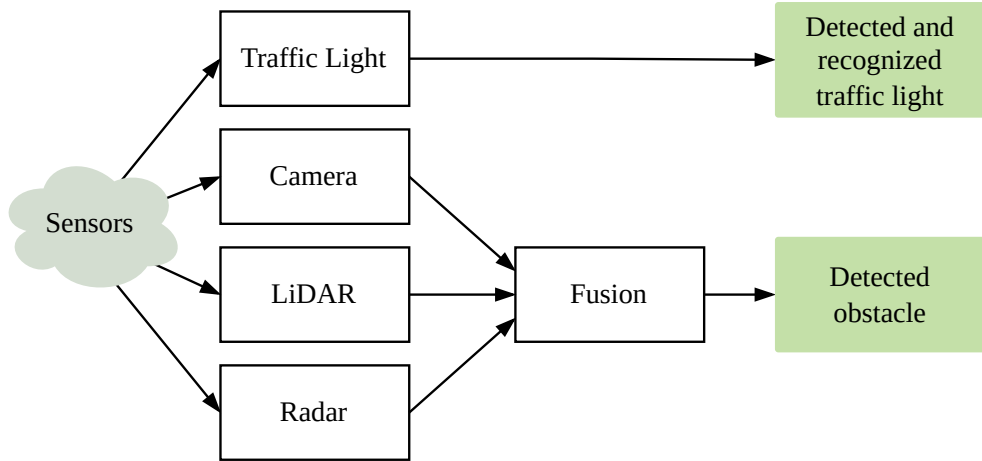


Fig. 2: General architecture of Perception.

one of the configurations shown by Apollo uses two different types of cameras, four LiDARs and two radars, for covering all possible angles. Each of them must be handled by a single module.

B. Generalization

In this work, we focus on Apollo v5-v7 (v7 is the latest version available while developing this paper) since the characteristics we build on for our analysis are common across all those versions. Those characteristics relate to the use of a subscription-notification communication framework such as CyberRT, and AI algorithms (DNNs in particular). Moreover, those characteristics are not expected to change in the foreseeable future.

Beyond Apollo and the CyberRT framework, on which Apollo builds and we use in our case study to support our findings, the conclusions we achieve can be generalized to other AD frameworks. This includes Autoware [17] and middlewares like the Robot Operating System (ROS) [18] (used by Apollo v3 and earlier versions) and its newest version ROS2 [19]. We highlight the following commonalities between Autoware, ROS, and Apollo.

ROS/ROS2 and CyberRT are based on message communication organized by topics, subscribers and publishers. The use of a message-based communication scheme implies that:

- Both framework families build on procedures, for manage messages, and define event- and time-triggered processes that read messages, process them, and write new messages as an output. This aspect is illustrated in Section V in Figure 9 and 10. Process activation and

communication in Apollo and ROS are analogous and the message processing schemes would require, at most, minor changes if Apollo were deployed ROS instead of CyberRT.

- Message dropping can occur. This fact relates to the intrinsic nature of AD frameworks, as well as other autonomous operation systems (e.g., robots), where a number of processes communicate asynchronously for simplicity and modularity. Hence, while most messages are intended to be processed, some of them may be superseded by newer information that arrives before the old one has been processed. CyberRT, and also ROS, hence, naturally support this behavior, which is not exclusive to the particular framework evaluated in our paper. In fact, we observed it in Apollo v3 (ROS) years ago, and the same happens with CyberRT. Therefore, what we discuss in Section V-B with Figure 11 should hold (at least conceptually) for other AD frameworks that use message-based communication.

Regarding Apollo and Autoware, they are very similar in terms of structure. Both are composed by several modules that take care of a specific functionality of the software. Modules like Perception, Planning, Control, Localization, or Map, are seen in both AD frameworks and build upon analogous frameworks for message and data exchange and process activation. Only few details discussed in Section V (like buffers and fusion buffers, or the specific case showed in Figure 11) are specific to CyberRT, yet they illustrate timing behavior comparable to that of other frameworks since, as discussed before, the asynchronous nature of the communication among a number of processes leads to behavior where some messages need being discarded

Regarding NNs, the uncertainties that those in Apollo bring are not Apollo specific, but common across most or even all networks. It is expected that all AD frameworks use NNs (image processing for cameras, point cloud processing for LiDARs, etc.), like Apollo and Autoware do.

Hence, the analysis we perform is not framework-specific and applies to other very popular framework like Autoware+ROS2. Both frameworks are consolidated, with Apollo+CyberRT already used by 120 industrial partners (mainly car manufacturers and top-tier AI companies). Further, our initial analysis of other frameworks like Apex.OS [20] and RTI Connex Drive [21], indicates that our analysis is also valid for them to a large extent.

III. ALGORITHMIC-RELATED VARIABILITY

We first introduce AI models with emphasis on NN models (Section III-A) and follow up with an analysis of the main potential sources of execution time variability at the algorithm level

(Sections III-B and III-C).

At algorithm-level, execution time variability can essentially stem from the input of the algorithm or, alternatively, from randomization features that are inherently part of the algorithmic solution. Our breakdown of these different sources of algorithmic variability is summarized in Figure 3.

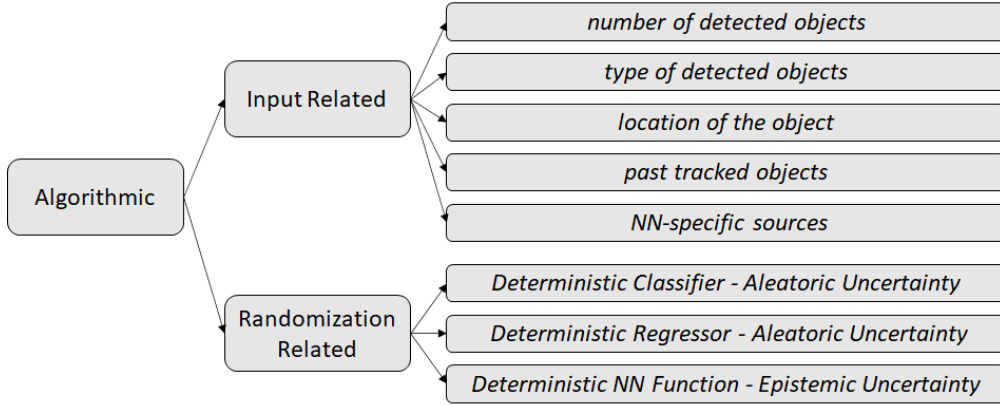


Fig. 3: Breakdown of algorithmic sources of variability.

We discuss these main categories in Sections III-B and III-C, identifying those aspects actually causing execution time variability and those that do not.

A. AI models and their usage in Apollo

The development of AD technology has gone hand in hand with advanced AI models [22], [23], [24]. In this work we focus on supervised learning models [25] that require labeled data in order to learn and perform tasks such as identifying objects in images. In this context, the most popular AI models are Deep learning/ Neural Network (DL/NN) models¹. Apollo exploits different DL models, as follows.

- The Perception module uses DL models to detect and identify objects. As can be observed from the second column of Table I, it uses inputs from different sensors (camera, radar and LiDAR). The particular tasks of Perception using DL models (third column of Table I) include obstacle localization and detection, lane line detection and classification, traffic light detection and recognition, and LiDAR segmentation.

¹We use the terms Deep Learning (DL) and Neural Network (NN) indistinctly.

MODULE	SENSOR	TASK	MODEL TASK	NN ARCHITECTURE
Perception	Camera	Obstacle localization	Regression	YOLO
		Obstacle detection	Regression	YOLO
		Lane line detection	Classification	MLP+CNN-1d
		Lane classifier	Classification	DarkSCNN
		Traffic light detection	Classification	R-CNN
		Horizontal traffic light recognition	Classification	CNN
	Front/rear radar	Object detection	Classification	MLP
	LiDAR	Segmentation	Classification	CNN
Prediction	-	Obstacle path and speed	Regression	RNN
		Intersection / turning direction prediction	Classification	MLP
		Trajectory prediction	Regression	Social LSTM
Planning	-	System coordination	-	Threshold based

TABLE I: The relevant Deep Learning models of Apollo [26] with their tasks.

- The Prediction module uses the perception information to anticipate what other objects will do next using different DL models. These DL tasks include predicting their trajectories, speeds, and intentions (e.g., whether they will turn left or right at an intersection).
- The Planning module decides what action the vehicle should take next, based on the predictions made by the previous two modules. Here, the DL models are not directly involved as the decisions are mainly taken based on reliable thresholds and hand-crafted rules.

Overall, our analysis of Apollo reveals that it heavily builds on supervised learning models, mainly in the perception and prediction modules.

In order to analyze execution time of supervised learning models, we divide them into classifiers or regressors [27]:

- Classification is a type of ML algorithm that is used to predict the class or category that an input data point belongs to. For example in order to classify images of animals we would use a classification model that would take in an image and output whether it represents a

dog, a cat, or a bird.

- Regression is another type of ML algorithm that is used to predict continuous values. For example, in order to predict the price of a house based on its size and location, we would use regression.

The fourth column in Table I identifies whether *classification* or *regression* tasks are performed in Apollo (see Figure 4), based on a NN model architecture specified in the rightmost column in the same table. The latter includes YOLO (presented later in this section), MLP [28], CNN-1d [29], DarkSCNN [30], R-CNN [31] and Social LSTM [32].

The main classification and regression part of Apollo are those related to obstacles detection using the camera (as shown in the first and second rows in Table I). Both classification and regression tasks take an image in input. Regression outputs the bounding box coordinates (e.g., as real values) for each detected object, whereas classification outputs the corresponding class label (e.g., vehicle, pedestrian, etc.).

In Apollo, these two tasks are performed by a single deep learning model called YOLO (You Only Look Once) [33]. YOLO consists in a particular NN architecture trained on millions of images so it can accurately detect objects even when they are partially occluded or rotated. However, the higher the lack of information, the worse the YOLO's results.

YOLO reserves part of the NN outputs to classification, and the rest to regression. In particular, we are interested in identifying the execution time variability of these types of algorithms (classifier or regressor), which we analyze in more detail next.

- From a probabilistic standpoint [34], given a set of classes, $C = \{c_1, c_2, \dots, c_{N_c}\}$, a classifier tries to estimate the conditional probability for each class, $P(Y = c_i | X)$ where Y is the random variable to forecast and X the dependent variable already known (i.e. the input variables).
- Instead, a regressor usually tries to estimate the mean (or median) of $p(Y | X)$, i.e. the possible valid values of Y given a certain input X . These are, generically, conditional statistics of that continuous distribution $p(Y | X)$.

In both cases, there are two main sources of execution time variability that can be identified at the algorithmic level. First, the input-related variability, which covers the variability that comes from the properties of the input information, and second, the intrinsic randomization, which refers to the variability stemming from stochastic behaviors of the considered DL model, as we discuss below.



Fig. 4: Illustrative example detected objects using Apollo. The execution time depends on the detected object number and their individual predicted class.

B. Apollo Input Related Execution Time Variability

Object count, type and location. From an input-related standpoint, there are different potential factors that can significantly impact the execution time.

- (I1): The number of detected objects, with larger number of objects potentially requiring more time to perform the task.
- (I2): The type of detected objects that affects the action taken by Apollo.
- (I3): The location of the object in the forecasted regions, e.g., the Apollo behavior changes if a pedestrian is on the road or the sidewalk, or whether a vehicle is in the same lane or not.

Those three factors can be seen in Figure 3, in the list of input related factors, along with two additional factors that we discuss later on. The variability generated by the aforementioned three factors is better explained with illustrative examples.

- (E1): Given two captured images, $\{X_1, X_2\}$, where the detection model identifies the same number of objects and equally classified, if they also share the location on the predicted regions, the execution time is indistinguishably similar from an algorithmic point of view.
- (E2): Given two captured images, $\{X_1, X_2\}$, where the detection model identifies the same number of objects but differently classified, depending on the subsequent routines for each kind of class, the execution time can differ between these two images.
- (E3): Given two captured images, $\{X_1, X_2\}$, where the recognition model detects a dif-

ferent number of objects, the execution time can vary because the executed code will be different.

Overall, varying the number of detected objects (**I1**) determines how many times the object processing functions are called, since they are called for each object detected by iterating over a list of detected objects. Hence, naturally, the execution time of the object detection process is the addition of the time to process each one of the objects.

The type of a given detected object (**I2**) is used as the condition to decide how the object must be processed since actions may vary for different object types. For instance, if an object in the road in front of the car is another vehicle, the actions to take will differ drastically to the case where it is a pedestrian. In the former case, it is acceptable driving at high speed if the other car is also driving at a similar or higher speed in the same direction. However, if the object is a pedestrian, its speed can only be low and having it in front of the car is likely to trigger an emergency reaction to avoid the collision. All in all, algorithms used for different object types will differ arbitrarily, and hence also their execution times will.

Finally, also the location of a detected object (**I3**) has an impact on the algorithm used to process the object. For instance, if a pedestrian is in front of the car, an emergency reaction can be triggered, whereas if it is in one side of the car, only some actions can be forbidden or allowed only at low speed (e.g., a sudden turn to the side of the pedestrian). Hence, as for the type of objects, algorithms may differ arbitrarily, and their execution times will vary accordingly.

Past tracked objects. Apollo keeps a “state” with the information that captures whether certain scenarios have similar execution time. At an algorithmic level, the state is characterized by the number of tracked objects, their predicted class, and their location. Tracked objects are maintained in an internal Apollo list, including all the detected obstacles, to monitor them until they disappear from the scene for enough time to be deleted. This tracking process is performed inside the segmentation step described in Table I by the so-called HM object tracker. The HM object tracker is a combination of the Hungarian algorithm [35] for detection-to-track association and the Robust Kalman Filters [36] for motion estimation.

(I4): Apollo also includes information of the past tracked objects as part of the input data for the prediction module.

Therefore, those considerations made for **I1**, **I2** and **I3** also apply to past tracked objects (**I4**), with comparable impact on execution time since algorithms can be executed a variable number

of times, for variable past object counts, as for **I1**, and possibly using different algorithms, depending on the object type and location, as for **I2** and **I3**.

NN specific input variation. Finally, we identify an input-related variability that is specific to NN implementations.

(I5): The weights in the NNs ultimately determine the behavior of the NN layers as the majority of layers are parametric on weights. Such weights are determined or affected by the input data, as provided from previous layer.

A NN typically comprises multiple layers: most of them, or at least those dominating the NN execution time [37] at system operation, perform matrix multiplications. Typically, those matrix multiplications are realized with either multiplications and additions, or multiply-and-add units. If data operated is floating-point (as in the case of Apollo), we can find that execution time varies depending on the values operated. For instance, some floating point unit implementations may check whether one of the input values is zero and, if it is, skip the multiplication or the addition simply delivering a zero or the other input operand as output respectively with much shorter latency. Yet, the impact of these effects are limited since weights are constant for a given NN, and input data sampled from the surrounding environment (e.g., with a camera) changes slowly across consecutive observations. This results in small variations occurring progressively over time and, ultimately, inducing limited performance variations.

However, while these input variations are typically low in current implementations, the increased need for efficiency is pushing towards a transition to event-triggered sensors rather than time-triggered ones. In this scenario, only regions of the input data (e.g., of an image) experiencing changes are processed. Each changing region is processed with varying degrees of resolution based on the estimated importance of the event. Moreover, such processing may use sparse matrix multiplication support to operate only non-zero data to save power [38]. Overall, execution time variations can become prominent due to the varying number and size of the regions to be processed, the resolution used to process each region, and whether data matrices encoding data have a larger or lower fraction of zeros.

C. Intrinsic Randomization Related Execution Time Variability

Another source of variability that can arise at algorithmic level refers to the non-deterministic behavior of the DL model in use. This directly connects with the probabilistic uncertainty sources of any general forecaster system [39], [40], [41], [42], [43], which mainly can be divided into:

- Domain uncertainty [44], [43], which can occur when DL models are trained on data that does not accurately represent the distribution of data in the real world (e.g. an outlier or anomaly behavior appears).
- Epistemic uncertainty [41], which corresponds to the model bias (i.e. decisions about the model that limits its possibility to learn the real approximated process). This type of uncertainty is assumed to be mitigated by gathering more information or considering a better family of models (e.g. considering that the NN weights are random variables instead of one deterministic value).
- Aleatoric uncertainty [41], [42], which refers to the unavoidable randomness of the variable to be predicted given the same input information, i.e. the output variable depends on hidden input variables we do not have access to (e.g. due to occlusion effects, doubtful scenarios). Consequently, this type of uncertainty is irreducible and must be explicitly modeled.

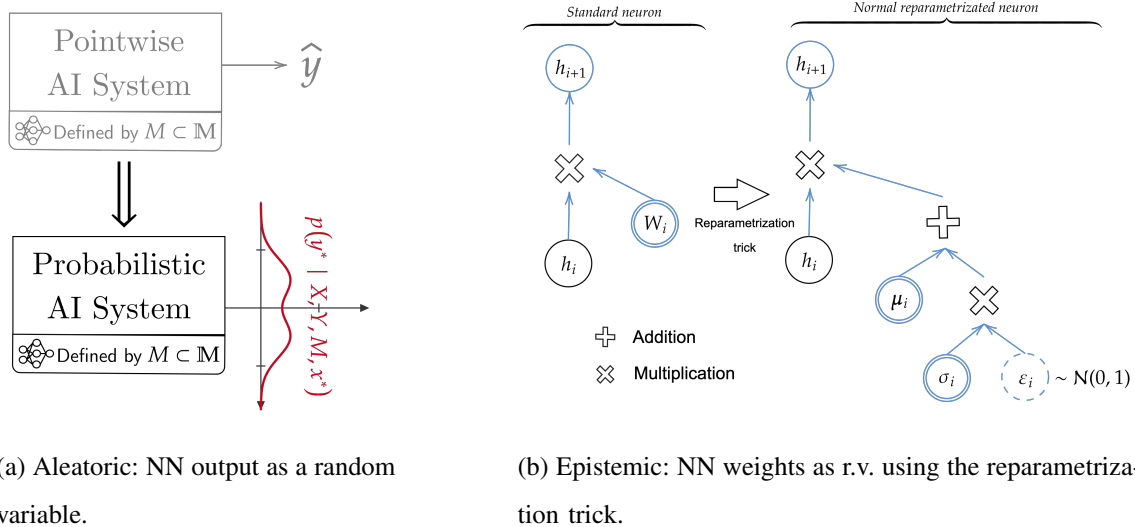


Fig. 5: Process to convert a deterministic NN to stochastic one [42].

All these sources of uncertainty can have significant impact on decision-making. However, only the last two can mainly inject a stochastic behavior on execution time. Namely, a wrong decision (from a domain uncertainty viewpoint), regardless of other uncertainties, would produce always the same wrong output and, consequently, its execution time will always be the same. Differently, adding a stochastic behavior of the DL model in terms of considering a family of possible models (to avoid epistemic errors) or in terms of considering variability in the predicted

outputs (to capture aleatoric errors), could lead to different outputs that can have non-negligible impact on execution time.

Following, we describe the changes to model aleatoric and epistemic uncertainty using the aforementioned classification and regression DL models.

1) *Deterministic Classifier - Aleatoric Uncertainty*: In the classification case [34], given a set of classes $C = \{c_1, c_2, \dots, c_{N_c}\}$, the classifier estimates the probabilities $\{P(Y = c_i | X)\}_{i=1}^{N_c}$ for all the N_c classes. To do so, the standard optimization process for these NN classifiers involve minimizing the cross-entropy loss function, which corresponds to maximizing a multinomial likelihood from a probabilistic viewpoint, i.e.²

$$p(Y | X) = \prod_{c=1}^C \prod_{i=1}^N (\hat{y}_i^{(c)})^{y_i^{(c)}}. \quad (1)$$

Consequently, regardless of the empirical calibration problem [45], the outputs of this type of NNs approximate a probability vector [34]. Then, a common approach to select which class to report is to choose the class with the maximum value, i.e.

$$\hat{y} = \operatorname{argmax}_{c_i} P(Y = c_i | X), \quad (2)$$

as shown in Figure 5a (top figure). Apollo also considers this approach for all its classifiers, so that given a certain input, the NN output will be always the same selected class (even if, for instance, the second class with the highest value has a similar value to the first one). In other words, although the $\{P(Y = c_i | X)\}_{i=1}^{N_c}$ values are probabilities, they are considered simple numbers where the largest is chosen, without looking at the differences between the other class values, i.e. the aleatoric uncertainty is ignored, even it can be crucial to detect unclear scenarios that would require a conservative reaction.

In other real-world applications in autonomous driving systems [47], [48], [49], there are several ways to model the aleatoric uncertainty in DL classification models. Eventually, the goal is to consider the variability of the possible options given a certain input. As we illustrate in Figure 6, the model predicts probabilities for all the C classes, “pred_c” = $\{P(Y = c_i | X)\}_{i=1}^{N_c}$, collects this information, and allows carrying out a sampling process with those probabilities to explore the variability across different potential scenarios. Based on this extra available

²Note that the standard notation in probability p denotes that it is a probability mass function, while P is a probability measure.

<pre> <i>## Deterministic approach</i> import numpy as np sel_c = np.argmax(pred_c) return C[sel_c] </pre>	<pre> <i>## Probabilistic approach</i> import numpy as np return np.random.choose(C,1,p=pred_c) </pre>
--	--

Fig. 6: NumPy [46] sampling process using the output’s classifier. The goal is to obtain a probabilistic approach such as in Figure 5a (bottom figure).

information, several approaches can be applied to obtain an uncertainty index, such as computing the entropy between the forecasted class probabilities [50], [51] or the ratio between the first and second highest probability values [52].

Generally, an aleatoric-aware classifier reports not only the maximum probability class, as a deterministic classifier does, but also provides a measure of uncertainty that considers the probabilities associated to the other classes. This extra information can ultimately result in a change in the reported class, which can have impact in execution time as **I1** and **I4**. Additionally, for each detected object, the aleatoric-aware classifier may report a different top-ranked class, which can have a similar impact on execution time as **I2** and **I3**.

2) *Deterministic Regressor - Aleatoric Uncertainty*: Similarly, in the regression case [53], the standard output of DL regressors is a statistic of the conditional distribution $p(Y | X)$, i.e.

$$\hat{y} = \text{stat}(p(Y | X)), \quad (3)$$

which typically corresponds to the mean (when the least squares are used as an optimization metric in the learning process [54], [55]) or the median (when the least absolutes are, instead, used as the optimization metric [56], [57]), like for the YOLO obstacle localization or the speed prediction shown in Table I. Apollo implementation considers the conditional mean, and it is represented in Figure 5a (top figure). Crucially, in the mean or median cases, the learned statistic is a point-wise approximation of the distribution $p(Y | X)$, therefore, we may be omitting essential information about the probability of the reported value but we can ensure that the NN is always the same given a certain input.

```

## Deterministic approach
# Directly predict value
return pred_v

from numpy.random import standard_normal
## Probabilistic approach
# Mean forecasting
o = pred_mu
# Scale forecasting and sampling
o += standard_normal(1) * pred_s
return o

```

Fig. 7: Difference between directly predicting a value or assuming that $p(Y | X)$ comes from a normal distribution, approximating its parameters (“pred_mu” and “pred_s”) using the DL model, and then sampling from them, see Figure 5a.

Analogously to Subsection III-C1 for the classification case, in Figure 7 we show an approach to move from a point-wise estimation of the conditional probability $p(Y | X)$ to a parametric distribution estimation [58], [59], [53], which allows the DL model to capture, not only the mean value, but also its corresponding conditional variance (both parameters, “pred_mu” and “pred_s”, will be the NN outputs to be optimized), as shown in Figure 5a (bottom figure). This idea of variance can be extended to richer conditional parametric estimators [60] or even non-parametric quantiles [61], [62], [63] using DL models.

As in the case of classifiers, an aleatoric-aware regressor reports not only a conditional statistic of the variable to be predicted, but also provides a view of the overall conditional distribution. Similarly to the classifier case, this extra information can ultimately result in a change in the action to take, which can have impact in execution time as **I1**, **I4** and also in **I2** and **I3**.

3) *Deterministic NN Function - Epistemic Uncertainty*: Each of the regression or classification NN models presented so far has been deemed as a single mathematical or probabilistic function. For example, a YOLO object detector that corresponds to a single classifier. However, when we add the epistemic uncertainty dimension, we need to consider that there might be several different NN models that are sufficiently accurate in general. Nonetheless, their differences can provide crucial information on the level of confidence in the model’s prediction and help to avoid particular model bias effects that arise when relying on a single NN model. This holds

regardless of the aleatoric ability of the network to avoid taking the maximum (Eq. (2)) or point-wise (Eq. (3)) approaches introduced before.

The epistemic variation of the model’s output can be seen as considering a proper family of models richer enough to capture this model bias limitation. In other words, whether a diverse enough ensemble of NNs [64] or a Bayesian approach over the parameters is considered [58], [65], [66], it could become a reliable solution in these terms.

In particular, in Figure 8, we present how a deterministic weight could be considered as a random variable, following the reparametrization trick presented in Figure 7. This change constitutes one of the essential modifications to build a Bayesian neural network [58], one way to model epistemic uncertainty.

<pre>## <i>Deterministic approach</i> w = 30</pre>	<pre>from numpy.random import standard_normal ## <i>Probabilistic approach</i> mu = 30 s = 5 w = mu + s * standard_normal(1)</pre>
--	--

Fig. 8: Converting a deterministic NN weight into a random variable that depends on two deterministic parameters, following Figure 5b.

As for aleatoric uncertainty, epistemic uncertainty can impact object count and type, inducing execution time impact analogous to that of **I1**, **I2** and **I4**.

Overall, standard DL models are not explicitly used from a probabilistic standpoint. While this allows to reduce the sources of algorithmic-related variability, it clearly limits the capability of these models to tackle with Functional Safety requirements because they are not able to model the underlying uncertainty [67]. For the sake of creating reliable DL models, aleatoric and epistemic uncertainty modeling is essential, even if it introduces new sources of variability on execution time. In the presented framework, we highlighted how the probabilistic viewpoint can be compatible and tackled with the already in-use models of Apollo, shown in Table I, and treated in terms of induced variability.

IV. PLATFORM-RELATED VARIABILITY

While the impact of algorithmic timing variability stemming from AI solutions (Section III) on AD software has not been thoroughly explored, significant effort has been devoted to study the variability generated by the computing platform.

In fact, a massive amount of works in the literature analyze and characterize hardware-induced variability, and propose hardware and software techniques to mitigate its impact. In this line, this section provides a general overview of the main family of techniques used to reduce computing platform variability, which includes the underlying MPSoC hardware and the system software (Hypervisor and/or Real-Time Operating System).

A. *Hardware variability*

Hardware execution time variability emanates partially from the use of high-performance hardware features like out-of-order and super-scalar CPU execution pipelines, out-of-order memory operation execution, speculation, etc. All these high-performance micro-architectural designs are in modern COTS computing platforms used or considered for use in critical systems. They provide good average performance at the cost of diminishing system timing predictability [68] and making the computation of execution time bounds much more difficult. While alternative predictable multicore designs have been proposed that provide a compromise between predictability and performance [69], the use of COTS platforms often remains the only strategically viable solution.

MPSoC-induced execution time variability is also heavily affected by multicore contention that causes the duration of a task to depend on the load its co-runner tasks put on hardware shared resources. In fact, domain-specific safety standards and support documents [70], [6], [7] capture specific compliance requirements for multicore software certification, with time determinism and predictability as common requirements across all of them.

Contention modelling of the main shared resources received significant attention in last years. The main resources analyzed are caches, memories, and interconnects [71], [72], [73], [74], [75], [76], [77], [78], [79], [80], [81], [82], [83]. With this respect, several families of techniques have been proposed.

Time and space partitioning of shared resources on top of existing hardware and software solutions have been leveraged by some researchers. These are typically handled either at the system configuration [84], [85], or run-time monitoring and enforcement of resource usage [71].

More recently, several works show how the use of hardware quality of service (QoS) features [86], [87], [88] can provide effective means to control multicore timing interference [89], [90], [91].

Application refactoring is suggested by other works suggest under a given system configuration so that their memory operations are only allowed to happen in dedicated phases (e.g., read-compute-write). This, together with local private memory support, helps to remove or tighten contention bounds [92], [93], [94].

Segregation is another solution deployed by some works that build on information about set and bank indexing in caches and memory. In particular, hypervisor/RTOS techniques force different tasks to be mapped to different cache sets and DDR memory banks/ranks [85], [95]. Other solutions control the access to memory to mitigate the maximum task can generate each other [96], [97].

Hardware designs have been proposed to contain contention, from specific arbitration and communication protocols [75], to bandwidth regulation [98], [99] and support for cache partitioning [95], [100].

B. System-Software variability

At the system software level, RTOS and hypervisor run-time support has been often advocated as a means to guarantee a minimum degree of predictability of top on loosely predictable hardware [101], [102]. A countless number of academic and commercial solutions have been proposed to improve system analyzability and meet mixed-criticality requirements in MPSoCs (ranging from separation kernels [103], [104] to fully-fledged hypervisors [105], [106]). These solutions, however, are out of the scope of this work. What is relevant to our discussion is that the run-time itself can contribute to the system jittery behavior with the inherent variability that arises from the plethora of interrupts and intra-/inter-core scheduling mechanisms. This is especially relevant for AD frameworks which are typically deployed on top of middleware layers that execute on generalist operating system support (i.e. Linux) where the software under analysis is not the only software using the system resources. This is the case of ROS2 [18] and CyberRT [107] frameworks where not all OS background service can be realistically switched off.

V. DATA-FLOW RELATED VARIABILITY

A. Data-Flow in Apollo

AD frameworks [14], [17] build on modular software architectures to favor integration and inter-changeability of software and hardware elements from different providers. Such modularity is typically achieved by deploying the autonomous framework on top of a message-passing based middleware layer organizing the flow of data from sensors through modules in the system.

Apollo is not an exception to this modularity trend, with installations of Apollo (from v3.5 on) based on the CyberRT [107] middleware operating system. CyberRT is an open-source high-performance run-time framework designed by the Apollo team specifically for AD. Compared to previous installations, which were based on ROS [18], CyberRT provides higher concurrency, lower latency, and higher throughput. Yet Apollo’s and Autoware’s main design principles are the same, in line of what was discussed in Section II-B. In particular, similarly to ROS, CyberRT supports message-based communication for the propagation of data from input sensors to the different functional modules and, eventually, take specific actions through actuators. Data flow in CyberRT builds on three main elements: *channels*, *readers*, and *writers*. A channel represents the means supporting the flow of data through typed messages, associated to a specific single type or topic. Modules can interact by receiving (reader) or sending (writer) messages through dedicated channels.

Channels data-flow features are realized using *fixed-size buffers*, with typically one buffer defined per channel. Due to their limited capacity, buffers only hold a sequence of the latest messages, ordered by aging. Elements in a buffer can be accessed by readers and writers either by message age or retrieving the last received message. *Fusion buffers* are a special case of buffer that are responsible for storing messages from multiple channels (up to 4 in the considered implementation) hence realizing data fusion among multiple input sources.

Each Apollo module is realized as a (set of) CyberRT component(s) that essentially comprises a set of readers/writers, to manage input and output data-flow, and a main procedure, catering for the algorithmic implementation of the supported functionalities. Figure 9 exemplifies Apollo types of interactions by focusing on the *Planning* module internal and interface with the *Localization*, *Prediction*, *Routing* and *Traffic Light* modules. *Planning* is a representative module in terms of data-flow as it exploits a fusion buffer and all different types of accesses to channels. The module is subscribed to three channels (hence the fusion buffer handles three channels)

and the module procedure *Process* accesses two further channels via internal readers during its execution. Procedures P_L, P_{Pr}, P_R, P_T are the main function of the connected modules while W_0, W_1, W_2, W_3, W_4 and W_{Pl} represent the writer processes that store messages from channels into buffers. Except for W_0 , which manages data from a sensor, all writer processes handle the output of their channels. The semantic difference is expressed by using different arrow styles (solid and dashed) from writer processes to P_{Pl} . Inward dotted arrows depict possible subscriptions that are not considered in the example.

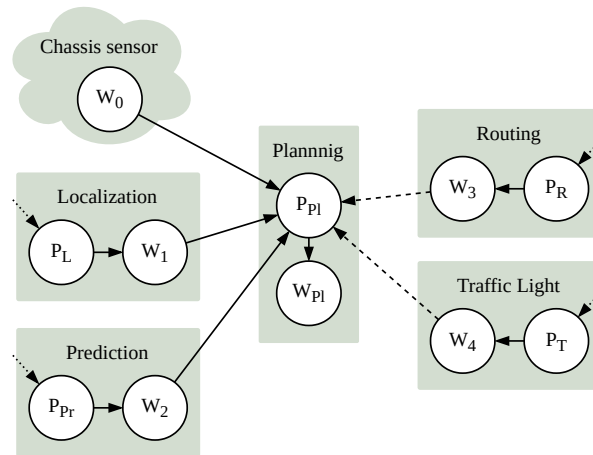


Fig. 9: Simplified view of *Planning* internals and connected modules.

Components rely on *Routines* to fetch messages from the channel buffers and forward the obtained data to the *Process* function. In terms of CyberRT, a routine is in charge of accessing the input buffer to get the messages that are arguments of the function they handle. When messages are available, the module procedure is executed and results, if any, are published to the corresponding output channel (topic).

With fusion buffers, the execution of the procedure is conditioned to the availability of a message in one ‘critical’ channel, whereas the latest available message is used for the remaining channels. Note that an attempt to fetch a message may fail despite the buffer not being empty since channel buffers can be queried for messages with a specific age.

The activation of each Apollo module is regulated by either a time interval or a specific event, see Figure 10. In both activation scenarios, a routine R is responsible for fetching new messages from channel buffers C_i , possibly merging the information (in the case of a fusion buffer), use the received information to execute the module procedure, thus realizing the required

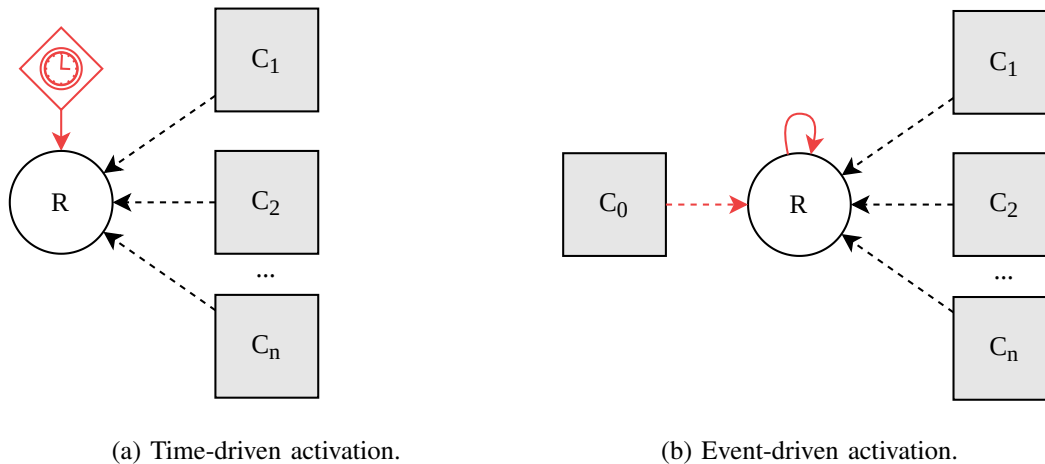


Fig. 10: Apollo-CyberRT activation modes.

functionality. Eventually, R can also forward the received information or newly produced one into other channels.

In the **time-driven** scenario (Figure 10a), the routine is triggered periodically, based on a design-time requirement placed on the functionality implemented by the module. The last received message in all input channels is unconditionally retrieved and processed. In fact, this type of activation always causes the execution of the module procedure: the execution of R only fails the fetch attempt when the buffers are totally empty, which is only possible at initialization phase, since the buffers are never completely flushed.

The **event-driven** scenario is slightly more constrained since the activation of R depends on the availability of a new message (based on aging information) in the ‘critical’ channel, represented as C_0 in Figure 10b. While R is periodically executed, it can only trigger the module procedure when a new message is available in C_0 . No constraint is placed on messages from the other channels (if any) in the module and the latest available message is used. All Apollo modules can be described by the composition of a variable number of activation scenarios in accordance with the data flow across modules and their sub-modules.

B. Propagation and Resonance of Execution Time Variability

Regardless of the scenario, the time between two successive successful activations (triggering the execution of the module procedure) should match the desired frequency of operation for all modules. However, the desired operation frequency is considerably exposed to the variability

arising from the execution of the component procedures and how this variability propagates among modules because of inherent data-flow dependencies.

The execution time of the algorithmic part of an Apollo node is exposed to several factors: (i) intrinsic input-related variability (Section III-B), (ii) platform-level variability (Section IV), and (iii) non-determinism of the specific machine learning solution implemented (Section III-C). All these factors contribute to a fluctuation in the rate at which messages are retrieved and sent from/to the corresponding channels. And even small variations can make the difference with respect to whether a message is dropped from an input channel or to how recent (and relevant) is the information sent to an output channel. Depending on the specific module, the use of outdated or loosely updated messages can lead to a suboptimal or wrong decision. On the contrary, dropping some messages may lead to discarding precious information for the decision process.

Timing variability *propagates* across modules because of the inherent data-flow dependencies and may break the overall design assumptions on frequency of operation. All in all, the data-flow dependencies act as a sounding (resonance) board for all sources of timing variability and non-determinism.

We illustrate the propagation and resonance effects of execution time variability in an example focusing on the *Planning* module in Apollo, see Figure 9. Writers $W_0 - W_4$ represent those processes in charge of storing messages to buffers of specific channels to which *Planning* is subscribed. In particular,

- W_0 , W_1 , and W_2 send messages to the extra-node channels that convey the inputs of the *Planning*'s *Process* procedure P ;
- W_3 and W_4 , instead, update node-local channels that are accessed during the execution of *Process*.

For the sake of clarity in this example, we assume channels are accessed once at the beginning of each activation of P . W_0 writes to the critical channel triggering the activation of P (see Figure 10b) and exploits a FIFO fusion buffer F of 2 elements, where a combination of the latest messages from the buffers updated by W_1 , W_2 and the triggering message from W_0 are stored together.

Figure 11 provides an execution diagram of the processes and messages that collectively provide the module functionality. Labels representing the status of the fusion buffer F are attached to the time line, in correspondence with the arrival of W_0 messages. Those labels

are unfolded in Table II. Each circle represents one instance of W_x and hence a new message written to the corresponding channel, while p_0 , p_1 and p_2 are instances of P .

We assume all buffers are non-empty when the first message $M_{0,0}$ is received from W_0 . At this point, the latest messages from W_1 and W_2 buffers ($M_{1,0}$ and $M_{2,0}$) are fused with $M_{0,0}$ in a single message ($F_0 = [M_{0,0}, M_{1,0}, M_{2,0}]$) and stored in the fusion buffer. On each activation, P retrieves the latest message from the fusion buffer and from buffers handled by W_3 and W_4 . The first instance of P tries to fetch messages from the fusion buffer and p_0 is triggered when the W_0 delivers a message to the channel and F_0 is stored in the fusion buffer. At the same time, p_0 gets the latest messages from W_3 and W_4 buffers, and the execution continues. Following the timeline in Figure 11, let us assume that algorithmic and/or platform-related variability causes p_1 and p_2 activations to exhibit a longer execution time than p_0 , but still below the response time allowance. However, this behavior is shifting p_3 activation right after the generation of F_5 which ultimately causes F_3 to be dropped from the fusion buffer without being used.

The overall net effect of this underlying small jitter is losing the information in $M_{(2,2)}$ ³. This can have deep and bigger effects on the execution time behavior of the application, which can effectively affect the set of packets processed by the application.

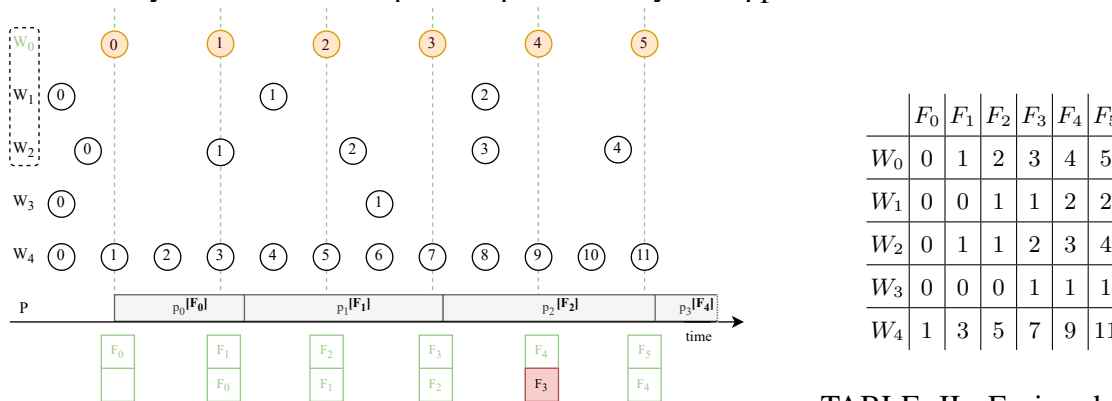


Fig. 11: Planning module execution diagram.

TABLE II: Fusion buffer states.

VI. RESULTS

A. Experimental setup

1) *Platform*: We run Apollo on an AMD Ryzen 7 1800X double-threaded eight-core processor and a GeForce GTX 1080 Ti GPU. The GPU in our configuration is discrete, similar to

³Note that while the impact of such scenario depends on the application semantics, it is arising without breaking P timing bounds.

those present in the latest automotive platforms such as NVIDIA Drive Pegasus [108]. High-performance GPUs employ high-bandwidth discrete memory to provide the required performance for memory-intensive workloads such as object detection and other DL workloads. Results have been collected inside a docker container. This configuration is similar to that obtainable in real vehicles, where the entire Apollo framework is run on top of a Linux operating system.

2) *Apollo processes under analysis*: We have performed a number of experiments supporting some of the insights in previous sections. In particular, we focus on Perception’s LiDAR segmentation process and the end-to-end processes of Prediction and Planning modules. We selected these three processes since, as explained in Section III-A and Table I, they build on DL models.

- The LiDAR segmentation process detects and segments out foreground obstacles like cars, trucks, bicycles, and pedestrians, taking as an input the point clouds from the LiDAR sensor.
- Prediction studies and predicts the behavior of all the obstacles detected by Perception. It receives obstacles’ data along with basic perception information, including positions, headings, velocities, and accelerations. Based in that, it generates predicted trajectories and their probabilities for those obstacles.
- Planning creates a safe and comfortable spatio-temporal trajectory for the autonomous vehicle to follow, using the obstacles’ predicted trajectories and the vehicle’s current state, route, and location.

We perform 1,000 runs of Apollo. Each run starts from the same initial system state, with no other application running concurrently. CyberRT implements *records*, a file format for storing message data similar to ROS bags. We generated one record file, recording a 35s run of Apollo obtained with the LGSVL simulator [109]. During the recording, Apollo faces a simple path: going straight and turning left, passing through two junctions. The vehicle crosses with around 20 vehicles. We used this record file to avoid possible variabilities coming from the simulator.

B. LiDAR segmentation

Table III shows the results for the LiDAR segmentation process. The first observation is that the number of messages processed across runs varies between 512 and 535 (see first column). Each message count has been observed in between 0.10% (once) and 29.10% (291 times) of the runs, with (i) 528-532 being the most observed number of messages processed (89.8% in total) and (ii) more than 5% of observations each, as shown in the second column. Aleatoric variation, as described in Section III-C, along with some platform effects external to Apollo (e.g., operating

NUM MESSAGES	OBSERVATION	MIN (s)	MAX (s)	MIN: AVG PER MSG (ms)	MAX: AVG PER MSG (ms)
512	0.10%	14.31	14.31	27.95	27.95
513	0.10%	14.71	14.71	28.68	28.68
518	0.10%	12.33	12.33	23.80	23.80
519	0.40%	13.68	15.62	26.35	30.10
520	0.10%	13.62	13.62	26.18	26.18
521	0.20%	13.31	13.53	25.54	25.96
522	0.10%	13.48	13.48	25.83	25.83
524	0.30%	12.95	13.87	24.71	26.46
525	0.90%	12.52	14.98	23.84	28.54
526	1.50%	12.30	14.10	23.39	26.80
527	2.50%	12.25	14.91	23.24	28.30
528	6.90%	12.19	14.60	23.08	27.65
529	14.20%	12.19	14.58	23.04	27.56
530	29.10%	12.26	14.58	23.12	27.51
531	27.80%	12.27	15.07	23.11	28.38
532	11.80%	12.28	14.91	23.08	28.04
533	3.60%	12.33	14.10	23.13	26.46
534	0.20%	12.53	12.97	23.46	24.28
535	0.10%	13.24	13.24	24.75	24.75

TABLE III: Execution time of the Perception’s LiDAR segmentation process taken from 1000 executions of Apollo.

system interrupts) triggered variations large enough to induce the aforementioned variations in terms of messages processed.

If we look into the aggregated execution time of this process in the whole run (i.e. to process all the messages), one could expect that, the larger the number of messages processed, the higher the execution time. However, as shown in the third and fourth column in the table (in seconds), such correlation is not observed and execution time ranges for different message counts overlap. In fact, for instance, the fastest run processing 528 or 529 messages takes 12.19 seconds, which is less than the time taken by any of the 63 runs with fewer messages (between 512 and 527). This relates to the fact that, small variations in the execution of the different jobs of

the process can lead to varying interference due to, for instance, concurrent access to memory with other processes of the application, or due to other jobs being scheduled (or not) in the same hardware thread, hence potentially flushing cache contents, which may occasionally cause LiDAR segregation jobs to take longer.

Finally, the two rightmost columns show the execution time needed per message for each different message count (in milliseconds). Again, no specific trend is observed other than a more significant difference between maximum and minimum values of the most observed message counts, which is expected since there are more instances to consider.

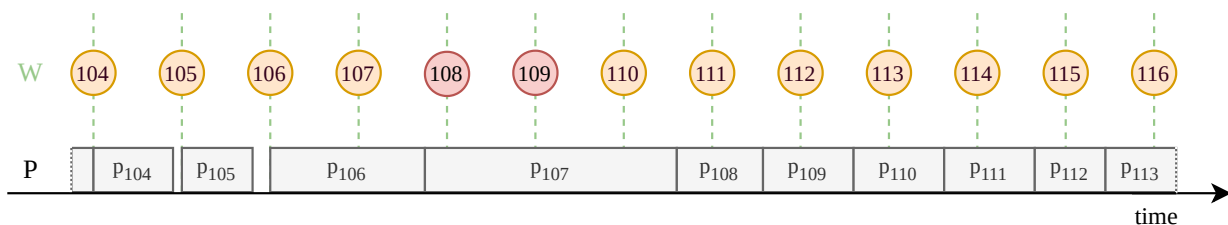


Fig. 12: Chronogram of one run of the LiDAR segmentation process.

Figure 12 shows a snapshot of one of the runs of the LiDAR segmentation process. In particular, we show the period with the arrival of messages between M_{104} and M_{116} , which arrive at a regular rate dictated by the LiDAR sensor. Messages are processed timely by the LiDAR segmentation process from the beginning of the run until M_{105} (inclusive). However, M_{106} takes longer to be processed by the processing job p_{106} . Then, p_{107} processes M_{107} much later than its arrival time (the time scale is irrelevant for this discussion). This already brings a concern that pure execution time analysis would hide: the freshness of M_{107} when processed is much worse than for previous messages, which are processed right after their arrival. Since p_{107} takes also long to process its message, p_{108} starts after the arrival of M_{110} , which has already replaced M_{108} and M_{109} . Those messages are lost because, in this case, the channel buffer can only store a single element. Again, looking only into execution times fails to capture the information related to the number of messages lost, and even looking into the number of messages lost does not capture whether those are lost sporadically or in bursts. Finally, from p_{109} onwards messages M_{111} onwards are processed with no further message lost, but we observe that those messages are processed systematically with some lag since their arrival, hence with worse freshness than messages until M_{106} . Such information is crucial since it relates to the

NUM MESSAGES	OBSERVATION	MIN (s)	MAX (s)	MIN: AVG PER MSG (ms)	MAX: AVG PER MSG (ms)
539	0.20%	3.87	4.37	7.19	8.10
540	1.80%	3.85	4.37	7.14	8.10
541	7.10%	3.80	4.39	7.03	8.11
542	16.20%	3.82	4.36	7.05	8.04
543	22.80%	3.81	4.22	7.01	7.77
544	19.40%	3.83	4.34	7.04	7.98
545	14.80%	3.83	4.19	7.03	7.69
546	8.10%	3.83	4.35	7.02	7.97
547	4.30%	3.86	4.25	7.06	7.78
548	3.30%	3.86	4.26	7.04	7.77
549	1.40%	3.85	4.10	7.01	7.48
550	0.30%	3.92	4.08	7.13	7.41
551	0.20%	3.90	3.94	7.08	7.15
552	0.10%	3.87	3.87	7.00	7.00

TABLE IV: Execution time of Prediction taken from 1000 executions of Apollo.

response time of the system (from sensing till actuation), which may affect QoS, or even safety if the response time exceeds the allowed bounds.

C. Prediction

Prediction process results are shown in Table IV. We observe less variability regarding the number of messages than for the LiDAR segmentation process (a range of 13 instead of 23 w.r.t. a similar total number of messages). Further, the 88.4% of the total observations is concentrated in 6 message counts, from 541 to 546, with more than 5% of the observations each, similarly to what we observed for the LiDAR segmentation process. Moreover, again, there is no correlation between the number of messages processed and execution time.

D. Planning

Table V shows the results of the Planning process. In this case, there is even less variability regarding the number of messages (between 525 and 531 except in one run with 511). And even

NUM MESSAGES	OBSERVATION	MIN (s)	MAX (s)	MIN: AVG PER MSG (ms)	MAX: AVG PER MSG (ms)
511	0.10%	24.89	24.89	48.71	48.71
525	0.20%	18.50	24.46	35.24	46.59
526	1.00%	23.63	28.93	44.93	54.99
527	10.00%	21.97	30.13	41.69	57.18
528	39.00%	21.63	31.07	40.97	58.84
529	36.40%	22.62	32.04	42.76	60.57
530	13.00%	23.51	30.63	44.36	57.79
531	0.30%	25.40	29.00	47.83	54.61

TABLE V: Execution time of Planning taken from 1000 executions of Apollo.

within that narrow message count range we observe that a narrower range (527-530) accounts for 98% of the runs. As for the other processes, no clear correlation exists between number of messages processed and execution time since neither the lowest execution time corresponds to the lowest number of messages processed, nor the highest execution time corresponds to the highest number of messages processed. In the case of Planning, we observe that, while almost all runs concentrate in between 525 and 531 messages processed (99.9% of the runs), one run was able to process only 511 messages. In this particular run, while the number of messages processed is clearly lower than in other runs, the total execution time is within the range of execution times observed for the most frequent cases (526-530, 99.4% of the runs). It is quite obvious that those measurements do not provide enough information to understand the potential additional implications of the number of messages lost.

E. Corollary

Overall, it follows that no meaningful conclusion can be derived on the comparison among different runs by just looking into execution times. However, some runs could provide higher quality results over time by being able to process more messages than others. Hence, this motivates the need for new metrics enabling a better analysis of the quality of different executions (e.g., when using different scheduling approaches). In fact, even results in the tables do not reflect whether messages are processed right after reception or short after being superseded by newer messages, with the former case being much better since the response time of the system is lower.

This is better illustrated in the chronogram in (Figure 12), where we see that the freshness of some messages at the time of being processed can get substantially degraded in comparison to others despite not being lost, with potential impact on the QoS, or even system safety. Therefore, a new family of metrics is needed for a thorough evaluation of autonomous driving frameworks.

VII. ON THE DEFINITION OF NEW METRICS

From a system perspective, we already observed how AD systems heavily depend on the correctness of the design of the propagation of data from the diverse sensors through the different modules and sub-modules, up to the actuators. In Section V we discussed how data-flow assumptions may be extremely fragile in practice, due to the impact on timing behavior of the different sources of variability and non-determinism, as identified in Section III and Section IV for the algorithmic and platform elements, respectively, see Figure 13. In particular, design-time assumptions on the absolute and relative frequency at which the different functionalities are executed (modules) may be unmet at operation because of the execution behavior of each (sub-)module.

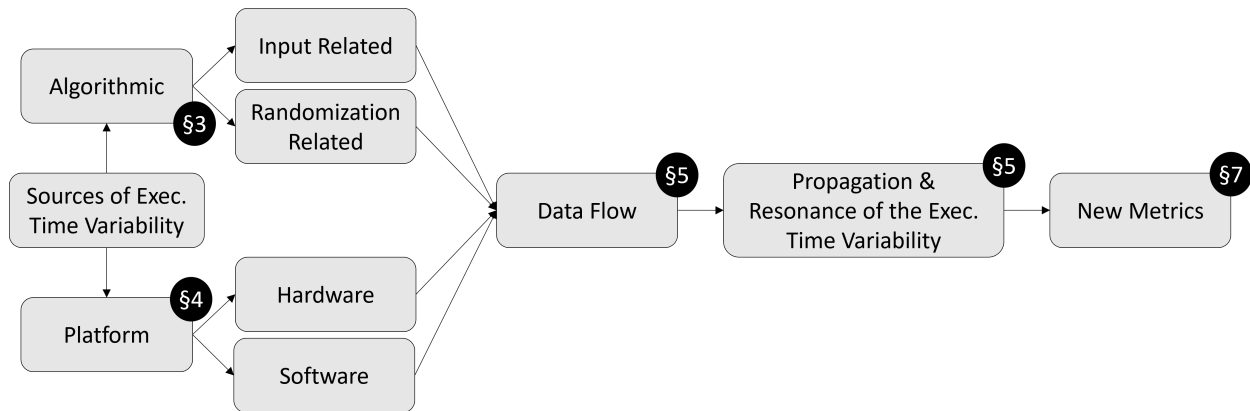


Fig. 13: Relation between source of execution time variability, its propagation and resonance, and the proposed metrics.

A. Communication Patterns

An important trait of AD frameworks is that the message-based frameworks they use generally supports both synchronous and asynchronous communication patterns.

Synchronous communication is modeled by imposing constraints on message aging: this is the most stringent type of requirement with respect to data-flow integrity as it will easily intercept

any misbehavior. However, synchronous communication is quite seldom enforced as considered too restrictive, given that sporadic, small deviations from the optimal frequency of operation may be allowed.

Asynchronous communication patterns are generally used to provide a block-free execution model where readers are simply retrieving the latest available message. Asynchronous message passing can hide data-flow integrity issues where (i) a message is not processed and propagated through the data-flow chain as a newer message has arrived, or (ii) a message is propagated well beyond its expected aging because the writer was not able to keep with the frequency assumptions. In the former case we may be dropping relevant information for the decision process, which is normal based on a status determined by recent message history. In the latter, we may be using outdated information to drive the decisions and, eventually, the actuators' actions.

With asynchronous communication timing misbehaviors can easily go unnoticed unless they cause negative consequences on the final system behavior (e.g. a car accident in the worst case). The fact is standard timing end-to-end metrics on the single modules or overall data-flow chain are not able to capture data-flow integrity violations unless they have catastrophic effects.

The problem at hand is to understand whether it is possible to define proper metrics to assess and evaluate data-flow requirements and assumptions on AD frameworks. Considering for example the data-flow relation between two modules writing and reading to the same channel: the reader can process a message at half the frequency of the writer or vice-versa. This can be a problem or not depending on whether this is a voluntary design feature or it is the consequence of execution time variability and non-determinism in the system.

We contend typical timing metrics as worst-case execution time and response time are not able to capture this type of misbehavior and, instead, they should be complemented by specific metrics based on data-flow and message events. Existing works, instead, mainly focus on end-to-end latencies [110], [111], [112], [113], [114] and only sporadically hint at other complementary metrics [115]. We identify the envisioned set of metrics below, distinguishing between module- and data-oriented ones. It is worth noting that the sought information is arguably accessible or can be derived from existing tracing and monitoring support [116].

B. Module-oriented metrics

This set of metrics focuses on the actual execution of the full AD chain and the involved functional modules. Such information is useful to capture end-to-end call context and execution rates for each module.

M1 Number of functions executed: captures the number of functions executed in a single end-to-end execution of a data-flow chain. This allows to relate response time to actual execution context since we may observe shorter response time just because some modules have not been executed.

M2 End-to-end execution of each module procedure: standard timing metric, allows to capture how sources of variability and non-determinism may directly affect the timing behavior of each module.

M3 Interval between module procedure activation: tracks the time between consecutive activations of a procedure module. In combination with message-oriented metrics, allows to intercept how inter- or intra-module variability affects execution time.

C. Data-oriented metrics

This set of metrics captures how data (messages) propagates through the data-flow chains, which is instrumental in understanding whether and how data-flow assumptions on relative and absolute execution rate are met.

M4 Channel refreshing rate: describes how frequently input data is produced by sensors or predecessor modules and injected into the data-flow. Such information is useful to comparatively assess message rate and module rate (M3). We are interested in min, max, avg, and std.

M5 Message persistence: tracks how long an input data value (specific message) can be propagating (hence used) in the chain of modules. Essentially this captures the maximum validity of a message once it enters the data flow. It allows to understand the impact of a message in the full AD chain. We are interested essentially in max, but also min, avg, and std could be informative.

M6 Number of dropped messages: derivative metric obtained by relating the number of messages produced and the number of messages read. Essentially, it tracks the number of message that are overwritten (or superseded) before being used. This information is useful to understand

whether part of the available information is disregarded, potentially leading to suboptimal decisions.

D. Summary

The above metrics are meant to provide complementary information to standard timing profiling to better capture the impact of variability, which does not necessarily (or not systematically) end up in a timing violation or evident misbehavior. In fact, we are expecting those metrics to show not only possible data-flow issues in the framework but also how timing variability and non-determinism affect and are magnified by data-flow relations and constraints.

For all the identified metrics we are interested in standard statistics such as min, max, mean or median but we are also interested in gathering more advanced metrics on quantile distributions. The use of more elaborated statistics on the collected measures are functional to the application of statistical timing analysis approaches [117], [118]. This family of approaches has been recently considered as a promising solution to cope with increasingly complex hardware and software architecture [119], [120], [63], like those deployed in the scope of AD solutions.

Furthermore, it is natural to consider the use of a subset of these metrics to guide system optimization. While end-to-end timing constraints are met, it may be desirable to optimize the system with respect for example shorter persistence or minimal message drop rate.

VIII. RELATED WORKS

Predictability and analyzability of the timing behavior are paramount concerns in the development of ADAS as well as fully AD solutions as the timing dimension increasingly plays a critical role in the overall system qualification and certification [7], [6]. In the following we consider related works on analyzability and predictability of AD frameworks deploying deep learning solutions, and on capturing the peculiarities of automotive functions from the timing perspective, in terms of end to end effects. We do not cover instead, related work on platform-related variability, which is already covered in Section IV.

AD frameworks, in particular, exhibit an exceptional degree of timing variability, which makes it more difficult to analyze and, ultimately, to successfully undergo the certification process [10]. A lot of effort has been devoted in the last decade to study timing variability and non-determinism in AD frameworks. The work in [9] provides a statistical analysis of the timing behavior of Apollo and reason on how the observed variability hampers the application

of conventional timing analysis methods. Deep NN implementation are a well-known source of timing variability: hardware and software solutions for predictable performance of NN deployed in AD frameworks and applications are reported in [13], [12], [11]. In particular, the use of advanced accelerator features in the Zynq Ultrascale [121] and the corresponding timing model are addresses in [11] and in [13], with the latter specifically focusing on Apollo performance. A software-level approach is, instead, proposed in [12] where a hypervisor layer is exploited to separate critical and non-critical functionalities in Apollo, with the critical ones being executed on top of a (more predictable) OSEK-compliant RTOS. The main focus in these works is on catering for better performance for NN-based automotive applications, while relying on a sufficiently accurate timing model. In a sense they focus on capturing timing variability, by analyzing or reducing it, while in this work we address the sources of such variability, including non-determinism, in AD frameworks.

The inherent characteristics of automotive systems have been addressed in several works aiming to conservatively model end-to-end response time of *cause-effect chains* of procedures in AD frameworks, with most of these works addressing ROS framework or its revamped version ROS2 [115], [110], [111], [112], [113], [114]. The main focus in these works is set on improving real-time aspects (e.g. [113]) while focusing on end-to-end latencies, beyond the typical concept of response time bounds, as a way to assess the correctness of the timing behavior automotive applications. The relevance of end-to-end delays, from an automotive and, more generally, distributed systems perspective, is advocated in [115] and further explored in [111], [112]. In [110], end to end delays are considered in relation to data freshness and used to formalize response time upper bounds for sporadic tasks. The implications on timing analyzability of the adopted middleware are explored in [113], [114], with specific proposals for improved timeliness and predictability in [113]. These works, however, are more focused on capturing how inputs propagate through the whole software architecture in relatively simpler automotive systems that do not suffer from the sources of variability and non-determinism we highlight in this paper.

IX. CONCLUSIONS

The rise of more advanced AI-based functionalities, and the use of more aggressive MPSoCs to deliver the required computing performance capabilities, is relentless in critical systems. This is driven by the huge potential benefits of software-controlled functionalities from increased energy efficiency to reduce accidents and fatalities in the roads and skies. The other side of the

coin is the challenge that AI-software and MPSoCs bring to well consolidated V&V in critical systems.

This paper provides a taxonomy of the different sources of variability brought by the use of AI on critical systems through the analysis of a representative AD framework. Our work shows that variability stems from different sources spanning across (i) the intrinsic behavior of AI software, (ii) the software frameworks, such as AD ones, where AI software is integrated into, and (iii) the platform characteristics. We show that the behavior of these elements is hard to predict – if at all possible – and, in some cases, purely random. We also show that those sources of variability are not independent among them and can interrelate in arbitrary ways, further challenging predictability. Building on those observations, we show that usual metrics related to execution time fall short to analyze the timing behavior of AI-based systems. Instead, other metrics considering other type of information such as, for instance, message lost and freshness of data processed emerge as complementary for a thorough analysis of AI-based systems and fair comparison of different design solutions.

DECLARATIONS

- **Funding.** This work has been supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GBC21/ AEI/10.13039/501100011033, and the European Research Council (ERC) grant agreement No. 772773 (SuPerCom).
- **Conflict of interest/Competing interests.** The authors have no relevant financial or non-financial interests to disclose.
- **Data availability statement.** The data supporting the results reported in this work are available from the authors upon reasonable request.

REFERENCES

- [1] A. J. Moshayedi, A. S. Roy, A. Kolahdooz, and Y. Shuxin, “Deep learning application pros and cons over algorithm,” *EAI Endorsed Transactions on AI and Robotics*, vol. 1, no. 1, 2 2022.
- [2] K. Cheon, J. Kim, M. Hamadache, and D. Lee, “On replacing pid controller with deep learning controller for dc motor system,” *Journal of Automation and Control Engineering*, vol. 3, pp. 452–456, 01 2015.
- [3] NVIDIA, “NVIDIA Orin Series System-on-Chip, technical reference manual,” march 2022, v1.0p.
- [4] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, “Xilinx adaptive compute acceleration platform: Versal™ architecture,” ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 8493. [Online]. Available: <https://doi.org/10.1145/3289602.3293906>
- [5] International Organization for Standardization, *ISO/PAS 21448. Road vehicles – Safety of the intended functionality*, 2019.

- [6] EASA, FAE, “General Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20). Amendment 23. Annex I to ED Decision 2022/001/R. AMC 20-193 Use of multi-core processors.” EASA, Tech. Rep., 2022.
- [7] International Organization for Standardization, *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
- [8] Certification Authorities Software Team, “Multi-core Processors - Position Paper,” CAST-32A, Tech. Rep., November 2016.
- [9] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, “Timing of autonomous driving software: Problem analysis and prospects for future solutions,” in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 267–280.
- [10] Lynx, *CHALLENGES BUILDING SAFE MULTICORE SYSTEMS*, 2020. [Online]. Available: <https://www.lynx.com/embedded-systems-learning-center/challenges-building-safe-multicore-mcp-software-systems>
- [11] F. Restuccia and A. Biondi, “Time-predictable acceleration of deep neural networks on fpga soc platforms,” in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 441–454.
- [12] L. Belluardo, A. Stevanato, D. Casini, G. Cicero, A. Biondi, and G. Buttazzo, “A multi-domain software architecture for safe and secure autonomous driving,” in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2021, pp. 73–82.
- [13] G. Sciangula, F. Restuccia, A. Biondi, and G. Buttazzo, “Hardware acceleration of deep neural networks for autonomous driving on fpga-based soc,” in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 406–414.
- [14] Baidu, “Apollo, an open autonomous driving platform.” <http://apollo.auto/>, 2018.
- [15] B. Templeton, “Baidu Unveils Ambitious Robotaxi Plan In China,” 2022. [Online]. Available: <https://www.forbes.com/sites/bradtempleton/2022/11/30/baidu-unveils-ambitious-robotaxi-plan-in-china/>
- [16] ApolloAuto, “Perception,” 2022. [Online]. Available: <https://github.com/ApolloAuto/apollo/tree/r5.0.0/modules/perception>
- [17] T. A. Foundation, “Autoware. An open autonomous driving platform.” <https://github.com/CPFL/Autoware/>, 2016.
- [18] M. Quigley et al., “ROS: an open-source Robot Operating System,” *ICRA Workshop on Open Source Software*, 2009.
- [19] S. Macenski, T. Foote, B. P. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Sci. Robotics*, vol. 7, no. 66, 2022. [Online]. Available: <https://doi.org/10.1126/scirobotics.abm6074>
- [20] Apex.AI, “Apex.OS. An end-to-end operating system for mobility, smart machines and IoT.” 2023. [Online]. Available: <https://www.apex.ai/apex-os>
- [21] RTI, “RTI Connex Drive. The leading safety-certified data-centric communications framework for software-defined vehicles.” 2023. [Online]. Available: <https://www.rti.com/drive>
- [22] Ü. Dogan, J. Edelbrunner, and I. Iossifidis, “Autonomous driving: A comparison of machine learning techniques by means of the prediction of lane change behavior,” in *2011 IEEE International Conference on Robotics and Biomimetics*. IEEE, 2011, pp. 1837–1843.
- [23] M. Al-Qizwini, I. Barjasteh, H. Al-Qassab, and H. Radha, “Deep learning algorithm for autonomous driving using googlenet,” in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 89–96.
- [24] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, “A survey of deep learning techniques for autonomous driving,” *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020.
- [25] M. Alloghani, D. Al-Jumeily, J. Mustafina, A. Hussain, and A. J. Aljaaf, “A systematic review on supervised and unsupervised machine learning algorithms for data science,” *Supervised and unsupervised learning for data science*, pp. 3–21, 2020.
- [26] ApolloAuto, “Apollo 3.0 Software Architecture,” 2018. [Online]. Available: https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_3.0_Software_Architecture.md

- [27] A. Smola and S. Vishwanathan, "Introduction to machine learning," *Cambridge University, UK*, vol. 32, no. 34, p. 2008, 2008.
- [28] H. Taud and J. Mas, "Multilayer perceptron (mlp)," in *Geomatic approaches for modeling land change scenarios*. Springer, 2018, pp. 451–455.
- [29] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*. Ieee, 2017, pp. 1–6.
- [30] Z. Peng, J. Yang, T.-H. Chen, and L. Ma, "A first look at the integration of machine learning models in complex autonomous driving systems: a case study on apollo," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1240–1250.
- [31] M. Liang and X. Hu, "Recurrent convolutional neural network for object recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3367–3375.
- [32] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, "Social lstm: Human trajectory prediction in crowded spaces," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 961–971.
- [33] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.
- [34] K. P. Murphy, *Machine learning - a probabilistic perspective*, ser. Adaptive computation and machine learning series, 2012.
- [35] E. Hamuda, B. Mc Ginley, M. Glavin, and E. Jones, "Improved image processing-based crop detection using kalman filtering and the hungarian algorithm," *Computers and electronics in agriculture*, vol. 148, pp. 37–44, 2018.
- [36] V. Lippiello, B. Siciliano, and L. Villani, "Adaptive extended kalman filtering for visual motion estimation of 3d objects," *Control Engineering Practice*, vol. 15, no. 1, pp. 123–134, 2007.
- [37] H. Tabani, R. Pujol, J. Abella, and F. J. Cazorla, "A cross-layer review of deep learning frameworks to ease their optimization and reuse," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, may 2020, pp. 144–145.
- [38] NimbleAI consortium, "NimbleAI: Ultra-energy efficient and secure neuromorphic sensing and processing at the endpoint." [Online]. Available: <https://www.nimbleai.eu/>
- [39] A. Der Kiureghian and O. Ditlevsen, "Aleatory or epistemic? does it matter?" *Structural safety*, vol. 31, no. 2, pp. 105–112, 2009.
- [40] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?" *arXiv preprint arXiv:1703.04977*, 2017.
- [41] E. Hüllermeier and W. Waegeman, "Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods," *Machine Learning*, vol. 110, no. 3, pp. 457–506, 2021.
- [42] A. Brando, "Aleatoric uncertainty modelling in regression problems using deep learning," 2022.
- [43] A. Brando, I. Serra, E. Mezzetti, J. Abella, and F. J. Cazorla, "Standardizing the probabilistic sources of uncertainty for the sake of safety deep learning," in *AAAI's Workshop on AI Safety*, 2023.
- [44] A. Ashukha, A. Lyzhov, D. Molchanov, and D. Vetrov, "Pitfalls of in-domain uncertainty estimation and ensembling in deep learning," *arXiv preprint arXiv:2002.06470*, 2020.
- [45] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *International conference on machine learning*. PMLR, 2017, pp. 1321–1330.
- [46] T. Oliphant, "NumPy: A guide to NumPy," USA: Trelgol Publishing, 2006–. [Online]. Available: <http://www.numpy.org/>
- [47] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7482–7491.

- [48] D. Feng, L. Rosenbaum, and K. Dietmayer, "Towards safe autonomous driving: Capture uncertainty in the deep neural network for lidar 3d vehicle detection," in *2018 21st international conference on intelligent transportation systems (ITSC)*. IEEE, 2018, pp. 3266–3273.
- [49] D. Huseljic, B. Sick, M. Herde, and D. Kottke, "Separation of aleatoric and epistemic uncertainty in deterministic deep neural networks," in *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 9172–9179.
- [50] M. H. Shaker and E. Hüllermeier, "Aleatoric and epistemic uncertainty with random forests," in *International Symposium on Intelligent Data Analysis*. Springer, 2020, pp. 444–456.
- [51] E. Hüllermeier, "Quantifying aleatoric and epistemic uncertainty in machine learning: Are conditional entropy and mutual information appropriate measures?" *arXiv preprint arXiv:2209.03302*, 2022.
- [52] T. Scheffer, C. Decomain, and S. Wrobel, "Active hidden markov models for information extraction," in *International Symposium on Intelligent Data Analysis*. Springer, 2001, pp. 309–318.
- [53] A. Brando, J. A. Rodríguez-Serrano, M. Ciprian, R. Maestre, and J. Vitrià, "Uncertainty modelling in deep networks: Forecasting short and noisy series," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 325–340.
- [54] M. Vault, "List of probability and statistics symbols," *Website [Internet]. [cited 26 May 2020]. Available: <https://mathvault.ca/hub/higher-math/math-symbols/probability-statistics-symbols/>*, 2020.
- [55] H. Pishro-Nik, "Mean squared error (mse)," *Website [Internet]. [cited 19 Sep 2020]. Available: <https://shorturl.at/tFGY4>*, 2020.
- [56] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate research*, vol. 30, no. 1, pp. 79–82, 2005.
- [57] T. Chai and R. R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)?—arguments against avoiding rmse in the literature," *Geoscientific model development*, vol. 7, no. 3, pp. 1247–1250, 2014.
- [58] D. J. MacKay, "Bayesian neural networks and density networks," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 354, no. 1, pp. 73–80, 1995.
- [59] C. M. Bishop, "Mixture density networks," 1994.
- [60] A. Brando, J. A. Rodriguez, J. Vitria, and A. Rubio Muñoz, "Modelling heterogeneous distributions with an uncountable mixture of asymmetric laplacians," *Advances in neural information processing systems*, vol. 32, 2019.
- [61] W. Dabney, M. Rowland, M. Bellemare, and R. Munos, "Distributional reinforcement learning with quantile regression," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.
- [62] N. Tagasovska and D. Lopez-Paz, "Single-model uncertainties for deep learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [63] A. Brando, I. Serra, E. Mezzetti, J. Abella, and F. J. Cazorla, "Using quantile regression in neural networks for contention prediction in multicore processors," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
- [64] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," *Advances in neural information processing systems*, vol. 30, 2017.
- [65] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural network," in *International conference on machine learning*. PMLR, 2015, pp. 1613–1622.
- [66] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*. PMLR, 2016, pp. 1050–1059.
- [67] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, "A survey of deep learning applications to autonomous vehicle control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 2, pp. 712–733, 2020.

- [68] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi, "Building timing predictable embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, pp. 82:1–82:37, 2014.
- [69] M. Schoeberl, S. Abbaspour, B. Akesson, N. C. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: time-predictable multi-core architecture for embedded systems," *J. Syst. Archit.*, vol. 61, no. 9, pp. 449–471, 2015.
- [70] Certification Authorities Software Team, *CAST-32A Multi-core Processors*, 2016.
- [71] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement," in *26th Euromicro Conference on Real-Time Systems, ECRTS, 2014*.
- [72] E. Díaz, E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla, "Modelling multicore contention on the aurixtm tc27x," in *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, 2018.
- [73] S. Schliecker, M. Negrean, and R. Ernst, "Bounding the shared resource load for the performance analysis of multiprocessor systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10, 2010, pp. 759–764.
- [74] D. Dasari and V. Nelis, "An analysis of the impact of bus contention on the wcet in multicores," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication*, ser. HPCC '12, 2012, pp. 1450–1457.
- [75] J. Jalle, J. Abella, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "AHRB: A high-performance time-composable AMBA AHB bus," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, 2014.
- [76] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Is your bus arbiter really fair? restoring fairness in AXI interconnects for FPGA SoCs," *ACM Trans. on Embedded Computer Systems*, vol. 18, no. 5s, pp. 51:1–51:22, 2019.
- [77] S. Lee, "Real-time wormhole channels," *Journal Of Parallel And Distributed Computing*, vol. 63, pp. 299–311, 2003.
- [78] J. Cardona, C. Hernandez, E. Mezzetti, J. Abella, and F. J. Cazorla, "NoCo: ILP-based worst-case contention estimation for mesh real-time manycores," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, dec 2018. [Online]. Available: <https://doi.org/10.1109/rtss.2018.00043>
- [79] S. Tobuschat and R. Ernst, "Real-time communication analysis for networks-on-chip with backpressure," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, mar 2017, pp. 590–595.
- [80] M. Becker, B. Nikolic, D. Dasari, B. Akesson, V. Nlis, M. Behnam, and T. Nolte, "Partitioning and analysis of the network-on-chip on a cots many-core platform," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 101–112.
- [81] Y. Qian, Z. Lu, and W. Dou, "Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip," in *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. IEEE, 2009, pp. 44–53.
- [82] M. Hassan and R. Pellizzoni, "Bounding DRAM interference in COTS heterogeneous mpsocs for mixed criticality systems," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, 2018.
- [83] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *RTAS, 2014*, pp. 155–166.
- [84] *ARINC Specification 653: Avionics Application Software Standard Standard Interface*, ARINC Inc., June 2012.

- [85] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, “A survey on cache management mechanisms for real-time embedded systems,” *ACM Computing Surveys*, vol. 48, no. 2, pp. 32:1–32:36, nov 2015.
- [86] Arm, *ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service Supplement to ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual*, 2017.
- [87] —, *ARM CoreLink QVN-400 Network Interconnect Advanced Quality of Service using Virtual Networks Supplement to ARM CoreLink NIC-400 Network Interconnect Technical Reference Manual*, 2017.
- [88] Arm, *Arm Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A*, 2022.
- [89] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, “Leveraging hardware qos to control contention in the xilinx zynq ultrascale+ mpso,” in *33rd Euromicro Conference on Real-Time Systems, ECRTS 2021, July 5-9, 2021, Virtual Conference*, ser. LIPIcs, vol. 196, 2021, pp. 3:1–3:26.
- [90] Falk Rehm and Jrg Seitter, “Software Mechanisms for Controlling QoS,” in *2021 Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Virtual Conference, February 01-05, 2021*, 2016, pp. 1485–1488.
- [91] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, “E-WarP: a System-wide Framework for Memory Bandwidth Profiling and Management,” in *RTSS*, 2020.
- [92] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha, “Coscheduling of cpu and i/o transactions in cots-based embedded systems,” in *2008 Real-Time Systems Symposium*, Nov 2008, pp. 221–231.
- [93] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, “A predictable execution model for cots-based embedded systems,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 269–279.
- [94] A. Biondi and M. Di Natale, “Achieving predictable multicore execution of automotive applications using the LET paradigm,” in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, April 2018, pp. 240–250.
- [95] S. Mittal, “A survey of techniques for cache partitioning in multicore processors,” *ACM Computing Surveys*, vol. 50, no. 2, pp. 27:1–27:39, jun 2017.
- [96] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms,” in *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2013*, pp. 55–64.
- [97] H. Aghilinasab, W. Ali, H. Yun, and R. Pellizzoni, “Dynamic Memory Bandwidth Allocation for Real-Time GPU-Based SoC Platforms,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, 2020.
- [98] F. Farshchi, Q. Huang, and H. Yun, “BRU: bandwidth regulation unit for real-time multicore processors,” in *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*, 2020, pp. 364–375. [Online]. Available: <https://doi.org/10.1109/RTAS48715.2020.00011>
- [99] M. Pagani, E. Rossi, A. Biondi, M. Marinoni, G. Lipari, and G. C. Buttazzo, “A bandwidth reservation mechanism for axi-based hardware accelerators on fpgas,” in *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany*, ser. LIPIcs, vol. 133.
- [100] J. Cardona, C. Hernández, J. Abella, and F. J. Cazorla, “Maximum-contention control unit (MCCU): resource access count and contention time enforcement,” in *Design, Automation & Test in Europe Conference & Exhibition, DATE*, 2019, pp. 710–715.
- [101] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith, “Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems,” in *RTSS*, 2016, pp. 57–68.

- [102] G. Gracioli and A. A. Fröhlich, “On the design and evaluation of a real-time operating system for cache-coherent multicore architectures,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 49, no. 2, pp. 2–16, 2015.
- [103] LYNX Software technologies, “LynxSecure Separation Kernel Hypervisor,” 2022. [Online]. Available: <https://www.lynx.com/products/lynxsecure-separation-kernel-hypervisor>
- [104] Sysgo, “PikeOS product overview,” https://www.sysgo.com/fileadmin/user_upload/data/flyers_brochures/SYSGO_PikeOS_Product_Overview.pdf, 2021.
- [105] Siemens, “Jailhouse Hypervisor,” <https://github.com/siemens/jailhouse>, 2022, [Online; accessed 24-February-2022].
- [106] A. Crespo, I. Ripoll, and M. Masmano, “Partitioned embedded architecture based on hypervisor: The XtratuM approach,” in *European Dependable Computing Conf. (EDCC)*, 2010, pp. 67–72.
- [107] ApolloAuto, “CyberRT,” <https://github.com/ApolloAuto/apollo/tree/master/cyber>, 2021. [Online]. Available: <https://github.com/ApolloAuto/apollo/tree/master/cyber>
- [108] NVIDIA, “NVIDIA DRIVE Hyperion 7.1,” 2023. [Online]. Available: <https://developer.nvidia.com/drive/hyperion-7.1>
- [109] L. Electronics, “Svl simulator: An end-to-end autonomous vehicle simulation platform,” 2022. [Online]. Available: <https://www.svlsimulator.com/>
- [110] M. Dürr, G. von der Brüggen, K. Chen, and J. Chen, “End-to-end timing analysis of sporadic cause-effect chains in distributed systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, pp. 58:1–58:24, 2019.
- [111] S. Mubeen and T. Nolte, “Applying end-to-end path delay analysis to multi-rate automotive systems developed using legacy tools,” *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*, vol. 2015, 07 2015.
- [112] J. Schlatow and R. Ernst, “Response-time analysis for task chains in communicating threads,” in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, 2016, pp. 245–254.
- [113] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, “ROSCH: real-time scheduling framework for ROS,” in *RTCSA*, 2018, pp. 52–58.
- [114] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, “A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance,” in *RTSS*, 2021, pp. 41–53.
- [115] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, “A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics,” in *RTSS - Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2008, pp. 41–53.
- [116] Z. Li, A. Hasegawa, and T. Azumi, “Autoware_perf: A tracing and performance analysis framework for ROS 2 applications,” *J. Syst. Archit.*, vol. 123, p. 102341, 2022.
- [117] H. Zeng, M. D. Natale, P. Giusto, and A. L. Sangiovanni-Vincentelli, “Using statistical methods to compute the probability distribution of message response time in controller area network,” *IEEE Trans. Ind. Informatics*, vol. 6, no. 4, pp. 678–691, 2010.
- [118] E. Kang and L. Huang, “Probabilistic analysis of timing constraints in autonomous automotive systems using simulink design verifier,” in *SETTA*, ser. Lecture Notes in Computer Science, vol. 10998. Springer, 2018, pp. 170–186.
- [119] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, “Timing of autonomous driving software: Problem analysis and prospects for future solutions,” in *RTAS. IEEE*, 2020, pp. 267–280.
- [120] S. Vilardell, I. Serra, E. Mezzetti, J. Abella, F. J. Cazorla, and J. del Castillo, “Using markovs inequality with power-of-k function for probabilistic wcet estimation,” in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
- [121] XILINX, “Zynq UltraScale+ Device. Technical Reference Manual. UG1085 (v2.1),” 2019.