Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions

Miguel Alcon*,†, Hamid Tabani*, Leonidas Kosmidis*, Enrico Mezzetti*, Jaume Abella*, Francisco J. Cazorla*
*Barcelona Supercomputing Center (BSC)
†Universitat Politecnica de Catalunya

***Abstract*—The software used to implement advanced functionalities in critical domains (e.g. autonomous operation) impairs software timing. This is not only due to the complexity of the underlying high-performance hardware deployed to provide the required levels of computing performance, but also due to the complexity, non-deterministic nature, and huge input space of the artificial intelligence (AI) algorithms used. In this paper, we focus on Apollo, an industrial-quality Autonomous Driving (AD) software framework: we statistically characterize its observed execution time variability and reason on the sources behind it. We discuss the main challenges and limitations in finding a satisfactory software timing analysis solution for Apollo and also show the main traits for the acceptability of statistical timing analysis techniques as a feasible path. While providing a consolidated solution for the software timing analysis of Apollo is a huge effort far beyond the scope of a single research paper, our work aims to set the basis for future and more elaborated techniques for the timing analysis of AD software.**

I. Introduction and Motivation

The provision of increasingly advanced (and complex) software functionalities, e.g. Autonomous Driving (AD), is a key competitive advantage in every new product in the critical embedded market attracting significant interest from industry [41, 24, 56]. Supporting those advanced software functionalities requires complex software frameworks capable of performing real-time processing of a massive amount of diverse data, consistently coming from a score of on-board sensors, like cameras and LiDARs, just to mention a few. Moreover, those data are inherently involved in the critical AD decision-making process, from perception to planning, for which advanced AI algorithms are sought [48, 45].

The effectiveness and scalability of traditional Verification and Validation (V&V) approaches are threatened by the complexity and unboundedness of the input and result spaces of functionalities such as perception and tracking [55, 6]. The untenable number of potential inputs from the operational environment, and the non-deterministic nature of decision-making algorithms, complicate the definition of worst-case scenarios in both functional and non-functional dimensions [55]. As a result, it is hard to define budgets for software timing, relevant criteria for software timing V&V, and adequate testing methodologies.

To illustrate AD software's extremely variable timing behavior, Figure 1 shows boxplot diagrams of the observed per-frame execution time variability (jitter) of each of the modules of an AD framework, Apollo [1], when running under a representative set of inputs on our GPU-based target platform, see Section II-B. Boxplot diagrams show the median, the quantiles, maximum, minimum values and outliers across different executions. The observed jitter (max vs. min) is vast across all modules, up to 21x (and 6.1x on average). To make things worse, the execution times present arbitrary distributions that vary across modules. This is illustrated in Figure 2 that shows the histogram (bars) and the cumulative distribution function or CDF (line) of observed execution times, required to process each frame, for two software modules of Apollo. The x-axis shows execution times (in milliseconds), the left y-axis the frequency of occurrence for the histogram (for a 1,000 observations sample), and the right y-axis the fraction of observations for the CDF.

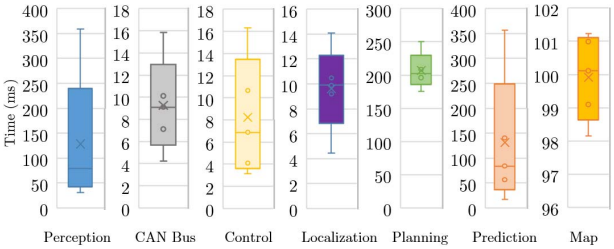


Fig. 1: Observed per-module execution time of Apollo

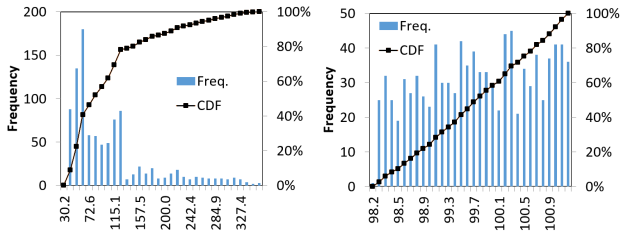


Fig. 2: Execution time (ms) analysis of two Apollo modules.

The unconventional amount and distribution of execution time values exhibited by Apollo modules is largely determined by the inherent variability of the deployed algorithm, though it is also caused by the complexity of the hardware platforms necessary to sustain the performance and timing requirements of the intended functionalities. Both hardware and functional complexity result in scenarios not easily analysable with prevailing software timing analysis methods [57]. This occurs because such complexity undermines the accuracy and scalability of static analyses and the significance of measurement-based approaches [3].

In this line, contributing to the state of the art with an

2642-7346/20/$31.00 ©2020 IEEE
DOI 10.1109/RTAS48715.2020.000-1

efficient software timing solution for AD software frameworks like Apollo is an overwhelming objective that will still require long-term efforts by the community to be designed and developed. In this paper, we perform an analysis of the execution time variability of Apollo when run on a GPU-based platform and reason on the sources behind the observed variability. We also present some directions in designing a software timing solution for Apollo, and analyze the effectiveness of statistical-based timing analysis approaches to address such timing variability, together with their application to Apollo's software running on a high-performance platform. These contributions are meant to offer a solid baseline for future works to master Apollo's execution time variability.

Analysis. We make an in-depth analysis of the jitter exhibited by Apollo [1] when run on a GPU-based high-end system. We focus on the jitter in per-frame processing time of each Apollo's module. To our knowledge, this is the first attempt to apply timing analysis to Apollo or to any AD software framework of such scale and complexity. Our results show that the observed variability is huge and also intrinsic across modules from perception to planning. For the YOLO-based object detection [51] module, on which we performed a deeper analysis, our results show that observed jitter does not only happen at the end-to-end module level but also at the main stage (i.e. functions) level of the module.

Reasoning. We analyze some of the sources of non-determinism of Apollo with emphasis on Apollo's built-in randomization features. We analyze an example function in Apollo, RANSAC, that instantiates specific randomization features. We show that RANSAC, in fact, combines deterministic input parameters and randomized input values, generated within the function itself. The observed variability in RANSAC's execution time caused by each of these factors is on average 241x and 80x, respectively. Instead, the inherent variability of the platform, e.g., due to hardware and software initial state, is significantly lower, nearly 5% on average.

Prospects. We focus on the analysability of Apollo as a representative example of a class of AD software frameworks. We highlight how its software characteristics, in combination with the complex hardware platform (required to sustain the framework performance requirements), are not comparable with conventional embedded critical systems. In particular, we discuss how randomness, huge input space, and execution scenarios make it difficult apply established timing analysis approaches [57, 58, 52, 3].

Statistical support. Recent statistical approaches to timing analysis [14, 20, 35, 9] seem to offer a promising alternative to conventional deterministic techniques, when coping with the inherently huge variability and non-determinism of AD solutions, as those deployed in the Apollo framework. The timing behavior of AD systems is better described by an execution time distribution rather than a single value. The intuition behind the application of probabilistic techniques in the analysis of Apollo timing builds on considering the timing behavior of a program as the outcome of a random variable, so that it can be reasoned in terms of probability of occurrence.

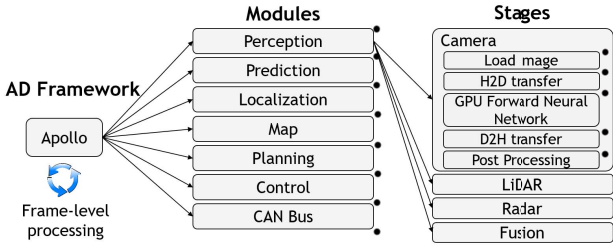


Fig. 3: Apollo AD system pipeline. Dots indicate the instrumentation points we use for extract timing behavior.

In this line, we discuss the scope of applicability of statistical approaches by considering their requirements and inherent limitations. In particular, we focus on a specific incarnation of Measurement-Based Probabilistic Timing Analysis (MBPTA) based on Extreme Value Theory (EVT) [18] and review its preconditions for applicability. MBPTA cannot be directly applied or cannot deliver reliable results whenever: (i) the inputs and execution conditions under which execution time observations are collected are not representative, or upper-bound those that can arise during operation time; and (ii) input samples (observations) do not comply with required statistical properties, such as, for example, independence and identical distribution (i.i.d).

In the scope of this work, we argue that those properties are either naturally met by Apollo when executing in our target platforms or need to be enforced at the level of the overall test-design strategy for the functional behavior. We also show that probabilistic timing analysis can accurately upper-bound the observed execution time distributions of Apollo industrial software when run on a real platform.

The rest of the paper is organized as follows: Section II introduces Apollo and our experimental framework. Section III presents the outcome of our analysis of the variability exhibited by Apollo. Section IV analyzes some of Apollo's sources of non-determinism behind the observed variability and also discusses how they impair software timing analysis. Section V discusses the application of a specific statistical technique for Apollo timing analysis. Section VI summarizes the related works. Section VII presents the main conclusions of our work.

II. Apollo and Our Experimental Setup

A. Apollo AD Framework

Apollo [1] is an industrial-quality AD software framework with over 110 industrial partners, most of them top-tier AI companies and car manufacturers. Apollo has been already deployed on a variety of prototype vehicles (including autonomous trucks) and supports state-of-the-art hardware such as the latest LiDARs and cameras, from Velodyne and other vendors, as well as GPU acceleration.

Apollo is structured as a set of processes that are meant to execute on a recurring basis. The execution model is organized into stages where each stage is allocated to a specific functional step in each module. Figure 3 presents the main software modules in Apollo:

- **Perception** identifies the area surrounding the autonomous vehicle by detecting objects, obstacles, and, traffic signs. It is considered as the most critical and complex module of an AD system. It fuses the output of several types of sensors such as LiDAR, radar, and camera to improve its accuracy.
- **Localization** estimates where the autonomous vehicle is located using various information sources such as GPS, LiDAR and IMU. State-of-the-art localization algorithms, including the one in Apollo, are capable of localizing the position of the vehicle at centimeter-level accuracy.
- **Prediction** anticipates the future motion trajectories of the perceived obstacles.
- **Planning** plans the spatio-temporal trajectory for the autonomous vehicle to take.
- **Control** executes the planned spatio-temporal trajectory by generating control commands such as acceleration, braking, and steering.
- **CanBus** is the interface that passes control commands to the vehicle hardware. It also passes chassis information to the software system.
- **Map** acts like a library. Instead of publishing/subscribing messages, it works as a query engine support that provides ad-hoc structured information regarding the roads.

One of the key components of the AD framework is the perception module, given (i) its complexity, instrumental to deal with inputs from various components (e.g., video cameras, short- and long-range radars and laser sensors), and (ii) its long execution time that represents a large fraction of the overall execution time of the AD framework. For this reason, we focus on this module to perform the stage-level analysis.

Apollo uses a variant of YOLO [51] for camera-based object detection, as the main part of the perception module. YOLO (You Only Look Once) is an award-winning, widely-used object detection system. Its most computationally-intensive function is a Convolutional Neural Network inference algorithm. The main stages of the YOLO object detection module are shown in Figure 3 as camera macro-stages. Every second, each camera captures multiple frames, and the object detector processes them on a frame-by-frame basis.

- For every frame the detector first *loads* the frame (in an appropriate format) into the main memory.
- Then, all the data is moved to the GPU memory (*host-to-device transfer*).
- Once the data is stored in the GPU memory, *GPU kernels* are launched to perform the neural network evaluation.
- The result of the operations is transferred back to the main memory (*device-to-host* transfer).
- As the last step, some *post-processing operations* are performed to finalize and publish the result of the detection.

Note that camera-based object detection is part of the perception module, which is fused with the LiDAR-based object detection and Radar processes.

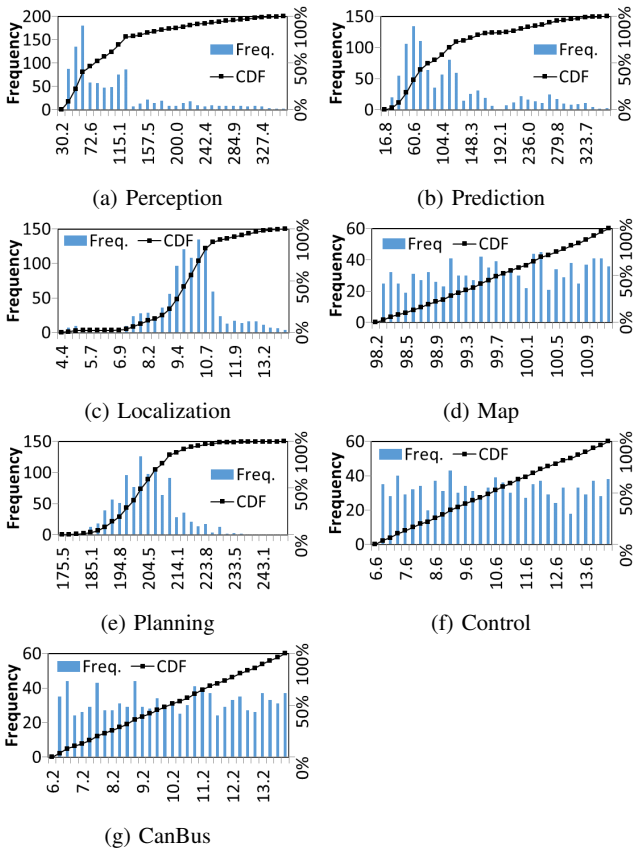
B. Experimental Platform

We run Apollo on an x86 platform using 8 AMD Ryzen 7 1800X CPU cores and 64 GB of DDR4 RAM at 2133 MHz. In order to satisfy the computational needs of Apollo, our platform is equipped with a Pascal-based high-end GPU (the NVIDIA GeForce 1080 Ti with 3584 CUDA cores). While drastically different from traditional automotive architectures such as the AURIX TriCore, the selected target platform resembles state-of-the-art automotive Systems on Chip (SoCs) targeting the automotive AD market. For example, the two variants of the NVIDIA Drive PX2 platform, AutoCruise and AutoChauffeur, have similar CPU and GPU configurations. The former comprises a single Tegra X2 SoC, which contains 4 ARM Cortex-A57 and 2 Denver cores, combined with an integrated Pascal GPU. The latter contains two Tegra X2 SoCs and 2 discrete Pascal-based GPUs. Moreover, the ARM A57 CPUs used in these platforms exhibit similar hardware complexity as that of the x86 cores in our platform, since both are superscalar, out-of-order CPUs, with several levels of cache. Note that the GPU in the automotive platforms is integrated, i.e. both devices share the same memory, whereas our GPU is discrete, thus requiring data transfers. However, we have verified that data transfers account for less than 1% of the total execution time of Apollo. Therefore, the multiprocessing capabilities in the CPU side and the GPU architecture of our platform are representative of the automotive domain.

Due to the software dependencies of Apollo, the framework is executed on a Linux environment and it is built on top of ROS (Robotic Operating System) [50]. In order to minimize the jitter stemming from outside of the application, i.e. from the operating system or hardware behavior, we follow standard guidelines for real-time execution under Linux. In particular, we minimize the running services of the system to the bare minimum, stopping services such as mail services or the window manager. In addition, we assign real-time priorities from the Linux kernel to all scheduled tasks under analysis. We have further pinned tasks on specific cores in order to prevent costly task migration and remap all interrupts to a separate core not assigned to any real-time task. As we discuss in Section IV, this execution configuration results in a relatively low platform jitter, and it is the same configuration used for both measuring the platform variability and running Apollo.

C. Instrumentation Details

For time measurements, we used instrumentation points at module and stage boundaries, at the granularity shown in Figure 3. For modules using the CPU only, we use the `high_resolution_clock` of C++, which provides a high-resolution time counter. On the stages that use the GPU, such as camera-based and LiDAR-based object detection in Perception, we use NVIDIA CUDA events, which provide a reliable, high-resolution time counter for GPU tasks. This measurement method can account for the fact that GPU tasks are asynchronous to the CPU side without affecting the performance and timing of the software under analysis, which is not possible with regular CPU time counters. CPU counters



(a) Perception (b) Prediction

(c) Localization (d) Map

(e) Planning (f) Control

(g) CanBus

Fig. 4: Distribution of execution times (ms) of each module.

cannot be used for measuring the execution time of GPU tasks, because when a GPU task is called, the execution returns back to the CPU immediately, while the GPU executes the task in parallel. Hence, our instrumentation for obtaining time measurements can be regarded as having low overhead.

III. Analyzing Apollo's Execution Time Jitter

Next we report the main statistical properties of the observed execution time distributions. The main conclusions in this section are later used in Section V. We perform our analysis at module level except for the Perception module, where we perform a more detailed analysis at the stage level.

A. Module level analysis

We measure execution times at frame-level and capture the resulting distribution for each of the modules when they process real tracing data collected by autonomous car sensors.

Figure 4 shows the observed execution time histogram and CDF of each Apollo module as described in the introduction for Figure 2. As it can be seen, jitter distributions have different shape and dispersion, hampering their analysis. This phenomenon is quantified in the Table I that shows different measures of dispersion that allow us analyzing and comparing the distributions. It shows:

- (1) Minimum and (5) maximum values observed in the execution time sample.

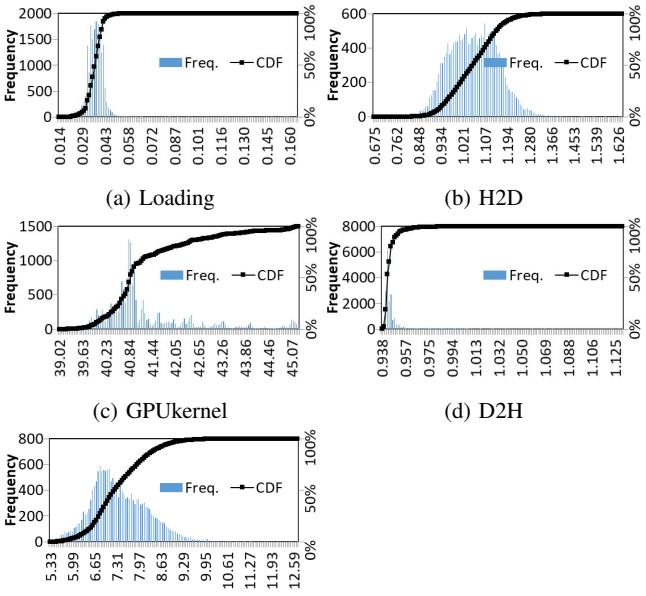
TABLE I: Measures of dispersion for Apollo modules. Values in the first 5 rows are in milliseconds.

	Per	Pred	Loc	Map	Plan	Con	CAN
(1) Min	30.2	16.8	4.4	98.2	175.5	6.6	6.2
(2) Q1	53.0	56.4	9.2	99.1	196.1	8.6	8.2
(3) Q2	78.6	84.0	9.9	99.8	202.3	10.4	10.1
(4) Q3	120.8	140.5	10.5	100.5	208.8	12.2	12.0
(5) Max	359.2	356.6	14.1	101.2	250.4	14.3	13.9
(6) CV	0.69	0.69	0.15	0.01	0.05	0.21	0.22
(7)IQRn	0.86	1.00	0.13	0.01	0.06	0.35	0.38
(8) Kurt	1.65	0.59	2.31	-1.13	1.22	-1.16	-1.19
(9) Var	11.9	21.2	3.5	1.03	1.4	2.1	2.2

- (2) Quantiles Q1, (3) Q2 and (4) Q3. Quantiles are cut points dividing the range of a probability distribution into intervals: Q1 (25% below and 75% above), Q2 (50% below and 50% above), and Q3 (75% below and 25% above).
- (6) The estimated coefficient of variation CV that provides the ratio between standard deviation (σ) and the mean (μ) of the sample. Thus, values close to 0 indicate that the standard deviation is very low in relative terms, whereas high values (e.g., above 0.5) indicate high variability.
- (7) The Inter-Quantile Range normalized ($IQRn$) that provides similar relative information since $IQRn = (Q3 - Q1)/\mu$, but focusing only on the central 50% of the values.
- (8) The excess kurtosis ($Kurt$) that provides information on whether tail values (those below $\mu-\sigma$ and above $\mu+\sigma$) are abundant and distant from the mean: $Kurt < 0$ indicates that tail values are less significant than in a Gaussian distribution, thus closer to the mean and/or less frequent; and $Kurt > 0$ that outliers are abundant and/or distant from the mean. For instance, $Kurt = -1.2$ suggests a uniform distribution since tails are bounded.
- And (9) the variation between the max. and min. values.

Based on Table I, we derive the following conclusions:

1) The observed variability (*Var*) between the minimum and maximum recorded execution times is high (up to 21x for Prediction), above 2x for 5 out of 7 modules, and low only for the Map module. Moreover, Q3 is 2x higher than Q1 for the 2 modules with the highest maximum execution time (Perception & Prediction).
2) The CV is low only for 2 modules (Map and Planning), moderate for 3, and very high for 2 (Perception and Prediction). For the latter modules, a high CV indicates that values are highly spread, both central and tail values.
3) $IQRn$ shows that the dispersion of the central values is huge for two modules (0.8-1.0) and moderate for another 2 (around 0.35), thus indicating that dispersion is relevant even for the central part of the distribution.
4) $Kurt$ is high for 4 modules. While this is irrelevant for Localization, since values are relatively low, it is quite relevant for Perception, Prediction, and Planning, whose dispersion and execution time are high. High $Kurt$ for those modules indicates that extreme values are abundant or significant. By analyzing minimum and maximum values, we see that those values are far beyond Q1 and Q3, so that we can expect gentle slopes in their tails.



(a) Loading (b) H2D

(c) GPUkernel (d) D2H

(e) PosPro

Fig. 5: Execution time (ms) distribution of YOLO stages.

B. Stage level analysis

We extend our jitter analysis at the stage level for the Perception module, as representative of the complexity of AD software. Our goal is analyzing whether large jitter is caused by just few stages while the others are exhibiting a much narrower jitter distributions – so statistical analysis is required only for those few stages, whereas conventional timing analysis techniques can be used for the rest. In this experiment, we profile the per-frame processing time of each stage in the Perception (YOLO).

As it can be seen in Figure 5 and Table II, all stages follow the observed trend at the module level as they exhibit significant jitter and have arbitrarily different distributions:

1) Loading, Host-to-Device (H2D), and Device-to-Host (D2H) stages have very low execution times in relative terms, so even if dispersion is very high for some of them, this is irrelevant in practice.
2) GPU kernels show tiny dispersion in the central part ($IQRn = 0.02$), but very large relative dispersion in the tails ($Kurt = 2.09$). While the maximum is far away from the central part of the distribution in relative terms, it is close in absolute terms (11% higher than the median), so dispersion in absolute terms is low.
3) Post-processing (PosPro) has moderately high dispersion in both, the central part of the distribution and the tails. Hence, a gentle slope is expected for its upper tail.

Figure 3 shows that the perception module consists of 4 main submodules, being Object Detection for the camera just one of them. The others are Object Detection and tracking for LiDAR, sensor fusion and radar. Hence, camera object detection execution time, see Figure 5 and Table II, is lower than the overall perception execution time.

TABLE II: Measures of dispersion (YOLO stages). Values in the first 5 rows are in milliseconds.

	Loading	H2D	GPU	D2H	PPro
(1) Min	0.01	0.68	39.02	0.94	5.33
(2) Q1	0.03	0.99	40.61	0.94	6.77
(3) Q2	0.04	1.06	40.86	0.94	7.24
(4) Q3	0.04	1.13	41.60	0.94	7.92
(5) Max	0.16	1.64	45.25	1.13	12.70
(6) CV	0.14	0.09	0.03	0.01	0.11
(7)IQRn	0.14	0.13	0.02	0.00	0.16
(8) Kurt	113.2	0.01	2.09	147.8	0.38
(9) Var	16	2.4	1.2	1.2	2.4

C. Summary

Effectively deriving timing bounds requires methods to model highly variable execution times. Those methods must not impose constraints on the distribution since the observed jitter distributions present arbitrary shape and dispersion.

IV. Sources of Execution Time Variability and Impact Software Timing

A. Reasoning on Apollo's variability

Several well-known sources of execution time variability exist in modern critical embedded systems. We categorize them as follows.

Platform. At hardware level, they relate to the use of complex heterogeneous high-performance platforms based on GPUs or FPGAs, e.g., the NVIDIA Drive PX2 platform or the Intel GO platform. Complex System-on-Chip might make execution time to vary due to their initial state dependence, which is hard to control. At software level, low-level drivers (e.g., CUDA driver) and the operating system (e.g., memory location of the actual buffers used for inter-thread communication), can also keep some internal state, thus affecting execution time [13] (more details on our hardware/software configuration are shown in Section II-B). Also, the execution of each function in Apollo modules can take a variable execution time, causing variation in the way modules overlap in time. The net result is that the set of instructions of each module that overlap in time and compete for resource varies across frames.

Randomization and non-determinism are inherent traits of several machine learning based state-of-the-art AD algorithms, which differentiates them from conventional software solutions [16, 1, 11, 17, 49]. In fact, non-determinism is necessary for the AD functionality to take rapid and efficient decisions. For example, randomized path planners are a common approach to cope with the complexity of exhaustive, deterministic path planning [22]. Randomization is also used in the Probabilistic Roadmap Method (PRM) and Rapidly-exploring Random Trees (RRT), where random selection of configurations is a core step in the rapid generation of planning solutions [22]. Also in perception, either model- or graph-based segmentation algorithms usually incorporate randomization elements [47]: a clear example of the former is RANSAC (Random Sample Consensus); when it comes to the latter, two

1 **RANSAC** (*Input:* Vector V of vectors of size 2, integers *maxIters* and N,
2 float *inlierThresh*,
3 *Output: vector C of size 4*)
4 Let n = size(V), $q_1 = \lfloor n/4 \rfloor$, $q_2 = \lfloor n/2 \rfloor$ and $q_3 = \lfloor 3 \cdot n/4 \rfloor$
5 **If** $n < N$ **then** throw an error and **return** False
6 **For** $j = 1, 2, ..., maxIters$ **do**
7 Generate randomly r_1, r_2, r_3, between 0 and $2^{31} - 1$
8 Let $x_1 = r_1 \pmod{q_2}$, $x_2 = q_2 + r_1 \pmod{q_1}$, $x_3 = q_3 + r_1 \pmod{q_1}$
9 Initialize matrices A and B as follows:
10

$$A = \begin{bmatrix} V[x_1,0] \cdot V[x_1,0] & V[x_1,0] & 1 \\ V[x_2,0] \cdot V[x_2,0] & V[x_2,0] & 1 \\ V[x_3,0] \cdot V[x_3,0] & V[x_3,0] & 1 \end{bmatrix},$$

$$B = \begin{bmatrix} V[x_1,1] & V[x_2,1] & V[x_3,1] \end{bmatrix}$$

11
12 Let vector c = findSolution(c, colPivHouseholderQr(A) $\cdot\, c = B$),
13 *Inliers* $= 0$, *res* $= 0$ and $y = 0$
14 **For** $i = 1, 2, ..., n$ **do**
15 $y = V[i,0]^2 \cdot c[0] + V[i,0] \cdot V[i,0] \cdot c[1] + c[2]$
16 **If** $|y - V[i,1]| \leq$ *inlierThresh* **then** $++$*Inliers*
17 *res* $+= |y - V[i,1]|$
18 **If** *Inliers* $>$ *maxInliers* or (*Inliers* $=$ *maxInliers* and *res* $<$ *minRes*) **then**
19 $C[3] = 0$, $C[2] = c[0]$, $C[1] = c[1]$, $C[0] = c[2]$,
20 *maxInliers* $=$ *Inliers*, *minRes* $=$ *res*
21 **If** *Inliers* $> n \cdot$*earlyStopRatio* **then break**
22 **If** *maxInliers*$/n <$ *goodLaneRatio* **then return** False
23 **Else** $T = V$
24 V = clear(V)
25 **For** $i = 1, 2, ..., n$ **do**
26 $y = T[i,0]^2 \cdot C[2] + T[i,0] \cdot C[1] + C[0]$
27 **If** $|y - T[i,1| \geq$ *inlierThresh* **then** $V = V \cup \{T[i]\}$
28 **Return** True

Fig. 6: RANSAC fitting algorithm in Apollo's lane detection.

illustrative examples are CRF (Conditional Random Field) and MRF (Markov Random Field).

As a concrete example, Figure 6 shows a variant of RANSAC fitting algorithm as it is implemented in Apollo's lane detection module. As it can be seen, the function, in line 7, generates three random values r_1, r_2, and r_3 which are then used to produce values x_1, x_2, and x_3 respectively in line 8. Based on these randomly generated values, the function initializes two matrices *A* and *B* in lines 10 and 11. These matrices are then used in $findSolution$ function, line 11, in which a mathematical equation is solved to obtain another vector c. Note that the randomly generated values have a cascade effect on the flow of the function and, in fact, we have observed that depending on the values of these matrices during the initialization phase, the main loop, lines 6 to 27, can iterate for a different number of times.

Impact of input data on timing and timing variability. In order to analyze the impact of input data on timing variability, we focus on a controlled scenario in which we can reason on the variability caused by input data (both random and deterministic) and the platform related variability. Note that Apollo's modules have more than 130,000 lines of code, and 6,200 functions with intricate dependences and high cyclomatic complexity [55]. Furthermore, these functions are event-triggered by events arriving from the sensors at different frequencies. This makes the analysis of Apollo inputs overwhelmingly complex.

In particular, we focus on the RANSAC algorithm introduced in Figure 6 as it combines four deterministic input parameters and three randomized inputs . The input parameters

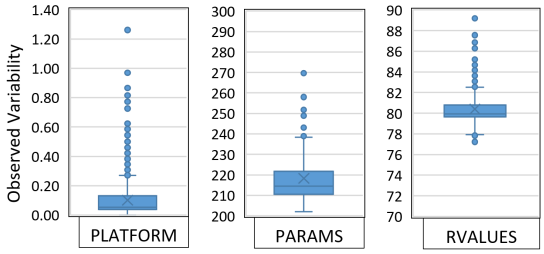


Fig. 7: Execution time variability of RANSAC

are a vector of matrices (`V`), an integer value showing the maximum number of iterations (`maxIters`), another integer value (`N`) describing the minimum number of data points required to estimate model parameters, and a floating-point value (`inlierThresh`) to determine data points that are fit well by the model. The 3 random inputs are r_1, r_2, and r_3. As part of Apollo, RANSAC is called 2,002 times each with a different set of values for the input parameters (`V`, `maxIters`, `N`, `inlierThresh`). In each call and iteration of RANSAC's main loop, random values are generated for r_1, r_2 and r_3. We have measured that the loop iterates at most 200 times. It is also worth noticing, that r_1, r_2, and r_3 have no dependence with the input parameters, i.e., they are generated randomly with the generation process and not influenced by the particular input parameters given to RANSAC. As an output, RANSAC produces the Matrix `C` and `true`/`false`.

From the execution of RANSAC as part of Apollo (RANSAC-native) we collect 2,002 sets of input parameters – one per invocation of RANSAC. We also collect several sets of random values corresponding to the values of r_1, r_2, and r_3 as part of several invocations to RANSAC. We use those values to feed a standalone version of RANSAC (RANSAC-standalone) under the following scenarios to capture the effect of input data: (DEF) same inputs data as RANSAC-native, both input parameters and random values; (FRAND) same input parameters as in RANSAC-native and fixed randomly generated values inside the function; (FPARS) same random values as in RANSAC-native, and fixed input parameters; and (FBOTH) that fixes both input parameters and random inputs.

We first compare RANSAC-native and RANSAC-standalone under the DEF scenario. Our results show that both produce the same outputs in terms of `C` and true/false. In terms of execution time, Figure 7 shows the variability in each of the scenarios. For FBOTH the left chart shows that the variability across runs under the same parameters and random inputs, i.e. due to the platform, is 5% on average (up to 1.26x). Under FRAND the variability caused by the input parameters (middle chart) is much higher ranging from 200x to 300x, with 214x on average. Finally, under FPARS the variability due to random values (right chart), is quite high ranging from 77x to 99x (79.9x on average) as well, though smaller than that due to input parameters. Overall, these results evidence the huge variability caused by random and deterministic input values, with reduced effect coming from the platform.

B. Approaching AD Software Timing

Static analysis approaches, while continuing to be an appropriate choice for the analysis of simpler, more predictable systems, can neither effectively model the increasingly complex hardware, nor deal with the structural and syntactical characteristics of exceptionally complex software functionalities [57, 3, 58, 52]. On the modeling side, building an accurate and reliable hardware model of modern heterogeneous platforms is rapidly becoming an untenable task, owing to their significant complexity and, often, by the non-disclosure of fundamental information [3, 52]. From an analytical perspective, instead, the typical program structure and code constructs found in complex AD functionalities pose a challenge, when not an impediment, to the various analysis steps in static timing analysis. In fact, the use of dynamic references (pointers), recursion, and unboundable loops, in combination with the intrinsic nature and (random) logic of typical AD advanced functionalities, often prevents the analysis from computing an absolute, realistic worst-case path [3, 33]. To overcome these limitations, static analysis approaches have typically indulged into conservative models and analysis assumptions that inevitably lead to overly pessimistic results.

Equally critical (and partially overlapping) issues also arise for industrially-established measurement-based timing analysis approaches [57], which cannot be straightforwardly applied to capture the entangled interactions between complex hardware and software functionalities. The behavior of AD software typically builds on deep, counter-intuitive, or even random input-output relations, that cannot be easily reconstructed. As a result, identifying (a priori) and triggering specific execution paths (typically among a huge number) or even fulfilling well-known code coverage requirements, such as Modified Condition/Decision Coverage (MCDC), becomes a cumbersome task [3, 57]. This scenario exacerbates the inherent shortcoming of conventional measurement-based approaches: the collected observations can only realistically be a small subset of the countless scenarios that can potentially happen due to the combination of software and hardware conditions, with the result of diminishing their predictive value [3]. Apollo modules exhibit extremely high cyclomatic complexity (number of linearly independent paths within a region of code), with several functions showing a cyclomatic complexity above 50, which is strongly discouraged [38], and ultimately does not allow to reach a satisfactory level of path coverage [7].

The orthogonal dimension of parallel execution also brings its own challenges to both static and measurement-based approaches. Bounding the timing interference potentially arising between, for example, Apollo modules due to contending accesses to shared resources is particularly challenging. The contention impact incurred by a module activation depends on the number and timing of requests sent by each module in the system to the shared hardware resources, which in turn depends on the particular traversed path as determined by the modules' input and sometimes potentially non-deterministic (random) algorithms. Static analysis, which normally handles multicore interference as an additive factor to be added to timing analysis results obtained in isolation [19, 15], generally fails to deliver sufficiently tight results. Dynamic approaches, instead, try to design specific tests to trigger the worst-case contention scenario [10], which is generally out of reach.

Representative Testing. In our view, the most feasible approach to follow for software timing budgeting and verification is that used for the verification of the software and hardware functional behavior in critical domains like avionics, where it is accepted that system complexity (hardware and software) makes it infeasible to scientifically prove the functional correctness of software or hardware and exhaustively test all possible conditions and scenarios [53]. On this account, full-path coverage is not required, as practically infeasible to achieve. Instead, a well-defined software-validation process, supported by the use of independent development and verification teams [53], is regarded as mandatory, with increasing rigor depending on the target DAL (Design Assurance Level)/ASIL (Automotive Safety Integrity Level).

The cornerstone of this approach [53] is *representative testing,* which applies to both functional and non-functional properties like software timing. In practice, the evaluated scenarios should account for sensitive algorithm characteristics – w.r.t. timing in our case – so that they have statistical relevance. How to achieve such representative testing is already addressed in the reference safety standard for AD systems, ISO21448 [30], which focuses on the *safety of the intended functionality* (SOTIF) and explicitly includes those functions that use machine learning algorithms, thus complementing the more general ISO26262. In particular, apart from sensor and actuator testing, SOTIF (section 10) states explicitly that *"relevant use cases and scenarios"* for the algorithm as well as those inputs that may trigger potentially hazardous behavior must be tested. Also, as part of the integration of the system, tests must include different environmental conditions (e.g., different visibility conditions). SOTIF also provides an annex describing the type of testing needed for perception systems, detailing that representative testing must include, not only usual driving conditions, but also *"conditions which are normally rare and less represented in normal driving but that might impact perception"*, *"uncommon scenarios that might increase the likelihood of a safety violation"* and additional tests *"based on system limitations"*.

Randomization impacts *dynamic scenario-oriented software* functional testing, the reference solution in the automotive domain [43, 30]. First, it complicates the definition of worst-case scenarios since the development and testing teams remain as the ultimate responsible for guaranteeing the coverage of relevant scenarios. And second, randomization also clouds the definition of what should be the correct result of a particular function. In fact, probabilistic indicators are generally accepted as a means to express a more fluid concept of correctness, better matching the outcomes of AD algorithms (e.g., object detection). In fact, outcomes are typically attached some degree of accuracy [46]. Interestingly, statistical and probabilistic concepts are not new to the analysis approach in automotive.

In fact, they are already accepted as part of automotive system analysis since, for instance, hardware failure rates and coverage are represented (and operated) with probabilities and percentages in the reference standard ISO26262 Part 5 [29]. Also, the recently issued SOTIF standard explicitly acknowledges the use of randomized test cases and random input data as a means to evaluate the residual risk for safety-critical systems in the automotive domain [30].

In the context of software timing, while not yet adopted by the automotive industry, a probabilistic treatment of the residual risk of software faults has already been shown to be compatible with ISO26262 [5]. Certification arguments to fit probabilistic reasoning in current standards have been already explored in the literature [54], showing that MBPTA can provide quantitative means to upper-bound the residual risk existing in any verification process of the timing of critical functions. In the specific context of AD systems, as discussed before, randomness is intrinsic to the delivered functionalities as they often build upon machine learning using random exploration techniques for efficiency purposes. This is, for instance, the case of Apollo. Therefore, any approach deployed for the timing analysis of this type of systems needs to account for some degree of randomness in the system timing behavior. Our view is that, in line with authors in [5], probabilistic reasoning can be considered an appropriate choice to model high execution times when their variability is, at least partly, caused by random events or choices.

V. On the Usage of Statistical Analysis to Master Apollo's Execution Time Variability

As for hardware failure rates and software functional behavior verification, statistical and probabilistic approaches [14, 20] are promising solutions also for the characterization of the timing behavior of this type of complex AD systems. Non-determinism caused by randomness makes such systems to exhibit a highly variable timing behavior where the worst-case execution scenarios are more accurately modeled as a distribution rather than an absolute value.

Statistical inference methods generally build on (simplifying) assumptions on the target population they are meant to model. A common assumption is that inference is made from a random sample of the population. However, guaranteeing the availability of a random sample of the relevant data is not always feasible [28]. This is the case, for example, of statistical inference on the timing behavior of embedded software systems, as the relevant population (execution times during operation) is simply unknown and unavailable at analysis time, where the inference reasoning is applied. Therefore, the inference data cannot be guaranteed to be a random sample of the target population.

For these reasons, a fundamental prerequisite in statistical timing analysis techniques is that the execution time distribution at analysis time matches or upper-bounds that during operation. This property is often referred to as *representativeness* [40, 25, 26] and is typically guaranteed by both, enforcing worst-case scenarios at analysis time and by exercising all the relevant program inputs. As discussed before, this latter aspect is clearly affected by the introduction of randomization in the functional behavior of the software under analysis. In the case of Apollo, the inference data is predetermined by the input data set provided by the Apollo consortium itself.

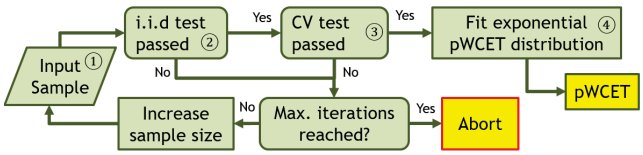


Fig. 8: Main steps in MBPTA-CV application.

A. Statistical and probabilistic approaches for timing analysis

We explore the use of EVT [18] for timing prediction as a representative of state-of-the-art statistical analyses. EVT has been shown suitable to model extreme events of processes in different domains (e.g., hydrology, stock market) *making no assumption on the distribution* of the random variable or its tail. The latter aspect fits particularly well the characteristics of Apollo (see Section III-A). There are, however, some inherent constraints to the applicability of EVT for timing analysis. EVT requires the input sample of observations of the process under study to be independent and identically distributed (i.i.d.) so that they can be directly processed by EVT. This is assessed statistically with appropriate tests for the input sample as a means to verify that the statistical properties of the sample match those of the process being modelled. Also, processes with dependencies across measurements may also be processed taking some cautionary measures (we refer the interested reader to other works for further details [14]). EVT fits the parameters of the distribution, either a Generalized Extreme Value (GEV) distribution or a Generalized Pareto Distribution (GPD) to the maxima (or minima) of the data sample to model the corresponding tail of the distribution. In EVT, the shape parameter is particularly relevant since it determines the rate at which the tail falls. Heavy tails ($\xi > 0$) are appropriate for unbounded distributions, whereas light tails ($\xi < 0$) are appropriate for bounded distributions. The limit distribution ($\xi = 0$) is an exponential tail.

Moreover, a sensible application of EVT requires the capability of relating measurements captured from the system at analysis time to the execution times that will occur during operation, which is typically referred to as *representativeness problem* [14, 40, 25, 26]. Guaranteeing representativeness relates to the coverage problem in measurement-based approaches. Some authors propose the use of specific hardware and/or software support to ease this task (i.e. by forcing execution time jitter to exhibit a purely random nature) [34]. The representativeness argument is a usual limitation of all measurement-based approaches, including probabilistic ones. In the case of AD frameworks in general and Apollo in particular, representative test cases are needed to train the neural networks used by the framework. We build the rep-

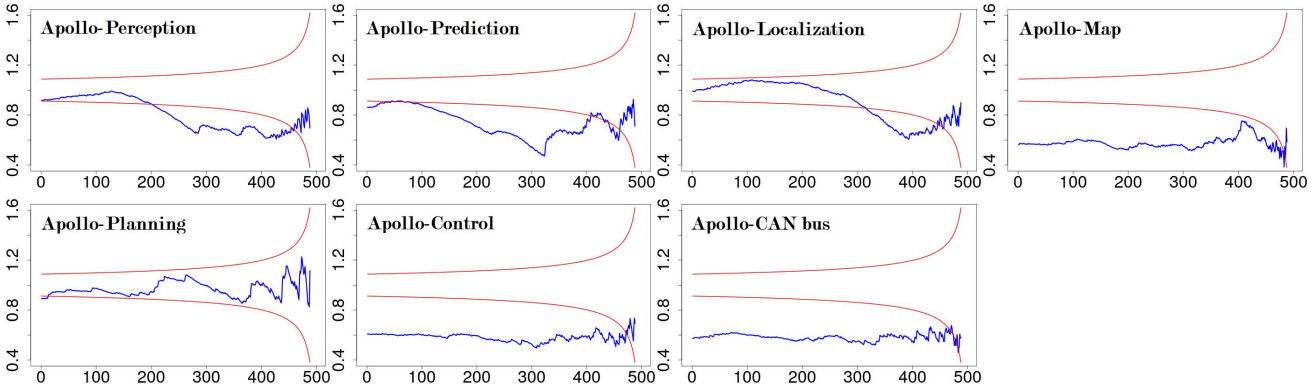


Fig. 9: CV-plots for the different Apollo modules (CV estimator value in the y-axis and excluded maxima in the x-axis).

resentativeness argument on those test cases used to train the neural networks, which we regard as representative in terms of both path coverage and path frequency, so that measurements from all paths can be used in a single sample for MBPTA [40].

B. MBPTA-CV

In this work, we use a specific MBPTA technique referred to as MBPTA-CV [4] (its code is available in [31]) that builds on EVT to upper-bound execution time probabilities for probability ranges beyond what has been observed. MBPTA-CV delivers an exceedance probability function that can be instantiated to derive pairs <execution time, exceedance probability>, so that we can obtain an exceedance probability that upper-bounds the exceedance probability for that execution time – which indeed can be zero. MBPTA-CV does not make any specific assumption on the input data provided: it is up to the user providing representative input data so that WCET estimates can be related to the target system behavior.

MBPTA-CV operates following the steps shown in Figure 8:

1) *Sample collection:* MBPTA-CV fits an exponential distribution to predict high execution times using at least 50 values. Hence, MBPTA-CV needs a sample of at least 100 execution time measurements since only half (the highest) values are considered for probabilistic Worst-Case Execution Time (pWCET) estimation, regarding the lowest values as not suitable to predict high execution times by definition.
2) *i.i.d. properties:* MBPTA-CV builds on an input sample where execution time observations are i.i.d., or at least the set of (high) values used for prediction is i.i.d. These properties are assessed with appropriate statistical tests [23][4]. When any of the tests is failed, the input sample size is increased by 50 measurements until the test is passed, which necessarily occurs with at most few iterations for i.i.d. distributions. If the sampled distribution does not meet i.i.d. properties, this step will not converge and, after a number of iterations, the process must be aborted. This would mean that MBPTA-CV cannot be applied to model the observed distribution.
3) *Exponential properties:* MBPTA-CV uses exponential tail distributions for pWCET estimation, as they are necessarily reliable upper-bounds to the tail distribution. In particular, it has been shown that processes with a finite maximum can be modelled with light tails (i.e. approaching asymptotically a maximum value), and reliably upper-bounded with exponential distributions, which are their limit distribution as explained before [4]. While this holds by construction for real-time programs – thus with finite execution time – it still needs to be conveniently tested on the input sample, since random samples may not match the properties of the distribution sampled with some probability. MBPTA-CV uses the CV estimator (described in detail later in this section) to assess exponentiality. If the latter property is not ascertained, the sample size needs to be increased (e.g., by 50 measurements) until the test is passed, which eventually occurs for distributions with a (finite) maximum, since, naturally, an increasingly larger sample approaches the sampled distribution. In practice, we use samples of at least 1,000 measurements instead of 100 since the larger the sample, the lower the chances of having false positives and false negatives for i.i.d. tests. Moreover the chances to pass the CV test increase for suitable distributions (those with a finite maximum). Note that considering larger input samples, as a means to cover high-impact and low-probability random events, is not necessary since the effects of each random value used in Apollo and YOLO in terms of variability are very limited in scope, typically restricted to a function call, and several independent random values are used in calls to the same function on each execution time observation [39].
4) *pWCET estimation:* Once we have a sufficiently large i.i.d. sample allowing exponential tail fitting, MBPTA-CV selects the set of tail values delivering the tightest fit, while preserving reliability. Such tightest fit is, in general, expected for the cases where the CV estimator is closer to the theoretical CV value. We refer the interested reader to the MBPTA-CV method [4] for details on the process to select the set of maxima used for tail fitting.

The obtained pWCET distribution bounds the actual execution time distribution during operation and appropriate residual risk upper-bounds can be used to obtain reliable execution time bounds [5]. Indeed, MBPTA-CV has been already shown to upper-bound tails reliably by comparing the obtained pWCET estimates for small samples (10^3 measurements) against very large samples (10^7 measurements), as long as the above steps are successfully applied [4].

C. Inputs

We executed Apollo using the datasets composed of real sensor data including camera, LiDAR, GPS and other data [1, 2]. The total size of these datasets, known as *bag* files, is in the order of Gigabytes of data per minute of driving, and they represent a massive amount of data collected from sensors of an equipped car driven in different traffic environments such as highway and urban roads. The Apollo team provides several of these *bag* files for different configurations and scenarios [1], which are used for the experiments in this paper.

Apollo's development and testing experts claim that the scenarios from which data is provided include a variety of road types, obstacle types and road environments, intended to provide as much coverage of relevant cases as possible [1]. In particular, driving scenarios include urban roads and high-speed scenes, with a variety of obstacles such as motor vehicles, non-motor vehicles, pedestrians and static obstacles, to name a few. In any case, representative testing is a necessary prerequisite for a reliable timing analysis of an AD framework and, as discussed before, this needs to be guaranteed by following specific design and testing processes [53].

D. Module-Level Analysis

Independence and Identical Distribution Tests. For identical distribution, we apply the Kolmogorov-Smirnov two-sample identical distribution test [23] and for independence the Ljung-Box test [12], as described in [4]. While dependencies may exist in the data, and so in the execution times, it has been shown that, in the context of EVT, dependencies are only relevant if they occur in the maxima [37]. Hence, we have applied the independence test on the set of maxima (high values) retained for pWCET tail fitting. As shown in Table III, the resulting p-values for both tests are above the significance level ($\alpha = 0.05$), and therefore, i.i.d. hypotheses cannot be rejected. This emanates from the fact that, while input data are not independent for neighboring frames, relatively distant frames are highly independent.

TABLE III: Results of the i.i.d tests (Apollo modules).

	Per	Pred	Loc	Map	Plan	Con	CAN
(i.)	0.172	0.523	0.246	0.067	0.296	0.515	0.679
(i.d)	0.985	0.996	0.989	0.597	0.957	0.206	0.098

Hence, we conclude that the intrinsic variability produced by the inputs and measurement collection of the execution times for the Apollo modules, given that execution times of the modules are necessarily finite, suffices to enable the use of MBPTA.

Exponential properties. In order to test the exponential properties of the execution time distributions, MBPTA-CV builds upon the CV estimator, as explained before. Such estimator is below 1 if the tail can be upper-bounded with exponential tails, exactly 1 if it is exponential, and above 1 if exponential tails cannot upper-bound it. This estimator is often shown graphically with the CV-plot [4, 21]. The CV-plots for the Apollo modules are shown in Figure 9. Note that, the CV corresponds to the ratio between the (theoretical) standard deviation and mean of the distribution, whereas the CV estimator is the ratio between the standard deviation and mean of the sample. The CV-plot shows the residual CV estimator (in the y-axis), so as we move from left to right, an increasing number of values of the sample is excluded (the lowest ones), and the residual CV for a set of values is obtained for the excesses[1] in the remaining set of values. For instance, in our case, the sample size is 1,000 measurements. The highest 500 values are used for the CV-plot. The x-axis value 150 indicates that the lowest 150 values out of the 500 used are excluded, thus keeping only the highest 350 values. The confidence interval in the CV-plot (within red lines) corresponds to the 95% confidence interval for $CV = 1$ for a Normal distribution of the residual CV estimator (so p-values below 0.05 would not be compatible with the exponential assumption), being such confidence interval wider as we use smaller samples (i.e. more values are excluded so fewer remain in the sample). Hence, whenever the blue line (CV estimator) is within the red lines, exponentiality cannot be rejected. If it is below both red lines, exponentiality can be rejected but exponential distributions are still a tail upper-bound. Finally, if the CV estimator is above both red lines, exponential tails cannot be used for that particular number of excesses. As expected, execution time distributions for all modules can be upper-bounded with exponential tails for any number of excesses since they are never above the confidence interval.

For completeness, we also show the Mean Excess (ME) plot for those execution time distributions (see Figure 10), which is a more popular way of assessing exponentiality. However, the ME-plot has only been used qualitatively so far and hence, is not suitable for an automatic tool as opposed to the CV estimator. The ME-plot shows, from left to right, the mean of the excesses over a threshold. If a tail converges towards a maximum value close to those in the sample, the spread of the excesses decreases as we increase the threshold and hence, the ME value decreases. As it can be seen, this is the case for all modules but Planning, for which the ME value decreases as we increase the threshold. These tails, while not exponential, have a sharper fall rate than exponential tails and hence, can be upper-bounded with exponential tails. If the ME value remains approximately constant or with no clear trend, an exponential tail distribution is appropriate, as it is the case for Planning (note that Planning has also the highest sustained CV values across all modules). Only when the ME value grows as the

[1]The excesses are the values above a threshold subtracting the threshold itself.

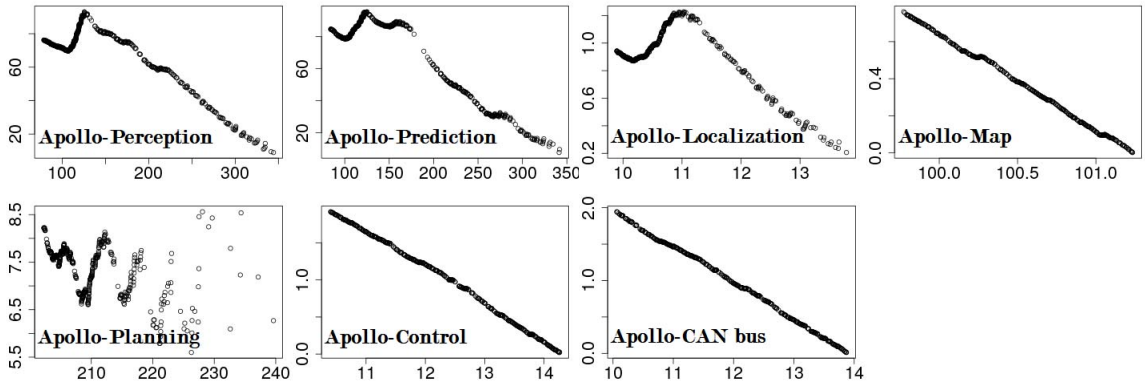


Fig. 10: Mean excess plots. The threshold in the x-axis and the mean of the excesses in the y-axis, in seconds.

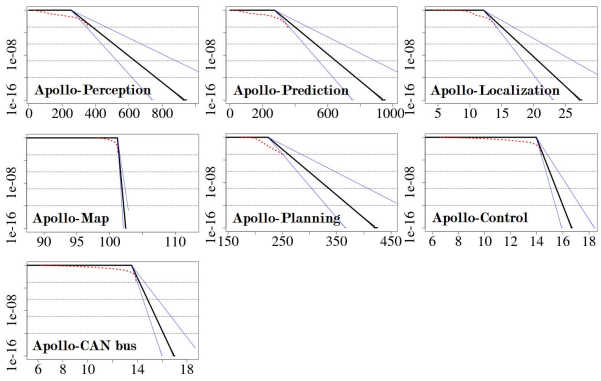


Fig. 11: pWCET estimates (seconds) for the different Apollo modules (exceedance probability in the y-axis).

threshold increases, exponential tails cannot be used. This does not occur for any of the Apollo modules.

pWCET estimates. In Figure 11, we show the pWCET estimates obtained from the collected execution times. We show point estimation (solid thick black line) and interval estimation (solid thin blue lines) for a confidence interval of 95%. pWCET distributions are shown in the form of complementary CDF plots, thus meaning that the lines indicate the execution time (x-axis) that would be exceeded with a probability upper-bounded by the corresponding value in the y-axis. For instance, for the Perception module, the probability of exceeding 1,000 seconds is up to 10^{-10} per run. In all cases, a pWCET estimate can be derived, and it always upper-bounds the observed values (dotted line). The decay rate of the pWCET curves shows the increase in execution time due to reducing the residual risk bound, that relates to the ASIL [5]. We categorize functionalities based on whether their pWCET distribution has high or low decay rate in the range of (relevant) exceedance probabilities per run: 10^{-9} to 10^{-15}.

- Map, Control and CAN Bus have high decay.
- Instead, Perception, Prediction, Localization, and Planning have low decay rate. This category results in relatively high pWCET estimates for decreasing exceedance probabilities, which opens the door to the use of tighter prediction models to reduce allocated CPU capacity.

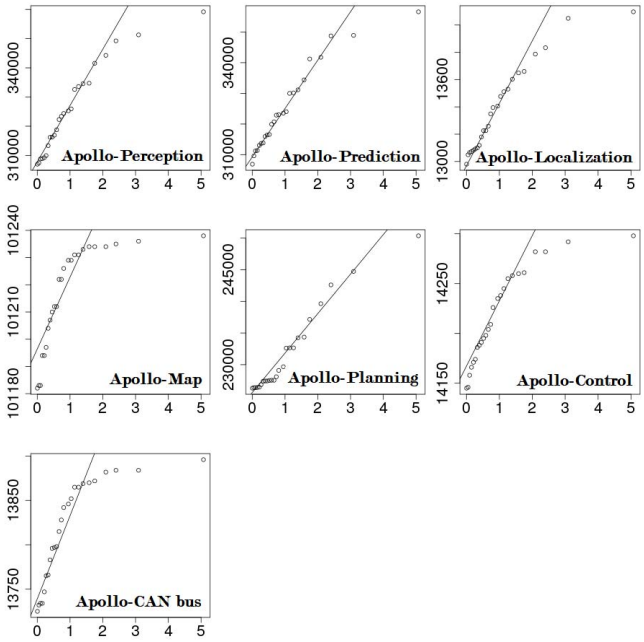


Fig. 12: QQ plot for the tail values used for pWCET estimation for the different Apollo modules (empirical quantiles in the y-axis and theoretical exponential quantiles in the x-axis).

Note that decay rates strongly correlate with $Kurt$ and maximum values, which indicate that, whenever $Kurt$ is high, and the maximum is relatively far away from the Q1-Q3 range, the slope of the tail can only be gentle.

For completeness, we assess the exponentiality of the tail values selected by MBPTA-CV for tail estimation. In particular, we compute the exponential QQ-plot for those values (see Figure 12). We observe that values concentrate in the vicinity of the line, thus reflecting exponentiality, except for the

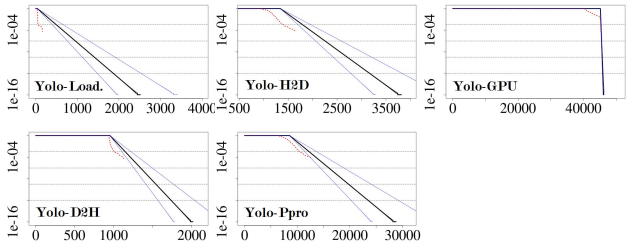


Fig. 13: pWCET estimates for YOLO stages.

highest values, which fall below the line, thus indicating that exponential distributions are actual upper-bounds. Therefore, this evidence further corroborates that the use of MBPTA-CV on Apollo modules is able to select appropriate tail values to fit reliable pWCET distributions.

E. Stage Level Predictions

We applied the same methodology to the different stages in the Perception module as implemented in YOLO.

Independence and Identical Distribution Tests. Given that modules present i.i.d. execution times across runs, their stages are also naturally i.i.d. Results in Table IV confirm this expectation with the resulting p-values well above the significance level $\alpha = 0.05$.

TABLE IV: Results of the i.i.d tests (YOLO Stages).

	Load	H2D	GPU	D2H	PPro
(i.)	0.067	0.106	0.974	0.246	0.297
(i.d)	0.988	0.974	1.000	0.863	0.991

Exponential properties. Exponential tests with the CV estimator show analogous results to those observed for the full modules, so we omit them due to space constraints, as they do not provide further insights. However, we note that the same positive conclusions for the use of EVT as part of MBPTA-CV reached for full modules, hold also for individual stages within modules.

pWCET estimates. As shown in Figure 13, pWCET estimates can also be obtained for the stages of a module. As for the module level analysis, slopes highly correlate with $Kurt$ values since most stages have both, high $Kurt$ values and gentle slopes for their pWCET estimates.

F. Summary

From the results in this section we conclude that statistical analysis offers a feasible path to address the variability observed in Apollo's modules and stages execution time variability. In terms of the approach, MBPTA-CV is intended to model extreme (high) timing behavior for execution times exhibiting some form of variation with either a random or non-obvious nature. This fits perfectly the case of test cases for AD software which, in line with SOTIF standard, build upon some form of random inputs or test cases. In terms of the procedure, our analysis shows that execution times fit the statistical properties needed for the application of EVT for pWCET estimation, namely independence, identical distribution and compatibility with the exponential assumption; and the application of MBPTA-CV delves pWCET distributions where tails are properly selected so that exponential distributions are a suitable fit.

VI. Related Work

The timing validation of automotive systems has been customarily based on the combination of dynamic measurements with a system-level timing model [8, 43], often extended to capture CAN or network-based communication between ECUs [42]. Some works have also reported on industrial experience in applying static timing analysis to automotive software [27, 32]. These works, however, focus on the timing characterization of traditional, arguably simple, automotive software, on relatively predictable hardware platforms. As such, they are unfit to capture and understand the execution time variability arising when shifting to complex AD software running on multicores and GPUs. The work in [59] advocates stochastic analysis for the characterization of end-to-end latencies, thus focusing on system-level aspects rather than timing characterization. Meanwhile, for the localization module, authors in [36] report large execution time variability, but do not analyze it as we have done in this paper for Apollo modules and stages.

In [44], the authors evaluate the use of NVIDIA's TX1 in real-time computer vision-based workloads. They use a combination of synthetic benchmarks, image processing samples from NVIDIA's CUDA SDK, and a closed-source road-sign recognition industrial case study. Our work differs both in size and complexity of the evaluated software, as we characterize the timing variability and analyze the timing behavior of an entire AD framework, far beyond the sole Perception module.

VII. Conclusion

The slant towards autonomous driving solutions is pushing for the adoption of advanced software functionalities exploiting complex AI-based algorithms. The inherent non-deterministic traits of AD software challenge both, effectiveness and scalability of conventional analysis approaches. In this work, we present an analysis of the timing variability of Apollo, an industrial-quality AD framework showing that Apollo modules and stages exhibit a highly variable timing behavior and arbitrary distributions. We analyze randomization as one of the reasons behind this variability, and show how it impairs some of the fundamentals of consolidated timing analysis techniques. We also discuss approaches and prospects to address this variability. In line with the latter, we show that statistical approaches are better equipped to effectively model the timing behavior of AD frameworks similar to Apollo. We illustrate this by analyzing the execution time of Apollo modules and single stages of the Perception module when running on a real board.

ACKNOWLEDGMENT

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the SuPerCom European Research Council (ERC) project under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773), and the HiPEAC Network of Excellence. MINECO partially supported Enrico Mezzetti under Juan de la Cierva-Incorporación postdoctoral fellowship (IJCI-2016-27396), and Leonidas Kosmidis under Juan de la Cierva-Formación postdoctoral fellowship (FJCI-2017-34095).

REFERENCES

[1] Apollo, an open autonomous driving platform. http://apollo.auto/, 2019.

[2] Apollo, an open autonomous driving platform, source-code and manuals. https://github.com/ApolloAuto/apollo, 2019.

[3] Jaume Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems, SIES*, pages 39–48. IEEE, 2015.

[4] Jaume Abella, Maria Padilla, Joan del Castillo, and Francisco J. Cazorla. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Design Autom. Electr. Syst.*, 22(4):72:1–72:29, 2017.

[5] Irune Agirre et al. Fitting software execution-time exceedance into a residual random fault in ISO-26262. *IEEE Trans. Reliability*, 67(3):1314–1327, 2018.

[6] Sergi Alcaide et al. Safety-related challenges and opportunities for GPUs in the automotive domain. *IEEE Micro*, 38(6):46–55, 2018.

[7] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. Software engineering challenges of deep learning. In *44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA*, pages 50–59. IEEE Computer Society, 2018.

[8] AUTOSAR. Recommended methods and practices for timing analysis and design within the autosar development process. Technical Report (n.645), 2017.

[9] Kostiantyn Berezovskyi et al. Measurement-based probabilistic timing analysis for graphics processor units. In *Architecture of Computing Systems - ARCS International Conference, Proceedings*, volume 9637 of *Lecture Notes in Computer Science*, pages 223–236. Springer, 2016.

[10] Jingyi Bin et al. Studying co-running avionic real-time applications on multi-core COTS architectures. In *Embedded Real Time Software and Systems, ERTS 2014*, February 2014.

[11] Mariusz Bojarski et al. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[12] George E. P. Box and David A. Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, 65(332):1509–1526, 1970.

[13] Alejandro J. Calderón et al. Understanding and exploiting the internals of GPU resource allocation for critical systems. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD*, pages 1–8. ACM, 2019.

[14] Francisco J. Cazorla et al. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1):14:1–14:35, 2019.

[15] Sudipta Chattopadhyay et al. A unified WCET analysis framework for multicore platforms. *ACM Trans. Embedded Comput. Syst.*, 13(4s):124:1–124:29, 2014.

[16] Chenyi Chen, Ari Seff, Alain L. Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *IEEE International Conference on Computer Vision, ICCV*, pages 2722–2730. IEEE Computer Society, 2015.

[17] Xiaozhi Chen et al. Multi-view 3d object detection network for autonomous driving. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 6526–6534. IEEE Computer Society, 2017.

[18] Stuart Coles. *An introduction to statistical modeling of extreme values*. Springer Series in Statistics. Springer-Verlag, London, 2001.

[19] Dakshina Dasari et al. Identifying the sources of unpredictability in cots-based multicore systems. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES*, pages 39–48. IEEE, 2013.

[20] Robert I. Davis and Liliana Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *LITES*, 6(1):03:1–03:60, 2019.

[21] Joan del Castillo and Isabel Serra. Likelihood inference for generalized pareto distribution. *Comput. Stat. Data Anal.*, 83:116–128, 2015.

[22] Mohamed Elbanhawi and Milan Simic. Sampling-based robot motion planning: A review. *IEEE Access*, 2:56–77, 2014.

[23] William Feller. *An introduction to Probability Theory and Its Applications*. 1966.

[24] Ford. Media Center Release. https://media.ford.com/content/fordmedia/fna/us/en/news/2017/02/10/ford-invests-in-argo-ai-new-artificial-intelligence-company.html, 2017.

[25] Samuel Jimenez Gil et al. Open challenges for probabilistic measurement-based worst-case execution time. *Embedded Systems Letters*, 9(3):69–72, 2017.

[26] Fabrice Guet, Luca Santinelli, and Jérôme Morio. On the representativity of execution time measurements: Studying dependence and multi-mode tasks. In *17th International Workshop on Worst-Case Execution Time Analysis, WCET*, pages 3:1–3:13, 2017.

[27] Jan Gustafsson and Andreas Ermedahl. Experiences from applying WCET analysis in industrial settings. In *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 382–392, 2007.

[28] Gerald J. Hahn and William Q. Meeker. Assumptions for statistical inference. *The American Statistician*, 47(1):1–

11, 1993.

[29] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[30] International Organization for Standardization. *ISO/PAS 21448. Road Vehicles – Safety of the Intended Functionality*, 2019.

[31] Jaume Abella. MBPTA-CV brief user guide, v1.0. https://zenodo.org/record/1065776#.XbBB8-gzaVk, 2017.

[32] Daniel Kästner et al. Timing validation of automotive software. In *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA. Proceedings*, pages 93–107, 2008.

[33] Raimund Kirner and Peter P. Puschner. Obstacles in worst-case execution time analysis. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 333–339. IEEE Computer Society, 2008.

[34] Leonidas Kosmidis et al. Fitting processor architectures for measurement-based probabilistic timing analysis. *Microprocess. Microsystems*, 47:287–302, 2016.

[35] George Lima, Dario Dias, and Edna Barros. Extreme value theory for estimating task execution time bounds: A careful look. In *28th Euromicro Conference on Real-Time Systems, ECRTS*, pages 200–211, 2016.

[36] Shih-Chieh Lin et al. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 751–766, 2018.

[37] Yue Lu, Thomas Nolte, Iain Bate, and Liliana Cucu-Grosjean. A new way about using statistical analysis of worst-case execution times. *SIGBED Review*, 8(3):11–14, 2011.

[38] Thomas J. McCabe. Cyclomatic complexity and the year 2000. *IEEE Software*, 13(3):115–117, 1996.

[39] Suzana Milutinovic, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Increasing the reliability of software timing analysis for cache-based processors. *IEEE Trans. Computers*, 68(6):836–851, 2019.

[40] Suzana Milutinovic, Enrico Mezzetti, Jaume Abella, Tullio Vardanega, and Francisco J. Cazorla. On uses of extreme value theory fit for industrial-quality WCET analysis. In *12th IEEE International Symposium on Industrial Embedded Systems, SIES*, pages 1–6, 2017.

[41] Detlev Mohr et al. The road to 2020 and beyond: What's driving the global automotive industry, 2013.

[42] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study. *Comput. Sci. Inf. Syst.*, 10(1):453–482, 2013.

[43] Nicholas Navet. Timing analysis of automotive architectures and software. 53rd Design Automation Conference (DAC), 2016. Invited Talk.

[44] Nathan Otterness et al. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. pages 353–364, 2017.

[45] Brian Paden, Michal Cáp, Sze Zheng Yong, Dmitry S. Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Trans. Intelligent Vehicles*, 1(1):33–55, 2016.

[46] Scott Pendleton et al. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5(1):6, Feb 2017.

[47] Scott Drew Pendleton et al. Perception, planning, control, and coordination for autonomous vehicles. *MDPI Machines*, 5, 2017.

[48] Cuong Cao Pham and Jae Wook Jeon. Robust object proposals re-ranking for object detection in autonomous driving using convolutional neural networks. *Signal Process. Image Commun.*, 53:110–122, 2017.

[49] Roger Pujol et al. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the NVIDIA xavier. In *31st Euromicro Conference on Real-Time Systems, ECRTS*, volume 133 of *LIPIcs*, pages 23:1–23:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[50] Morgan Quigley et al. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[51] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. pages 6517–6525, 2017.

[52] Jan Reineke. Challenges for timing analysis of multi-core architectures. Workshop on Foundational and Practical Aspects of Resource Analysis, 2017. Invited Talk.

[53] Leanna Rierson. Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance. 2017.

[54] Zoë R. Stephenson, Jaume Abella, and Tullio Vardanega. Supporting industrial use of probabilistic timing analysis with explicit argumentation. In *11th IEEE International Conference on Industrial Informatics, INDIN*, pages 734–740. IEEE, 2013.

[55] Hamid Tabani et al. Assessing the adherence of an industrial autonomous driving framework to ISO 26262 software guidelines. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC*, page 9, 2019.

[56] Toyota Motor Corporation. Toyota News Release. https://corporatenews.pressroom.toyota.com/releases/toyota+establish+artificial+intelligence+research+development+company.htm, 2015.

[57] Reinhard Wilhelm et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008.

[58] Reinhard Wilhelm and Jan Reineke. Embedded systems: Many cores - many problems. In *7th IEEE International Symposium on Industrial Embedded Systems, SIES*, pages 176–180. IEEE, 2012.

[59] Haibo Zeng, Marco Di Natale, Paolo Giusto, and Alberto Sangiovanni-Vincentelli. Stochastic analysis of can-based real-time automotive systems. *IEEE Transactions on Industrial Informatics*, 5(4):388–401, Nov 2009.