Defeating Barriers for Resource Usage Testing for Autonomous Driving Frameworks

Author:
Miguel Alcón Doganoc

Supervisor:
Hamid Tabani — Barcelona Supercomputing Center

Co-supervisor:
Jaume Abella — Barcelona Supercomputing Center

Tutor:
Miquel Moretó — Department of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona Supercomputing Center

Advanced Computing Specialization
Master in Innovation and Research in Informatics

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Department of Computer Architecture - Operating Systems
Barcelona Supercomputing Center

June 24, 2020

Acknowledgements

First of all, I would like to sincerely thank my master Thesis' advisors Hamid, Jaume and Fran, for their guidance and mentoring throughout this trip. I also want to thank the rest of the members of the CAOS group at the BSC, as they have always been available to lend me a hand and provide feedback on my work to improve it, especially to Leonidas and Enrico for collaborating in this research.

Moreover, I would like to acknowledge the BSC institution for financially support my master studies, and also to the following institutions that have partially supported this work: the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the UP2DATE European Union's Horizon 2020 (H2020) research and innovation programme under grant agreement No 871465, 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.

Finally, I thank my parents and sister for their unconditional support during all these years. Without you, this master Thesis would not be possible.

Abstract

The software used to implement advanced functionalities in critical domains (e.g. autonomous operation) impairs providing evidence that the software has enough resources to correctly execute (e.g. time and memory). 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 Thesis, we focus on Apollo, an industrial-quality Autonomous Driving (AD) software framework. AD systems, similar to other automotive safety-critical systems, must undergo a development process with exhaustive Verification and Validation (V&V) steps. Both steps are challenged by the inherent complexity of AD systems. Our work can be divided into two contributions.

First, we statistically characterise Apollo's observed execution time variability and reason on the sources behind it, aiming the verification step. We discuss the main challenges and limitations in finding a satisfactory software timing analysis solution for Apollo. While providing a consolidated solution for the software timing analysis of Apollo is a huge effort far beyond the scope of a single master Thesis, our work aims to set the basis for future and more elaborated techniques for the timing analysis of AD software.

Second, we enable software resource usage testing, including execution time bounds and memory, on Apollo. Resource usage testing is a mandatory validation step during the integration of safety-related real-time systems. This Thesis exposes the difficulties to perform resource usage testing for AD frameworks by analysing a complex and critical module of Apollo. Then, it provides some guidelines and practical evidence on how resource usage testing can be effectively performed, thus enabling end-users to validate their safety-related real-time AD frameworks.

Contents

Acronyms

AD Autonomous Driving.

AI Artificial Intelligence.

ASIL Automotive Safety Integrity Level.

AV Autonomous Vehicle.

CAN Controller Area Network.

CDF Cumulative Distribution Function.

D2H Device-to-Host.

DAG Direct Acyclic Graph.

DAL Design Assurance Level.

DLA Deep Learning Accelerator.

GDB GNU Project Debugger.

GPT Google Performance Tools.

H2D Host-to-Device.

HMI Human Machine Interface.

ISP In-System Programming.

LiDAR Light Detection And Ranging.

MBTA Measurement-Based Timing Analysis.

PVA Programmable Vision Accelerator.

QM	Quality Managed.
RANSAC	RANdom SAmple Consensus.
ROS	Robot Operating System.
SOTIF	Safety Of The Itended Functionality.
STA	Static Timing Analysis.
V&V	Verification and Validation.
WCET	Worst-Case Execution Time.
YOLO	You Only Look Once.

List of Figures

List of Tables

Chapter 1

Introduction

Each year, more than 1.35 million people die as a result of road traffic crashes [53], and millions of people are injured. Studies show that transportation is responsible for nearly 30% of the Europe's total CO_2 emissions, of which 72% comes from road transportation [55]. This does not include other drawbacks of the automotive sector, such as effect on soil, water and noise pollution, energy dependency or causing traffic congestion [44]. Autonomous vehicles (AV) can become part of the equation to solve some of these problems such as reducing fatalities and CO_2 emissions.

As all autonomous systems [71], AVs are capable of making decisions independently of human interference, and these decisions have to be taken while facing uncertainty. More precisely, AVs rely on AI, sensors and big data to analyse information, adapt to changing circumstances and handle complex situations on the road as a substitute for human judgement. Adding this to that at least 90% of vehicle accidents are estimated to be the result of human error, AVs could reduce or eliminate the largest cause of car accidents while also outperforming human drivers in perception, decision-making and execution. However, since autonomous systems are very complex, machine errors could lead to other problems.

1.1 Motivation

Now, AVs are starting to be a reality. Indeed, AD is attracting significant interest from the industry, primarily from the automotive [23, 29, 47, 73] and technology [7, 52, 77] sectors. Top companies like Ford, Toyota, Daimler, Google, Amazon, and NVIDIA are currently working on AD technologies. Even some of them already have AVs on the road. This is also the case for Baidu and its Apollo project. 52 AVs of Baidu droved 754,000 km on Beijing roads in 2019 [67]. Most recent news points out that during the COVID-19 pandemic, Baidu's driverless cars are helping to carry out frontline anti-epidemic work such as cleaning, disinfecting, logistics, and transportation with support from partner companies [15]. In particular, these Baidu's cars are driving under the control of the Apollo

AD framework [14], which is one of the most sophisticated open-source projects implementing the entire AD software stack. In this project, more than 120 OEMs, Tier1, Tier2, high-tech companies, and car manufacturers are participating. In this Thesis, we use Apollo as our reference AD system for our experiments and of our analysis.

AD systems differ from regular systems in that they are considered real-time and safety-critical [31, 72]. On the one hand, real-time systems are systems where their behaviour correctness depend not only on the logical results of the computations but also on the physical time they take to produce these results. They must guarantee the results are produced within specified time constraints. For instance, the braking system of a car must activate the brake before a deadline to guarantee its correct behaviour and the safety of the passengers. On the other hand, safety-critical systems are systems whose failure or malfunctioning may result in death or injury to people, loss or severe damage to equipment or property, or environmental harm. These systems must be protected to ensure safety is not compromised, and failures occur with negligible probability. Following the previous example, the braking system is also safety-critical. To sum up, safety-critical real-time systems must guarantee functional (e.g. correct output of an algorithm) and non-functional (e.g. meeting deadlines) requirements. In this Thesis, we focus on time correctness of AD systems in general, and of Apollo in particular.

1.2 Problem Statement, Objectives and Contribution

AD systems, similar to other automotive safety-related systems, must undergo a development process with exhaustive V&V steps, where each item is proven to adhere to its safety requirements with the degree of rigor dictated by safety standards such as ISO 26262 [33]. ISO 26262 is an international standard for functional safety of electrical and/or electronic systems in production automobiles defined by the International Organization for Standardization.

The development process of safety-related automotive systems is well-defined in ISO 26262. In particular, some safety goals are determined related to safety at the vehicle level. Those safety goals are mapped into specific safety requirements as the system is specified, so that items composing the system have some safety requirements to meet. Then, the architectural design of the system is performed propagating requirements. At each stage, some verification activities are performed, for instance, related to assessing whether all requirements are met. Then, software and hardware elements are designed and implemented, verifying that they meet their requirements. Once components are implemented, they are tested to detect flaws in their design. This requires defining appropriate test cases and metrics to assess test coverage. As components are integrated, larger tests are performed to validate the integration, the adherence to the requirements, and finally the correct operation of the vehicle. This test phase is referred to as the validation phase.

In safety-critical systems, on one hand, it should be guaranteed that each task has enough resources before its execution begins. On the other hand, timing verification for software items has received significant attention during decades with a plethora of techniques aimed at deriving estimates to the Worst-Case Execution Time (WCET) of tasks to verify that specific task schedules meet safety requirements (e.g. the braking system activates the brake before its deadline) [2, 20, 38, 40, 41, 50, 65, 81]. Instead, timing validation has received much less attention. Timing validation focuses on showing that derived timing budgets are not violated. The absence of violations serves as evidence for certification purposes on the timing correctness of the system. Therefore, resource verification is key for the successful execution of the tasks.

Automotive industry resorts to engineering practices based on creating stressing tests and collecting measurements, sometimes with the help of appropriate timing analysis tools that can be used for both timing V&V [68]. These techniques rely on the ability to collect information on the execution of the tasks under analysis. This is precisely challenged by forthcoming AD systems. This is so because AD systems build upon overwhelmingly complex software constructs. On the one hand, paradoxically, part of the complexity is introduced to ease software development and maintainability. This includes self-managed thread/process creation for specific functionalities, subscription of services through callbacks, and abundant use of objects and pointers shared across different modules, to name a few. As a representative and industry-level AD system, Apollo is built upon all this complexity. However, complex software structures can create unobvious and dynamic cross-process dependencies that available resource usage assessment tools fail to capture, thus being unable to measure, for instance, the actual execution time and memory requirements of AD software modules in general, and for their functions, in particular. Hence, validation teams lack the means to perform their work for AD frameworks.

The effectiveness and scalability of traditional V&V approaches are threatened by the complexity and unboundedness of the input and result spaces of functionalities such as perception and tracking [4, 70]. 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 [70]. As a result, it is hard to define budgets for software timing, relevant criteria for software timing V&V, and adequate testing methodologies.

In this Thesis, we focus on both resource verification factors, memory resource usage and timing budgets. In particular, we show the challenges to measure memory resource usage in sophisticated AD systems. Our study on Apollo AD framework shows that measuring resource usage is dramatically more complicated in comparison with traditional real-time systems. Regarding timing budgets of Apollo we show that in addition to the complexity of measuring timing budgets of highly coupled modules, we face significant timing variability across various modules.

In this line, contributing to the state-of-the-art with an 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. Nevertheless, we started working in this direction by, first, using measurement-based analysis to perform timing analysis on Apollo; and, second, proposing a set of guidelines to collect execution time and memory utilization measurements of AD modules and their components, thus enabling resource usage testing, as mandated in the automotive safety regulation ISO 26262 [33]. In particular, our contributions are the following:

1. Focusing 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 to apply established timing analysis approaches [1, 59, 79, 80].

2. Analysis of the execution time variability of Apollo when run on a GPU-based platform. We focus on the jitter in the per-frame processing time of each Apollo's module. Also, we performed a deeper analysis of the Apollo's camera object detector, which is one of the most critical processes of the Perception module.

3. Analysis of some of Apollo's sources of non-determinism with emphasis on its built-in randomization features. We analyse an example function in Apollo, Random Sample Consensus (RANSAC) fitting algorithm, that instantiates specific randomization properties.

4. Analysis of the difficulties and roadblocks to collect timing and memory utilization measurements of an AD software framework, using Apollo framework in general, and one of its key module (Perception), in particular, as a representative industry-level software module for guiding the discussion.

5. A set of remedies and guidelines to defeat those roadblocks, *En-Route*, to perform the resource usage testing of AD software in general, and Apollo in particular, with specific focus on timing and memory utilization concerns. *En-Route* guidelines aim at setting the basis for the development of a full methodology.

6. Assessment of *En-Route* on Apollo's Perception module. In particular, we showcase how execution times can be collected at fine granularities despite the complex and dynamic execution constructs of Apollo, and how memory utilization can also be collected and broken down across different Perception software components.

1.3 Thesis Organization

The rest of the Thesis is organized as follows. Chapter 2 presents background on the safety-related software development process and our reference AD system, the Apollo AD framework. Chapter 3 presents our analysis of the execution time variability of Apollo and the reasoning on the sources behind the observed variability. Chapter 4 describes the difficulties and roadblocks we suffered to collect timing and memory usage measurements of Apollo; *En-route*, our guidelines to defeat those roadblocks; and finally an assessment of *En-route* on Apollo's Perception module. Chapter 5 shows other works that are related with this Thesis. Chapter 6 concludes it and gives some insights about the future work to be done in this area.

The work done in this Thesis has been published in the 35th ACM/SIGAPP Symposium On Applied Computing (SAC) [5], and in the 26th Real-Time and Embedded Technology and Applications Symposium (RTAS) [6].

Chapter 2

Background

In this chapter, we provide some background on the development process of automotive systems as stipulated in the ISO 26262 [33] safety standard, with emphasis on the software part, as well as more detail on the Apollo AD framework [4, 14, 70].

2.1 Safety-Related Software Development Process

ISO 26262, the main functional safety standard for road vehicles, provides guidance on how to develop automotive safety-related electric and electronic systems. Following the hazard and risk analysis, safety goals and requirements are identified for the different software items. This process is followed by decomposition of each software item into atomic software and hardware units that need to be implemented without further decomposition. This process also propagates safety requirements to each item following specified decomposition rules. As a result, each item is attached an Automotive Safety Integrity Level (ASIL), ranging from A to D, where D is the most stringent safety level and A the least. Alternatively, some components are not allocated any safety requirement, thus being tagged as Quality Managed (QM), meaning that safety regulations do not impose any requirement on them. All safety-related items (those with some ASIL) undergo a design, V&V process, as dictated by ISO 26262, to obtain enough evidence that those items meet their safety requirements to a sufficient extent.

In the case of software, the development process in ISO 26262, see figure 2.1, consists of the requirements specification (6-6), software architectural design (6-7), and unit design and implementation (6-8) to reach the actual product. Then, the V&V phase starts with software unit testing (6-9), software integration testing (6-10), and software safety requirements verification (6-11). As part of this process, and, in particular, during unit and integration testing, resource usage testing may be performed to assess whether specific software items at different granularities (software units and integrated software) adhere to their requirements. However, those tests may still be limited due to the low level of integration at that stage, and resource usage testing must generally be repeated during the system V&V phase[1].

On the one hand, verification [34] is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. In software verification, two of the most used techniques to analyse the timing of software units are Static Timing Analysis (STA) and Measurement-Based Timing Analysis (MBTA) [79]. The main difference between both techniques is that STA measures the software timing through analytic methods, without the need of actually executing the program on real hardware nor simulators, while MBTA needs real execution times, collected under controlled scenarios, intending to predict what could happen in other scenarios. However, both techniques aim to estimate the WCET of the software unit under analysis. We expand deeply these concepts in section 3.1. On the other hand, validation [34] is the process of providing evidence that the system, software, or hardware and its associated products satisfy requirements allocated to it at the end of each phase, solve the right problem, and satisfy intended use and user needs. I.e., it is the confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled.

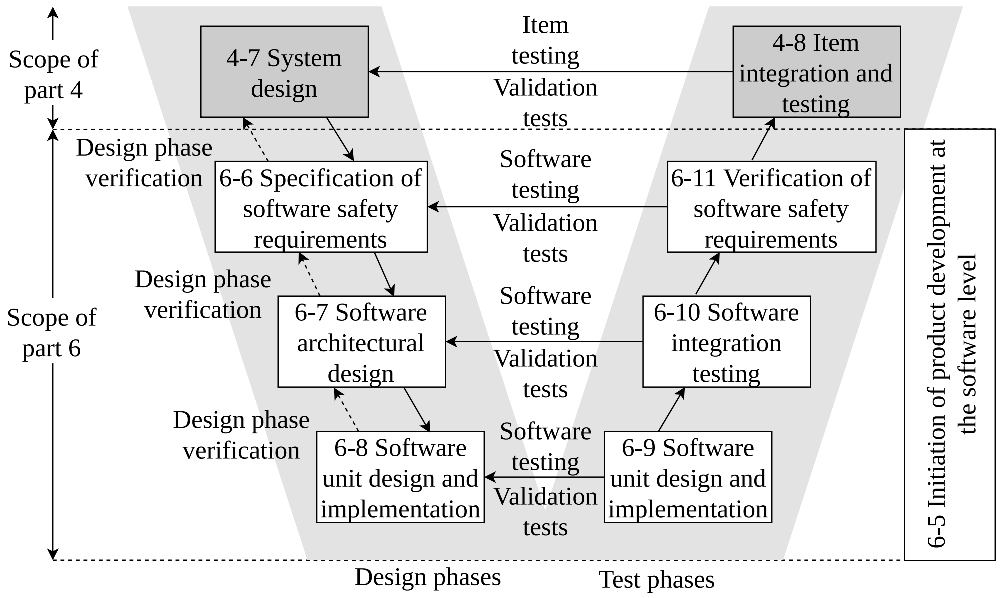

Scope of part 4
4-7 System design
Item testing
Validation tests
4-8 Item integration and testing
Design phase verification
6-6 Specification of software safety requirements
Software testing
Validation tests
6-11 Verification of software safety requirements
Scope of part 6
Design phase verification
6-7 Software architectural design
Software testing
Validation tests
6-10 Software integration testing
Design phase verification
6-8 Software unit design and implementation
Software testing
Validation tests
6-9 Software unit design and implementation
6-5 Initiation of product development at the software level
Design phases
Test phases

Figure 2.1: Software development process as described in ISO 26262 (picture taken from ISO 26262 Part 6 [33]).

System-wise, see figure 2.2, after the specification and system design, hardware and software product development occurs, where software development is shown in figure 2.1. Software and hardware items are then integrated to form a subsystem and, as indicated before, some testing is performed.

[1]ISO 26262 Part 6, devoted to product development at the software level, already states that "some aspects of the resource usage test can only be evaluated properly when the software integration tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests".

At this stage, since the platform is closer to its final state, further testing processes with higher confidence can be performed. For instance, using hardware-in-the-loop environments where a Simulink model feeds the subsystem and its outputs are obtained with a host that validates them either real-time or simply logs them for some offline processing.

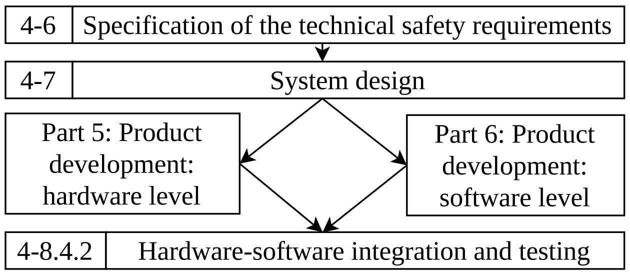

4-6
Specification of the technical safety requirements
4-7
System design
Part 5: Product development: hardware level
Part 6: Product development: software level
4-8.4.2
Hardware-software integration and testing

Figure 2.2: System development process as described in ISO 26262 (picture elaborated from ISO 26262 Part 4 [33]).

The aim of the resource usage test process during integration phases includes the following objectives:

1. Measuring minimum and maximum execution time, where the latter is of particular relevance for real-time systems.
2. Measuring memory requirements, in any type of storage (e.g., Flash memories, DRAM, SRAM, ROM) for code, (static) data, stack and heap.
3. Assess whether the task scheduling allows preserving all safety timing constraints (i.e. all tasks finish by their deadlines).

This information allows the integrator detecting unacceptable resource usage, as well as identifying the particular software component(s) causing excessive usage. For instance, the type of output obtained from these tests may be summarized in tables such as table 2.1 and table 2.2. In particular, for different software items of a hypothetical combustion engine, these tables show the measured and budgeted (planned) CPU and memory usage (DFLASH, PFLASH, and, RAM) in an Infineon AURIX CPU. By comparing expected and real values, engineers can determine whether budgets allocated where sufficient and, if they aren't, then they must debug the design to understand what was mispredicted and fix it.

2.2 Apollo Autonomous Driving System

Apollo [14] is an industrial-quality AD software framework with over 120 industrial partners, most of them top-tier AI tech companies and car manufacturers. Apollo has been already deployed on a variety of prototype vehicles (including autonomous trucks) and supports state-of-the-art

		Planned				
Item	Task	CPU	DFLASH	PFLASH0	PFLASH1	RAM
Pos mngmt	10ms	3%	280	120	0	80
Angle	5ms	9%	36	0	768	24
Torque monitoring	40ms	1%	16	16	0	0
Accel monitoring	40ms	1%	16	16	0	0
Power mode	2ms	4%	46	376	0	240
Torque_CTRL1	20ms	29%	2240	0	880	540
Torque_CTRL2	20ms	28%	2048	0	860	512

Table 2.1: Example of output of resource usage tests (planned part). Memory occupancy is given in KBs.

		Measured				
Item	Task	CPU	DFLASH	PFLASH0	PFLASH1	RAM
Pos mngmt	10ms	4%	308	168	0	96
Angle	5ms	9%	29	0	768	12
Torque monitoring	40ms	1%	10	21	0	0
Accel monitoring	40ms	1%	24	22	0	0
Power mode	2ms	2%	51	263	0	168
Torque_CTRL1	20ms	26%	2688	0	1320	270
Torque_CTRL2	20ms	14%	2048	0	774	410

Table 2.2: Example of output of resource usage tests (measured part). Memory occupancy is given in KBs.

hardware [8] such as the latest ADs[2] and cameras, from Velodyne [42] and other vendors, as well as GPU acceleration. It offers its partners the opportunity to develop their own AD systems through on-vehicle and hardware platforms. Regarding its software implementation, Apollo, similarly to most state-of-the-art AD systems, consists of a set of modules [9, 57], see figure 2.3. Each of the modules implement a crucial functionality of AVs. The main Apollo modules are as follows:

- **Perception** identifies the area surrounding the AV by detecting objects, obstacles, and, traffic signs, and it is considered as the most critical and complex module of an AD system. Perception module fuses the output of several types of sensors such as LiDAR, radar, and camera to improve its accuracy.
- **Localization** estimates where the AV is located, using various information sources such as GPS, LiDAR and an Inertial Measurement Unit (IMU). State-of-the-art localization algo-

[2]LIDAR, which stands for Light Detection and Ranging, uses laser pulses to build a 3D model of the environment around the car. Essentially, they help autonomous vehicles “see” other objects, like cars, pedestrians, and cyclists.

rithms, 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.
- **Routing** tells the AV how to reach its destination via a series of lanes or roads.
- **Planning** plans the spatiotemporal trajectory for the AV to take.
- **Control** executes the planned spatiotemporal trajectory by generating control commands such as accelerate, brake, and steering.
- **CAN Bus** (Controller Area Network Bus) is the interface that passes control commands to the vehicle hardware. It also passes chassis information to the software system.
- **Map** is similar to a library. Instead of publishing and subscribing messages, it works as a query engine support, which provides ad-hoc structured information regarding the roads.
- **HMI** (Human Machine Interface, or DreamView in Apollo) is a module for viewing the status of the vehicle, testing other modules and controlling the functioning of the vehicle in real-time.
- **Monitor** is the surveillance system of all the modules in the vehicle, including hardware.
- **Guardian** is a safety module that performs the function of an Action Center and intervenes should Monitor detect a failure.

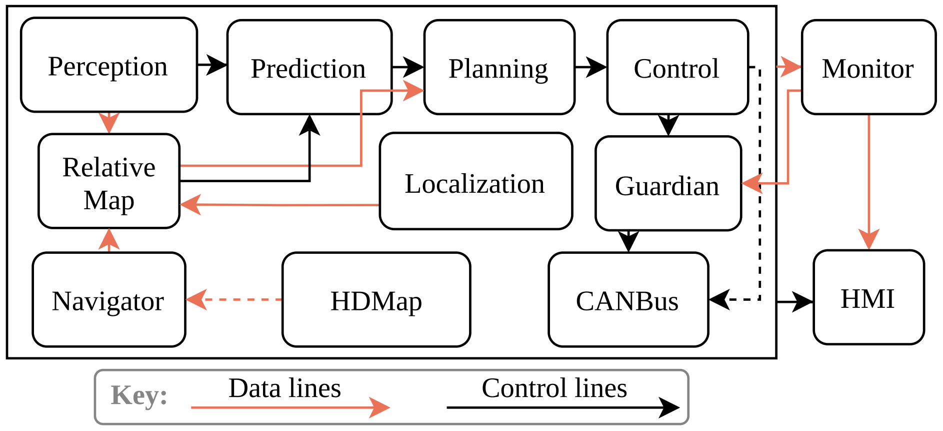


Figure 2.3: Interaction between Apollo's modules (picture taken from Apollo's documentation [9]).

For the sake of facilitating the installation and dependencies between numerous libraries, Apollo is provided inside several Docker container images. A container is a standard software unit that packages up code and all its dependencies so the entire application can run in a quick and reliable way from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

All modules in Apollo are implemented as *ApolloApps*, whose execution follows the code in figure 2.4. As it can be seen, Apollo uses three libraries for different purposes:

```
#define APOLLO_MAIN(APP)
  int main(int argc, char **argv) {
    google::InitGoogleLogging(argv[0]);
    google::ParseCommandLineFlags(&argc, &argv, true);
    signal(SIGINT, apollo::common::apollo_app_sigint_handler);
    APP apollo_app_;
    ros::init(argc, argv, apollo_app_.Name());
    apollo_app_.Spin();
    return 0;
}
```

Figure 2.4: Main function of an *ApolloApp*.

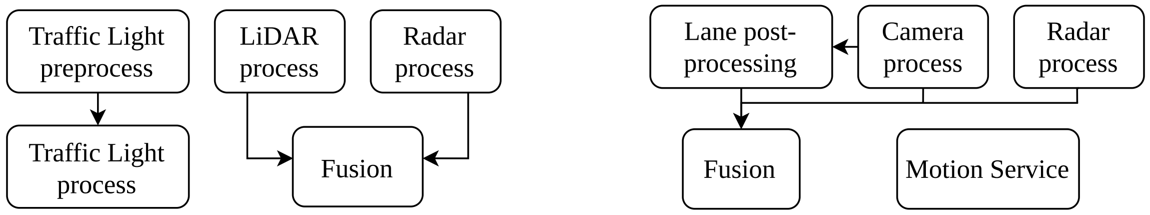
- The Google Logging Library (glog) [63], which implements an application-level logging, and provides logging APIs based on C++-style streams and various helper macros. This library contains the function `google::InitGoogleLogging`, which initializes it.
- The Google Commandline Flags (gflags) [62], which implements a C++ command-line flag processing. This library contains the function `google::ParseCommandLineFlags`, which looks for flags in `argv` and parses them.
- The Robot Operating System (ROS) [58] is a set of software libraries and tools that help building robot applications. Several AD systems, including Apollo, are relying on ROS as an middleware and communication system across modules. Function `ros::init` is from ROS and it is needed before calling any other *roscpp* (C++ implementation of ROS) functions in a node. Each *ApolloApp* is a ROS node.

To sum up, one module starts with the initialization of *glog* and ROS, and also loads the parameters from the `argv` and parses them using *gflags*. These parameters are given to the application through configuration files or as flags in the command line. After that, the module calls the `Spin` function before finishing its execution. This function initializes one or more ROS *spinners*. Then, they call their `spin` functions, which execute all the callback functions that are triggered during the runtime, until the client shuts down the module. A callback function is connected to a specific event, and it is triggered when this event occurs. In terms of ROS, a function can subscribe to an event (topic) and publish an event as well. But not only functions can publish an event or subscribe to it, other components of the AV such as sensors or the controllers of the accelerator, steering wheel or gearbox can do it too. Thus, ROS acts as a link between Apollo's modules, the car hardware, and also between both.

2.2.1 Perception Module

The Perception module [11] is in charge of the detection of the obstacles that surround the car. Its main functionality is to transform data from sensors (images, point clouds, etc.) into obstacles, thus knowing relevant information about them, like their position, size, orientation, etc.

The global configuration of input sensors for the Perception module can be represented as a Direct Acyclic Graph (DAG). With this, Apollo offers the possibility of building customized configurations, according to the requirements and the available hardware. These DAGs, along with other parameters of the input sensors, are defined in a configuration file. Apollo has implemented some of these configurations, which are available in the source files. In this work, we consider the DAG configurations shown in figure 2.5, as they are the ones that we could execute with the data (ROS bag files, see section 3.3.2) that Apollo provides.



(a) LiDAR (b) Camera

Figure 2.5: DAGs for two different configurations.

In these DAGs, nodes correspond to different processes and each of them is responsible for completing a specific task. Arrows indicate data dependencies between nodes of each DAG. For instance, in figure 2.5b, *Fusion* requires the output data of *Lane post-processing*, *Camera process*, and *Radar process*.

Apollo employs a variant of You Only Look Once (YOLO) [28,69] within the *Camera process* node of the camera configuration (figure 2.5b), as the main part of the Perception module. YOLO is an award-winning, widely-used and state-of-the-art object detection approach. Its most computationally-intensive function is a Convolutional Neural Network inference algorithm. Every second, in an AV, each camera captures multiple frames, and the object detector processes them on a frame-by-frame basis. The main stages of the YOLO object detector module are:

- 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.

2.2.2 Beyond Apollo

In this Thesis, we study Apollo as a representative and well-known AD framework. There are other well-known AD systems such as Autoware [30] with similar software architecture design, using ROS and Docker containers, thus facing similar challenges to the ones explored in this work. ROS is extensively used in robotics and other domains due to the interfaces offered to integrate modules either time-triggered or event-triggered, making code maintainability a key feature of ROS. However, such advantage comes at the cost of using abundant pointers, indirections and abstraction layers that lead to significant testing difficulties, as discussed in the rest of the Thesis. Therefore, we focus on Apollo without lack of generality, and our contributions and findings can be naturally extended to other domains and frameworks.

It is also worth pointing out that, in this Thesis, we used Apollo's version 3.0 as a basis for our work and experiments. All explanations regarding Apollo's implementation focus on this version specifically.

Chapter 3

Timing Analysis of Apollo

In this chapter, we first discuss how the complexity of AD systems difficult the application of established timing analysis approaches. Second, we report the statistical properties of the system on analysis' execution time distributions, which must be known before applying some timing analysis technique. And, third, we analyse the sources of timing variability of Apollo, reinforcing what has been discussed first. Final insights in statistical and probabilistic approaches for Apollo's timing analysis, which are beyond the scope of this Thesis, are presented in [6].

3.1 Timing Analysis of Safety-Critical Systems

In the real-time system's domain, the system (that is the specific hardware and software to be deployed) must pass some strict certification tests, for both functional as well as timing validation [76]. Each specific industry domain needs to apply different certification processes, such as the ones described in ISO 26262 for the automotive industry. Timing analysis techniques are used to verify if the applications that run on real-time systems fulfil their timing constraints. Because of the increasing complexity of hardware and software, the correct verification of meeting constraints is getting harder, in time and cost. Two of the most used techniques in the industry, STA and MBTA [79], find serious difficulties to verify the timing constraints of current complex systems such as AD frameworks. These techniques focus on safely and tightly estimating the WCET bounds of a process to quantify the maximum time it can last. It is tough to derive the WCET of a specific task in particular hardware since all the possible factors that could have an impact on the execution time should be explored, which is unfeasible for AD software in general. These factors can be, for instance, the platform in which the task is going to run or the different possibilities of input sets it can receive, to name a few.

STA computes the WCET estimation of the task under analysis through analytic methods. This means that STA does not need timing measurements of the program running on a real platform.

The possible execution paths the application can take during its execution are analysed and then put into an abstract model of the platform. With this information, the model computes an estimation of the execution time each path lasts, and the final WCET is derived using these estimations. Conversely, MBTA estimates the WCET of the task by running the program on the objective platform or on a simulator, measuring the time it takes to finish. After several iterations of this process, a WCET bound is computed in base of the execution time distribution observed and using the proper statistical methods.

STA 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 [1, 59, 79, 80]. 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 [1,59]. 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 STA. 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 [1, 39]. 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 MBTA approaches [79], 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 [1, 79]. This scenario complicates 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 reducing their predictive value [1]. 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 [46], and ultimately does not allow to reach a satisfactory level of path coverage [12].

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, 24], generally fails to deliver sufficiently tight results. Dynamic approaches, instead, try to design specific tests to trigger the worst-case contention scenario [16], which is generally out of reach.

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 [60]. 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 [60], 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 [60] 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 [35], 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 [35, 49]. 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 [56]. 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 [33]. 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 [35].

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 [3]. Certification arguments to fit probabilistic reasoning in current standards have been already explored in the literature [66], showing that measurement-based probabilistic timing analysis 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 in this chapter, 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 [3], 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.

3.2 Motivation

To illustrate AD software's extremely variable timing behavior, figure 3.1 shows boxplot diagrams of the observed per-frame execution time variability (jitter) of each of the modules of the Apollo AD framework, when running under a representative set of inputs on our GPU-based target platform (see section 3.3). 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 3.2 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.

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 [79]. This occurs because such complexity undermines the accuracy and scalability of static analyses and the significance of measurement-based approaches [1].

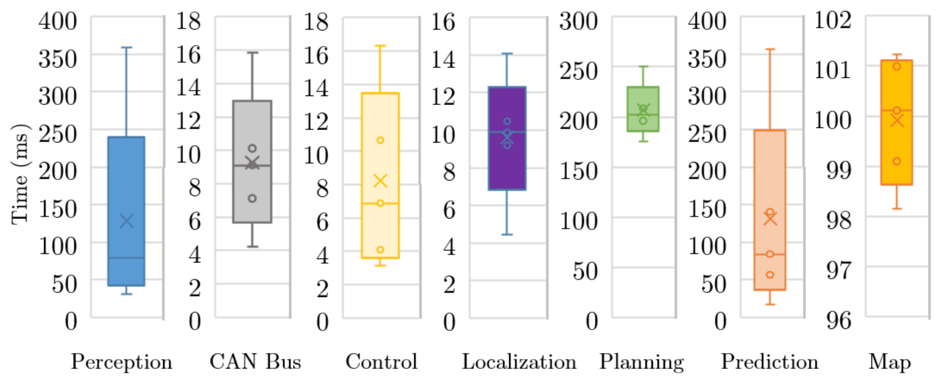


Figure 3.1: Observed per-module execution time of Apollo.

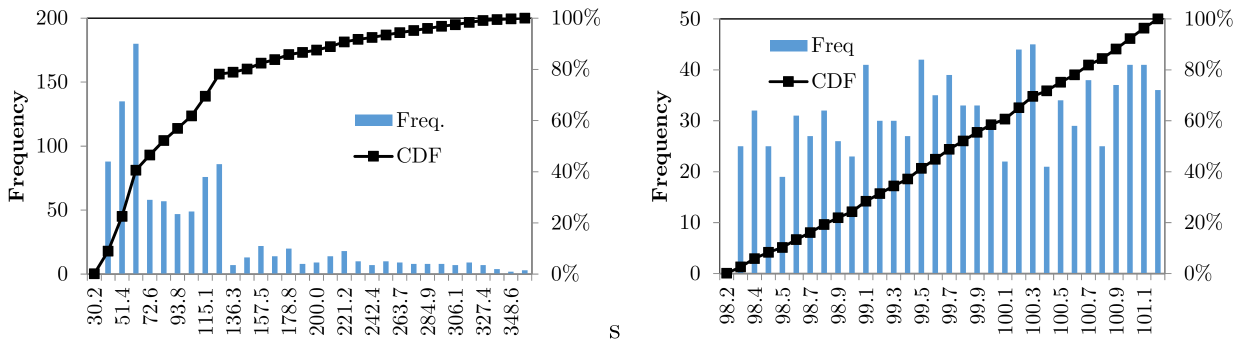


Figure 3.2: Execution time (ms) analysis of two Apollo modules.

3.3 Experimental Methodology

3.3.1 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, as mentioned before, it is built on top of ROS (Robotic Operating System) [58]. 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 3.5, 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.

3.3.2 Data

Apollo developers provide several data for different configurations and versions to test the framework, which is given as *bag* files. A bag is a file format in ROS for storing ROS message data. They are typically created by a tool like *rosbag*, which subscribe to one or more ROS topics, and store the serialized message data in a file as it is received. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics. So, playing back a bag file simulates a real situation recorded previously in a real scenario, thus enabling the framework's testing without owning an AV. For the version we are using, Apollo 3.0, we could only run Apollo successfully with two of the bag files they provide. One of them contains data coming from the LiDAR (point clouds), and the other from the camera (images), hence each of them matches the configurations presented in section 2.2.1.

3.3.3 Measurements

For time measurements, we used instrumentation points at module and node boundaries, at the granularity shown in figure 3.3. For modules using only the CPU, we use the `high_resolution_clock` of C++, which provides a high-resolution time counter. On the stages of the *Camera process* that use the GPU, 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 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

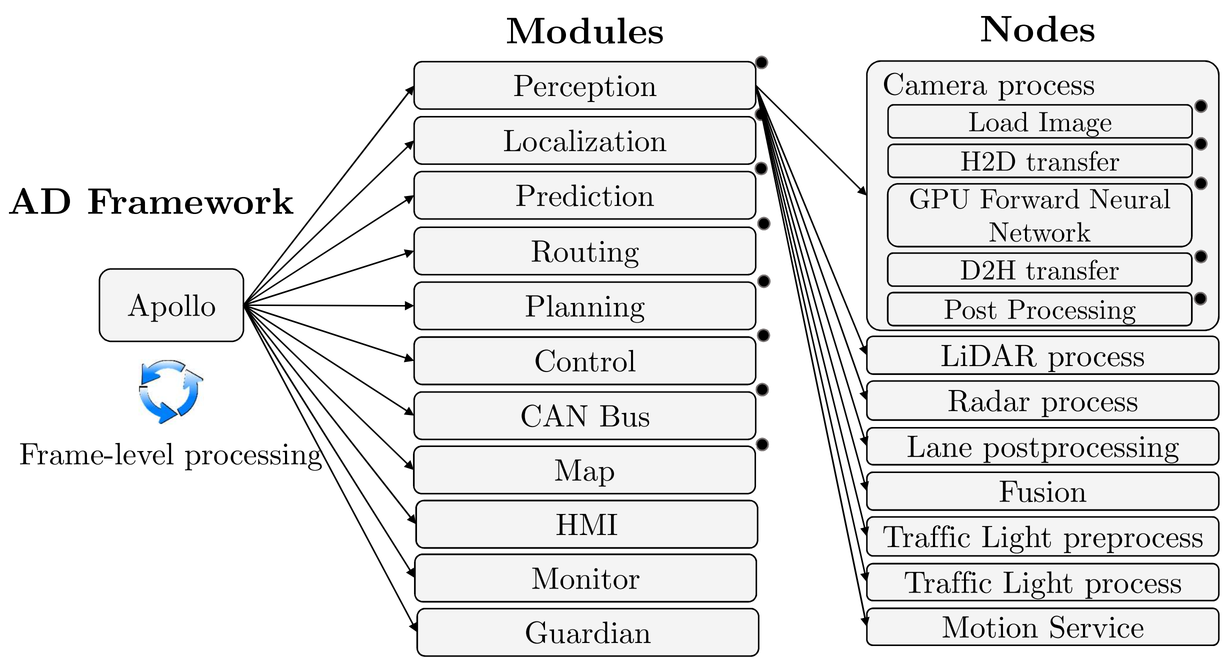


Figure 3.3: Apollo AD system pipeline. Dots indicate the instrumentation points we use for extract timing behavior.

the task in parallel. Hence, our instrumentation for obtaining time measurements can be regarded as having low overhead.

3.4 Execution Time Jitter

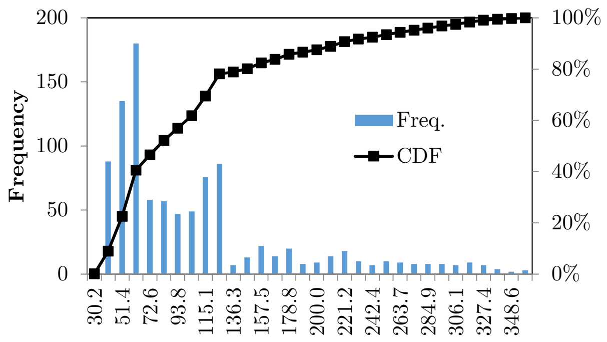
Next, we report the main statistical properties of the observed execution time distributions. We perform our analysis at module level except for the Perception module, where we perform a more detailed analysis at the stage level of the *Camera process* node.

3.4.1 Module Level Analysis

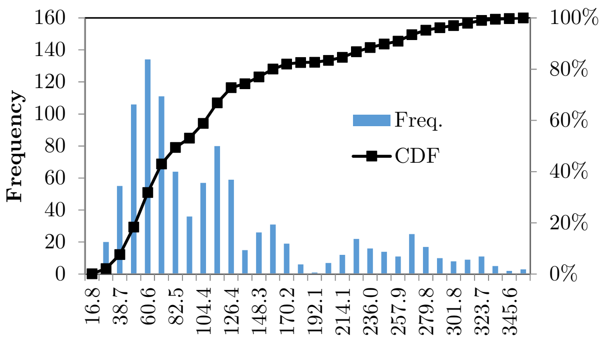
We measure execution times at frame-level and capture the resulting distribution for each of the modules when they process real tracing data collected by autonomous car sensors.

Figure 3.4 shows the observed execution time histogram and CDF of each Apollo module. As it can be seen, jitter distributions have different shape and dispersion, hampering their analysis. This phenomenon is quantified in table 3.1, which shows different measures of dispersion that allow us analysing and comparing the distributions. It shows:

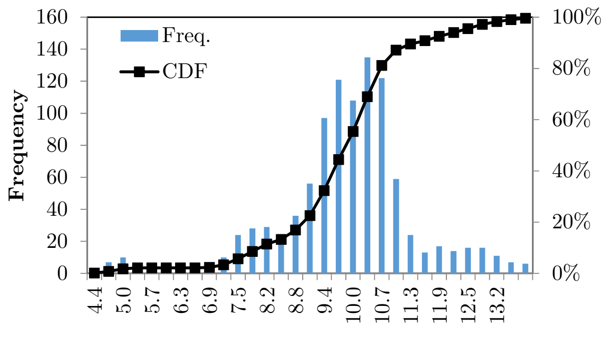
- (1) Minimum and (5) maximum values observed in the execution time sample.



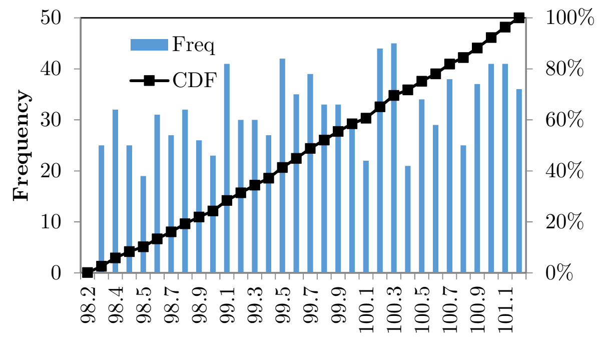
(a) Perception



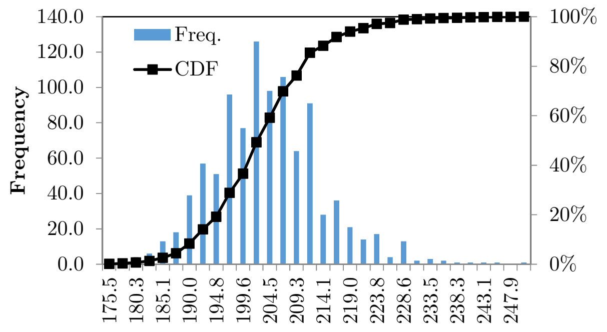
(b) Prediction



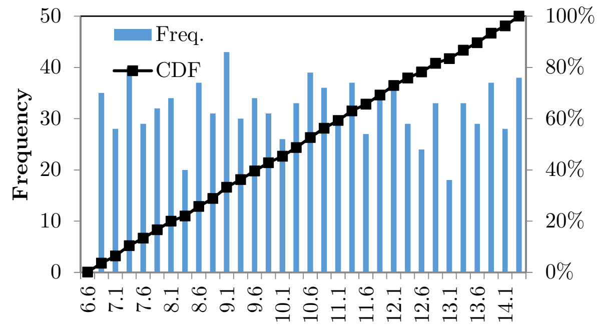
(c) Localization



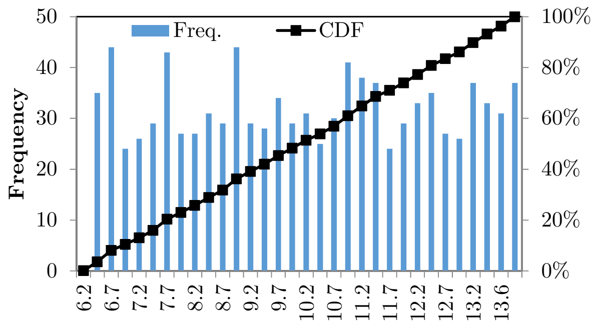
(d) Map



(e) Planning



(f) Control



(g) CAN Bus

Figure 3.4: Distribution of execution times (ms) of each module.

	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

Table 3.1: Measures of dispersion for Apollo modules. Values in the first 5 rows are in milliseconds.

- (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 3.1, 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 and 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 analysing 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.

3.4.2 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 analysing 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's YOLO object detector located in *Camera process* (see figure 2.5b).

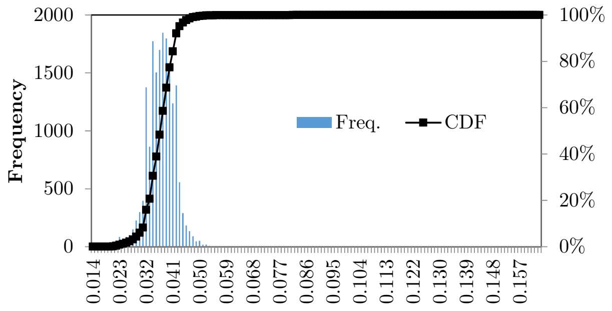
	Loading	**H2D**	**GPU**	**D2H**	**PPro**
(1) Min	0.01	0.68	39.02	0.94	5.33
(2) Q1	0.03	0.99	40.61	0.94	6.77
(3) Q2	0.04	1.06	40.86	0.94	7.24
(4) Q3	0.04	1.13	41.60	0.94	7.92
(5) Max	0.16	1.64	45.25	1.13	12.70
(6) CV	0.14	0.09	0.03	0.01	0.11
(7)IQRn	0.14	0.13	0.02	0.00	0.16
(8) Kurt	113.2	0.01	2.09	147.8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

Table 3.2: Measures of dispersion (YOLO stages). Values in the first 5 rows are in milliseconds.

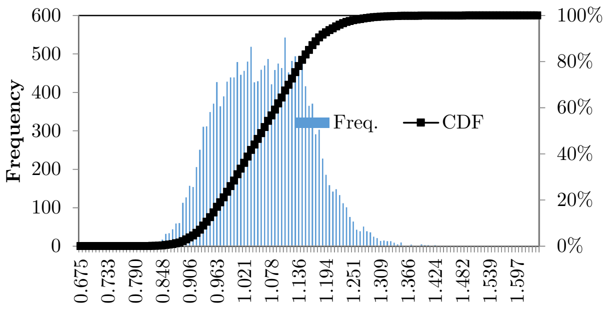
As it can be seen in figure 3.5 and table 3.2, all stages follow the observed trend at the module level as they exhibit significant jitter and have arbitrarily different distributions:

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.

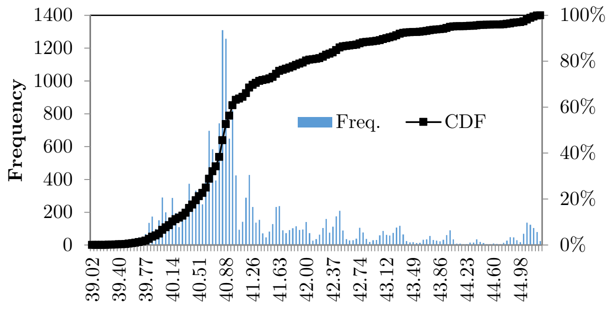
2. GPU kernels show tiny dispersion in the central part ($IQRn = 0.02$), but very large relative dispersion in the tails ($Kurt = 2.09$). While the maximum is far away from the central part of



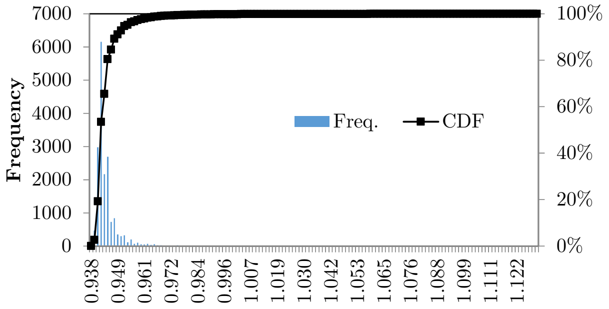
(a) Loading



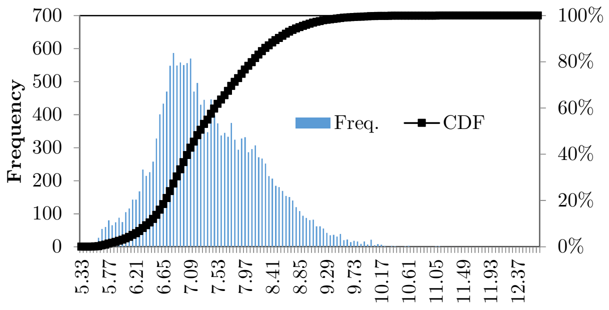
(b) H2D



(c) GPUkernel



(d) D2H



(e) PosPro

Figure 3.5: Execution time (ms) distribution of YOLO stages.

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.

As explained in section 2.2.1, the Perception module can have different shapes, and the *Camera process* is just one of the possible nodes that can compose the DAG. Moreover, YOLO is not the only work this node performs. Hence, camera object detection execution time (see figure 3.5 and table 3.2) is lower than the overall Perception execution time.

3.4.3 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.

3.5 Sources of Execution Time Variability and Impact Software Timing

3.5.1 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 [18] (more details on our hardware/software configuration are shown in section 3.3). 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

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 [14,17,21,22, 57]. 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 [27]. Randomization is also used in the Probabilistic Roadmap Method and Rapidly-exploring Random Trees, where random selection of configurations is a core step in the rapid generation of planning solutions [27]. Also in Perception, either model- or graph-based segmentation algorithms usually incorporate randomization elements [56]: a clear example of the former is RANSAC; when it comes to the latter, two illustrative examples are Conditional Random Field and Markov Random Field.

As a concrete example, figure 3.6 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 12, 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 analyse 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 [70]. Furthermore, these functions are event-triggered by events arriving from the sensors and other modules at different frequencies. This makes the analysis of Apollo inputs overwhelmingly complex.

In particular, we focus on the RANSAC algorithm introduced in figure 3.6, as it combines four deterministic input parameters and three randomized inputs . The input parameters 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, and with the data we used during the experiments, RANSAC is called 2,002 times each with a different set of values for the input parameters (`V`, `maxIters`, `N`, `inlierThresh`).

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

    Let vector $c$ = findSolution(c, colPivHouseholderQr($A$) $\cdot c = B$),
    $Inliers = 0$, $res = 0$ and $y = 0$
    For $i = 1, 2, ..., n$ do
        $y = V[i,0]^2 \cdot c[0] + V[i,0] \cdot V[i,0] \cdot c[1] + c[2]$
        If $|y - V[i,1]| \leq inlierThresh$ then $++Inliers$
        $res \mathrel{+}= |y - V[i,1]|$
    If $Inliers > maxInliers$ or ($Inliers = maxInliers$ and $res < minRes$) then
        $C[3] = 0$, $C[2] = c[0]$, $C[1] = c[1]$, $C[0] = c[2]$,
        $maxInliers = Inliers$, $minRes = res$
    If $Inliers > n \cdot earlyStopRatio$ then break
    If $maxInliers/n < goodLaneRatio$ then return False
    Else $T = V$
        $V = \text{clear}(V)$
        For $i = 1, 2, ..., n$ do
            $y = T[i,0]^2 \cdot C[2] + T[i,0] \cdot C[1] + C[0]$
            If $|y - T[i,1| \geq inlierThresh$ then $V = V \cup \{T[i]\}$
Return True
```

Figure 3.6: RANSAC fitting algorithm in Apollo's lane detection.

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 input 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 3.7 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.

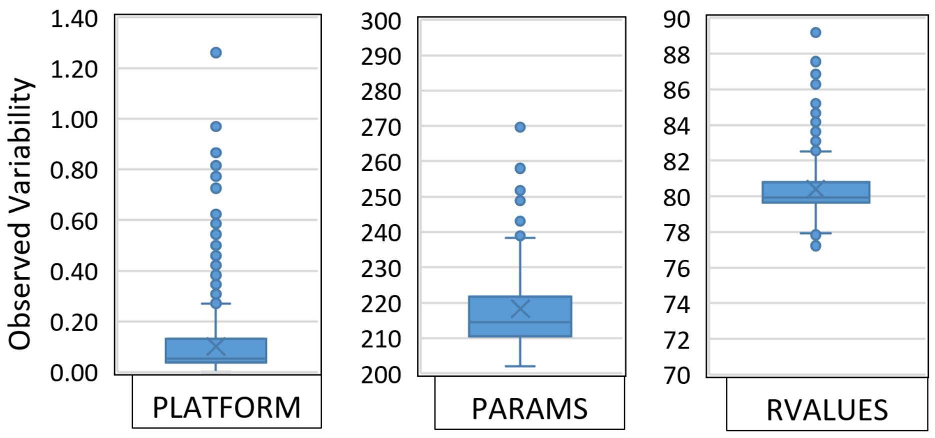


Figure 3.7: Execution time variability of RANSAC.

Overall, these results evidence the huge variability caused by random and deterministic input values, with reduced effect coming from the platform.

Chapter 4

En-Route

In chapter 3, we presented our solution for timing verification for AD frameworks like Apollo. Chapter 4, instead, focuses on validation. More precisely, on resource usage testing, a mandatory validation step during the integration of safety-related real-time systems.

4.1 Roadblocks for Resource Usage Testing on Autonomous Driving Software

Due to the stringent performance requirements of AD platforms, high-performance hardware is deployed to execute specific functionalities fast enough. For instance, input data sensed through a camera, LiDAR or radar, need to be processed at specific rates (e.g. 25 frames per second for camera-based input data). Since heavy parallel computations need to be performed at such high rates, hardware accelerators such as GPUs are needed [45]. This is the case for Apollo in general, and its Perception module in particular [11], whose most heavy computations are offloaded onto a GPU. The use of GPUs is the most common solution for massive computation requirements of such workloads. Therefore, Apollo's code is executed across CPUs and GPUs. Next, we review the difficulties experienced to perform resource usage tests in both computing components for the Perception module of Apollo as an illustrative example.

4.1.1 CPU Resource Usage Tests

We must first identify the tools to use to test the CPU parts of the Perception module (or any other part of any AD framework). In general, AD frameworks use arbitrarily complex programming constructs not suited for regular performance tools for safety-related systems, which are suited for highly-static program constructs, inline with the software development requirements imposed by ISO 26262. However, programming practices for Apollo differ noticeably from those indicated by

ISO 26262 and, instead, target different objectives such as performance efficiency, modularity and maintainability, which leads to the use of multiple threads, callbacks, asynchronous processing and the like.

To test such a complex CPU code, we considered initially the use of profiling tools such as *Valgrind* [75], *Google Performance Tools* (GPT) [64] or *Perf* (part of Linux), inherited from the general-purpose computing domain where programming constructs considered are less restrictive. In particular, our inspection of Apollo software revealed that Apollo developers have used GPT since, all the configuration hints needed for using it to profile Apollo are already embedded in Apollo's source files. In fact, Apollo documentation already includes detailed instructions to use GPT for profiling purposes [10].

We have profiled the execution of the Perception module of Apollo with GPT and results turned out to be disappointing. The Perception module runs several different nodes, depending on the input sensors available. For instance, for one of the input data sets provided along with Apollo, LiDAR and radar sensors are used to feed Apollo, and so 5 different nodes are used by Apollo (*LiDAR*, *Radar*, *Fusion*, *Traffic Light preprocess*, and *Traffic Light process* shown in figure 2.5a), which are managed by 5 different threads spawned automatically by the Perception module itself. Those nodes are in charge of performing the callback functions for each of those functionalities of Perception.

When using GPT to profile Perception, we obtained the call tree depicted in figure 4.1, where the fraction of execution time devoted to each of the functions is indicated along with each function. The first observation is that, despite all 5 Perception nodes are executed, the call tree obtained only reflects functions corresponding to the *Fusion* node and, despite all the other modules are also executed, GPT fails to provide any profiling information. In fact, we verified that the output of the execution was correct, matching the output of the non-profiled execution, and the 5 nodes were correctly spawned and executed as revealed by monitoring the execution of the framework. Thus, the first conclusion reached is that the complexity of Perception's structure already exceeds the capabilities of GPT. Moreover, even the call tree obtained does not reflect all functions executed as part of the *Fusion* node. In particular, as shown in the call tree, GPT reports that 95% of the execution time is spent running the `sleep` function. However, some functions processing large amounts of data that must be executed, are not reported by GPT, thus meaning that GPT even fails to profile properly a single Perception node. Note that, in order to validate our conclusions, that were primarily based on code inspection, we added debugging messages in the code in functions not shown in the call tree, both inside *Fusion* as well as in other nodes. The execution printed those messages, thus confirming the conclusions reached by code inspection on the fact that those functions were executed. Therefore, GPT simply failed to provide correct information in the call tree despite being the profiling tool recommended by Apollo developers.

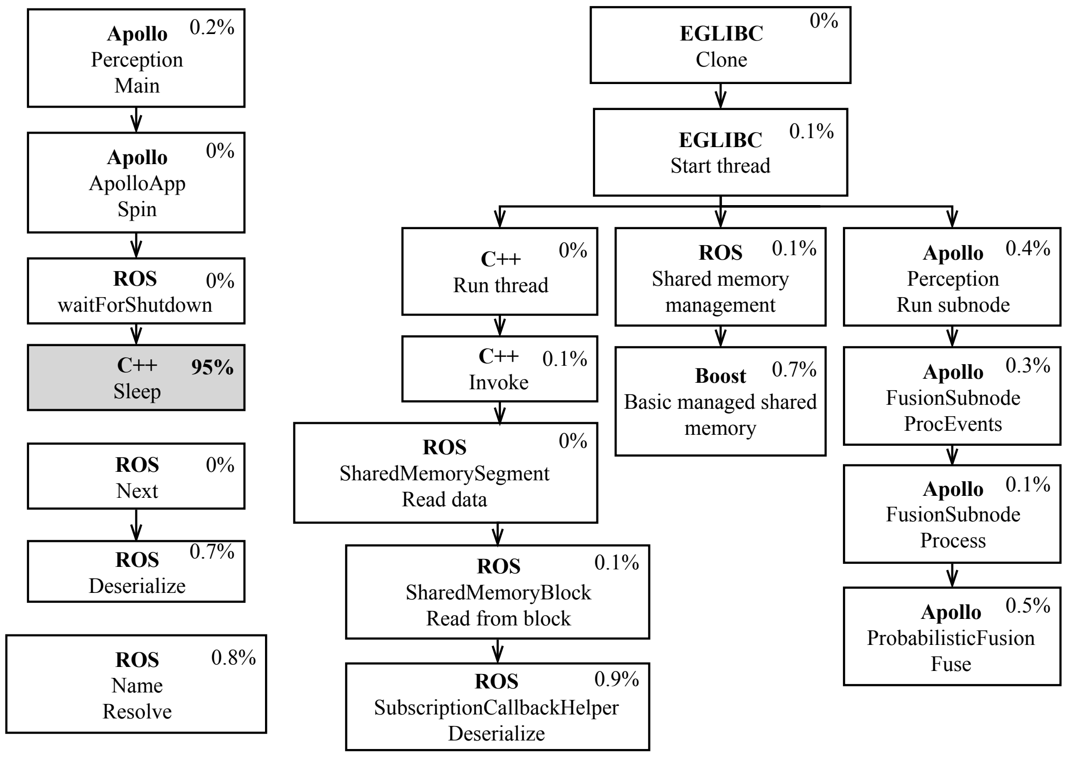

Apollo 0.2% Perception Main
Apollo 0% ApolloApp Spin
ROS 0% waitForShutdown
C++ 95% Sleep
ROS 0% Next
ROS 0.7% Deserialize
ROS 0.8% Name Resolve
EGLIBC 0% Clone
EGLIBC 0.1% Start thread
C++ 0% Run thread
C++ 0.1% Invoke
ROS 0% SharedMemorySegment Read data
ROS 0.1% SharedMemoryBlock Read from block
ROS 0.9% SubscriptionCallbackHelper Deserialize
ROS 0.1% Shared memory management
Boost 0.7% Basic managed shared memory
Apollo 0.4% Perception Run subnode
Apollo 0.3% FusionSubnode ProcEvents
Apollo 0.1% FusionSubnode Process
Apollo 0.5% ProbabilisticFusion Fuse

Figure 4.1: Call tree of the execution of the Perception module.

4.1.2 GPU Resource Usage Tests

We first identified two of the most suitable tools for profiling Perception's GPU code. Since it is intended to run on NVIDIA GPUs, we use *nvprof* [51] and NVIDIA Visual Profiler, which uses *nvprof*, for visualizing the profiled information. Then, when attempting to use *nvprof* to profile Perception, we experienced three issues, as detailed next.

Issue 1: no execution progress

The first and most challenging problem we have faced with *nvprof*, which occurred not only for the Perception module but for any Apollo module regardless of whether it uses the GPU or not, is that execution of the module seemed not to make progress at all, waiting in an infinite loop. Initially, we suspected that Perception was running slowly with the profiling tool rather than not making any progress, so we let the Perception module run for 24 hours. However, we observed no progress so we concluded that execution got simply stalled and the problem was not causing, instead, slow progress.

We attempted to find where and why execution got stalled, so we introduced printed messages in different parts of the module, but none of them was printed. Not even the message placed at the earliest possible execution point was printed. At this point, although we lack the means to double-check this hypothesis, we suspect that the problem relates to libraries loaded along with Apollo whose source code is not available and hence, cannot be inspected as we do for Apollo's open-source code.

As part of the debug process, we came out with some conjectures on whether the source of the stall with *nvprof* was the fact that CUDA calls occurred through multiple threads or because those threads were launched by ROS. For that purpose, we developed two programs with those features and profiled them with *nvprof*.

The first program creates several threads so that each of them launches and runs a CUDA kernel. The program is run and tested inside the Docker container to verify that, by using the container it does not affect the profiling process.

The second program, uses ROS with two nodes, a *subscriber* and a *publisher*. The *publisher* publishes ROS messages and whenever the *subscriber* receives a message, it creates several threads to launch and execute CUDA kernels. This program is designed to verify that the profiler is able to capture CUDA kernels that are launched through threads within ROS nodes.

In both cases, profiling worked properly with no stall at all, so we concluded that those code constructs are not per se the source of Perception's stall when profiled with *nvprof*.

Finally, we changed a number of profile options such as `profile-child-processes` or `profile-all-processes` without success. The first enables the profiling of the application and all child processes launched by it, and the second enables the profiling of all processes launched by the same user who launched the *nvprof* instance. In fact, only when we disabled the option to profile the application from start (`profile-from-start off`) execution progressed as expected. Nevertheless, this feature, as indicated by its name, disables profiling, so that, in order to profile Perception, we have to identify the parts of the code that we want to profile in order to apply a focused profiling. This is, in general, unwanted, since this increases the burden on the user side to identify what parts of the code need being profiled instead of letting the profile tool simply profile the whole module under analysis.

Issue 2: CUDA kernel identification

Related with the previous issue, and given that we want to test the resource usage performed by the GPU code, we need to identify those code sections where CUDA kernels are launched. However, this is a cumbersome task since Apollo builds upon a modified version of Caffe [36], a framework intended to manage Artificial Neural Networks, which are the most computing intensive element of Perception. Such framework makes extensive use of the GPU. However, CUDA calls are performed through a number of function calls that increase the difficulties to trace what particular calls are used and where in the code.

Issue 3: Lack of support for memory usage testing

The third issue relates to the lack of support to measure the memory usage performed by the code executed in the GPU. In particular, resource usage testing needs to determine not only end-to-end resource requirements, but also the requirements at finer granularities to help debugging and optimization during the development process. Unfortunately, we have been unable to identify any suitable tool that allows collecting this information for GPU code in an easy manner.

Overall, the presented problems challenge resource usage testing in complex AD frameworks such as Apollo. In the next section, we provide appropriate solutions to tackle these problems.

4.2 Guidelines

We introduce *En-Route*, our set of guidelines to enable resource usage tests for AD frameworks. Next, we introduce *En-Route* guidelines for CPUs and then for GPUs.

4.2.1 For CPU Platforms

Besides GPT, we have evaluated to what extent *Callgrind* (a profiling tool from *Valgrind*) and *Perf* allow performing resource usage testing for the CPU code of Perception. *Callgrind* simply did not work with Perception, so we discarded it. *Perf*, although provided better results than GPT, failed to profile Perception completely providing accurate measurements for all functions. Overall, we found no tool providing appropriate support yet. Thus, there is a business opportunity for software vendors to develop appropriate tools for resource usage testing of complex CPU code.

As part of *En-Route*, we had to rely on engineering work together with the limited support of tools such as *Perf* to build the call tree of Perception. Once this information was obtained, it was obvious where to place timers systematically to collect execution times at any desired granularity. Analogously, memory usage could be obtained using the *Massif* tool (part of *Valgrind*) [74]. In any case, CPU code has low memory requirements since heavy processing and thus, large sets of data collected from sensors, occur in the GPU in AD frameworks.

Overall, *En-Route* provides guidelines to address all roadblocks that impede otherwise performing resource usage testing in AD frameworks. However, as discussed before, a number of processes require some degree of user intervention due to the lack of appropriate tools. Yet, those processes which are not automated, are systematic in nature and tools can be developed to perform them. Thus, while being a disadvantage in the current state, the lack of automation of those processes is an opportunity for software vendors to develop and commercialize appropriate tools.

4.2.2 For GPU Platforms

En-Route addresses the issues identified in previous section, namely execution progress, CUDA kernel identification, and memory usage testing.

Execution progress

As explained before, we observed execution progress only when we disabled the `profile-from-start` option of *nvprof* (with value `off`). This, however, disables by default any profiling, so we need to introduce calls to `cudaProfilerStart` and `cudaProfilerStop` (CUDA Profiler API) in appropriate code locations to profile relevant code sections (i.e. those using the GPU). We have used these calls and assessed that they allow profiling specific sections of the Perception module, obtaining execution time information for all the CUDA kernels and API functions that the module calls within the code region profiled. Figures 4.2 and 4.3 show an example of how to use them.

```bash
nvprof -t timeout --profile-from-start off ./foo args
```

Figure 4.2: Bash command to perform the profiling in selected sections of the `foo` program.

```cpp
#include <cuda_profiler_api.h>
...
void foo (...) {
  ...
  cudaProfilerStart();

  // Section you want to profile

  cudaProfilerStop();
  ...
}
```

Figure 4.3: Example of C++ code that selects the section of code to be profiled.

Once profiling has been enabled, another issue appeared: how to stop execution to collect profiling information. Apollo, as any other AD framework, is intended to run continuously. Its execution can be terminated correctly sending a `SIGINT` signal. This signal triggers a function that stops all processes correctly and finishes their execution. However, when running Apollo profiled with *nvprof*, the `SIGINT` signal may be received by *nvprof* instead of Apollo, thus terminating the profiling process in a way that profiling information is not collected rather than terminating Apollo itself. In order to solve this problem, we came out with a solution that consists of the following steps:

- Set the `timeout` option of *nvprof*. Note that AD frameworks perform all their activities in a loop with specific deadlines. Hence, this information can be used to set the `timeout` to profile the appropriate number of iterations of each functionality.

- Let Apollo run longer than the scheduled `timeout` before sending a `SIGINT` signal, which will therefore arrive when *nvprof* has already finished. At this point, profiling information collected by *nvprof* has been recorded correctly, thus providing information on execution time of GPU-related code.

CUDA kernel identification

Identifying the code sections where profiling is needed, and so where the CUDA Profiler API needs to be used, is easy in simple programs. However, the Apollo framework has a complex structure, thus challenging the identification of the location of CUDA calls. Apollo builds upon Caffe for its Artificial Neural Networks, and it turns out not to be trivial identifying what particular functions of Caffe need being profiled, which would require inspecting all functions of all nodes of Perception (or other modules if they would have used GPU) to identify the Caffe functions to be profiled.

To simplify this process, *En-Route* imposes the profiling of all functions, thus relieving end users from having to track what functions are used in practice. Since this task would be tedious if applied manually, we have developed a Python script that automates the insertion of the profiling calls, thus easing the work of end users.

Memory Usage Testing

The last issue to solve for GPU code relates to the difficulties to obtain information about the memory usage of the CUDA calls. As explained before, no specific tool provides this feature on its own. Thus, to obtain memory usage information, *En-Route* builds upon the combination of two tools: the GNU Project Debugger [25] (*GDB*) and the NVIDIA System Management Interface [26] (*nvidia-smi*). In particular, our solution requires user intervention to determine the granularity at which memory usage must be assessed, and introduce breakpoints with *GDB* at the corresponding locations. Then, when running Perception and a breakpoint is reached, we use *nvidia-smi* to query how much memory is being used in the GPU, thus allowing to measure the amount of memory required at each point of the execution, as well as the amount of memory allocated between two consecutive breakpoints.

4.3 Experimental Methodology

4.3.1 Platform

Experiments were done on top of an NVIDIA AGX Xavier development platform intended for automotive systems [61]. It has an octa-core CPU based on Carmel ARM V8 64-bit architecture, and an NVIDIA GPU with 512 CUDA cores based on the Volta architecture. The AGX Xavier [78] also includes two Deep Learning Accelerator (DLA), Programmable Vision Accelerator (PVA), and a set of multimedia accelerators providing additional support for machine learning. The In-System Programming (ISP) has been enhanced to provide native HDR support, which is a promising property for camera-based object detectors, and higher precision math without offloading work to the GPU. Xavier features a large set of I/O and has been designed for safety and reliability supporting various standards such as ISO 26262 with ASIL level C. The CPU cluster is fully cache coherent and the coherency is extended to all the other accelerators on-chip. At platform level, Xavier features NVLink 1.0 supporting 20 GB/s in each direction for connecting a discrete graphics processor to Xavier in a cache coherent manner. In addition, it offers PCIe Gen 4.0 support (16 GT/s). In figure 4.4, the Xavier block diagram is depicted which shows the different elements we mentioned above and how they are linked with the block and with other components.

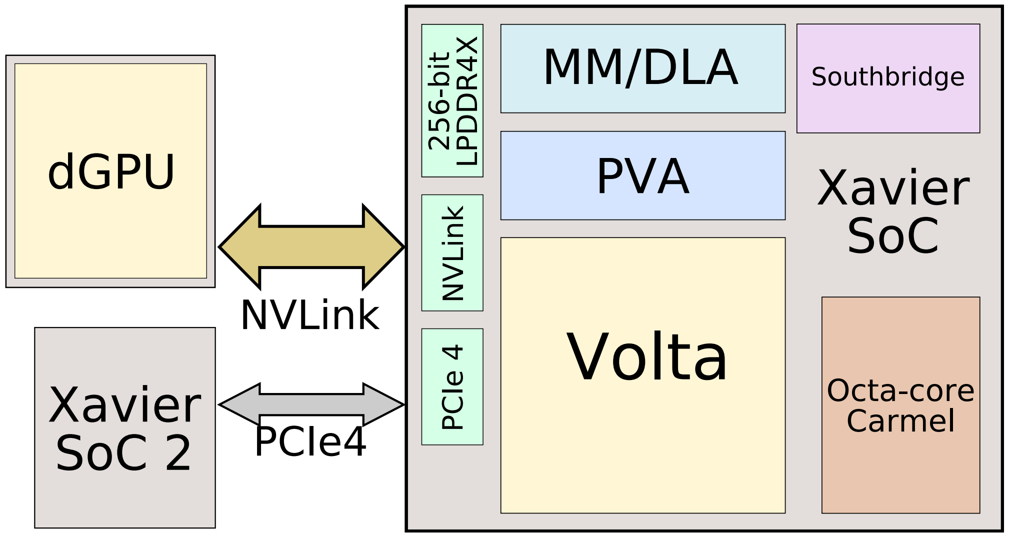


Figure 4.4: NVIDIA Xavier block diagram.

4.3.2 Data and Measurements

The data used in the experiments showed in this chapter, as well as how time and memory measurements were taken, are the same than in section 3.3.

4.4 Evaluation and Experimental Results

This section applies *En-Route* guidelines to the Perception module of Apollo. We provide execution time tests at node granularity for the CPU. Then, we provide execution time for GPU kernels. Finally, we provide results of the memory requirements tests.

4.4.1 Execution Time Usage Tests

For CPU Platforms

We have collected execution times for the different nodes in figures 2.5a and 2.5b. Note that node execution time in the CPU also includes GPU execution times for those nodes using the GPU. The relative execution time for each node is shown in figure 4.5, where each of the two input data setups (LiDAR and camera configurations) is normalized w.r.t. its total execution time. As shown, *En-Route* allows testing how much each function or node contributes to the total execution time of the module analysed. In particular, we observe that *Fusion* has a large contribution to the overall execution time for both input data sets, whereas *Radar* has a low contribution instead. We also note that there are three nodes in each case that take almost $\frac{1}{3}$ of the total execution time each: *Fusion*, *Lane post-processing* and *Camera* for the camera input set, and *Fusion*, *Traffic Light process* and *LiDAR* for the LiDAR input set.

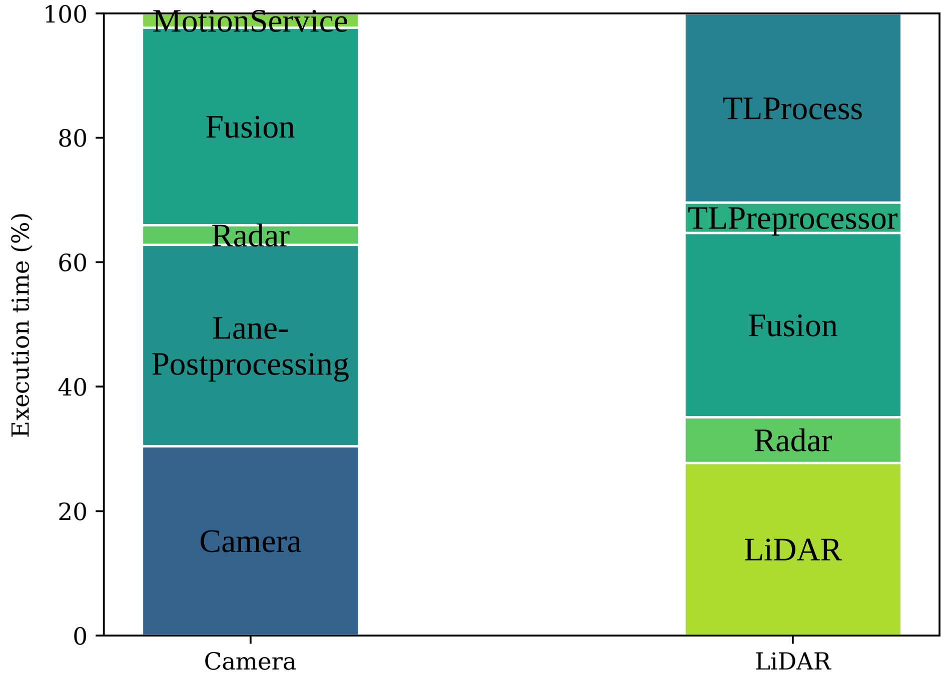
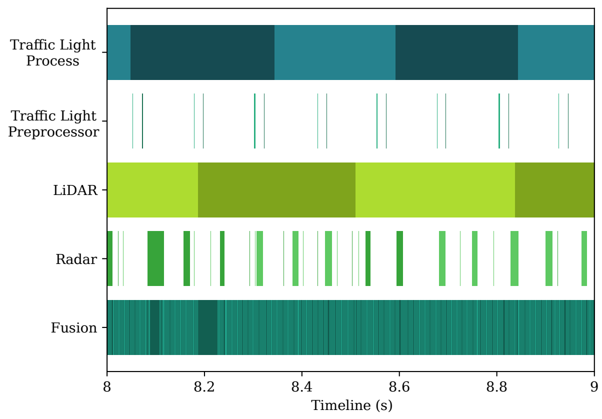
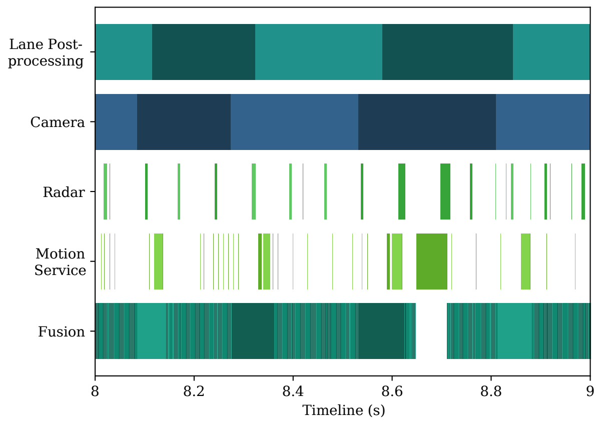


Figure 4.5: Execution time breakdown for both camera and LiDAR configurations. TL stands for Traffic Light.

In order to dig more into this behavior, we analyse the timelines for both input sets. Excerpts of those timelines, obtained with the *En-Route* guidelines, are shown in figure 4.6 for the 8-9 seconds time frame. Different color depth is used to indicate different jobs of the same task (node).



(a) LiDAR



(b) Camera

Figure 4.6: Excerpt timeline of the execution of Perception with both configurations. The x-axis is shown in seconds.

As shown, the different nodes run concurrently on the CPU and GPU. In particular, the three nodes dominating the execution time for each input set (see figure 4.5) run almost continuously starting a new job almost immediately after finishing the previous one. Instead, two other nodes (*Radar* is one such node in both input sets) do not run most of the time, having some significant time elapsed between the end of one job and the start of the following one. This information can be retrieved since *En-Route* allows collecting start and end times for each node and function, thus allowing to build both the timelines and the execution time breakdowns.

For GPU Platforms

In order to illustrate the results of *En-Route* to test execution times on the GPU, we report in table 4.1 the execution time of each CUDA kernel for the *LiDAR process* node. *En-Route* provides kernel execution times for each individual CUDA kernel of the node. This allows summarizing the data in the way shown in the table, where we report for each CUDA kernel the fraction of time devoted to that kernel w.r.t. the total time devoted to all kernels, the absolute accumulated execution time, the number of kernel calls, as well as the average, minimum and maximum execution time for each kernel. For instance, in the second row we find the results for function `sgemm_32x32x32_NN_vec`, intended to perform matrix multiplications. We see that this function takes 27.08% of the overall execution time spent by the CUDA kernels on the GPU, with a total of 368 calls, taking 411μs on average, for a total of 151ms.

We also collected execution times on the GPU of *Camera process* and *Traffic light process* nodes,

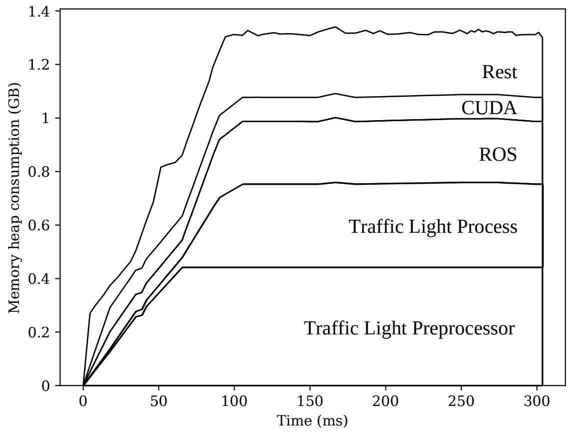
Time (%)	Time	Calls	Avg	Min	Max	Name
27.08	151.14ms	368	410.72μs	32.257μs	1.5219ms	sgemm_32x32x32_NN_vec
19.17	107.01ms	437	244.87μs	8.4160μs	1.1276ms	void caffe::im2col_gpu_kernel
13.16	73.466ms	653	112.51μs	864ns	1.9801ms	[CUDA memcpy HtoD]
9.99	55.766ms	115	484.92μs	242.70μs	1.2116ms	[CUDA memcpy DtoH]
9.26	51.662ms	115	449.23μs	41.730μs	895.69μs	void caffe::col2im_gpu_kernel
5.84	32.582ms	575	56.663μs	2.8160μs	245.80μs	void gemmk1_kernel
4.63	25.854ms	552	46.836μs	2.0800μs	217.77μs	void caffe::ReLUForward
4.14	23.112ms	115	200.97μs	63.395μs	354.90μs	sgemm_128x128x8_TN_vec
3.47	19.385ms	92	210.71μs	125.13μs	346.09μs	maxwell_sgemm_128x64_raggedMn_nn_splitK
1.56	8.7246ms	184	47.416μs	6.4320μs	124.04μs	void caffe::Concat
0.95	5.2876ms	184	28.736μs	12.320μs	78.563μs	void caffe::Slice
0.68	3.8039ms	69	55.129μs	24.929μs	114.66μs	void caffe::SigmoidForward
0.05	299.37μs	23	13.016μs	12.640μs	13.985μs	void caffe::mul_kernel
0.02	103.38μs	157	658ns	370ns	1.3520μs	[CUDA memset]

Table 4.1: CUDA kernels executed by *LiDAR process* node.

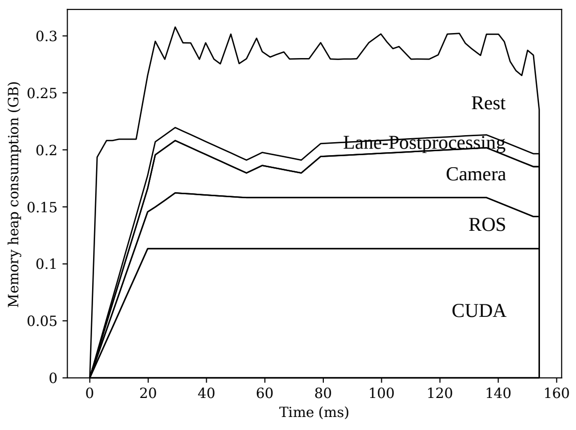
which are shown in tables 4.2 and 4.3, respectively. As expected, there can be seen several kernels from neural networks such as `caffe::ReLUForward`, `caffe::ScaleBiasForward` or `caffe::MaxPoolForward` in all tables, since the three nodes use them. Moreover, the most time-consuming kernels of Perception are related to matrix multiplications (`gemm` stands for General Matrix Multiply), followed by `caffe::im2col_gpu_kernel`, which rearranges discrete image blocks into columns. Both are commonly used in the image processing domain, and by neural networks too.

4.4.2 Memory Usage Tests

For CPU Platforms



(a) LiDAR



(b) Camera

Figure 4.7: Memory usage over time for the provided bag files and configurations.

Time(%)	Calls	Avg	Name
22.11	4455	285.44μs	maxwell_sgemm_128x64_raggedMn_nn_splitK
15.06	21600	40.111μs	caffe::im2col_gpu_kernel
14.24	19170	42.732μs	sgemm_32x32x32_NN_vec
10.75	622	994.41μs	[CUDA memcpy DtoH]
6.74	7710	50.296μs	[CUDA memcpy HtoD]
5.60	810	397.39μs	caffe::col2im_zgpu_kernel
4.32	17550	14.153μs	gemmk1_kernel
2.78	23625	6.7620us	caffe::ReLUForward
2.24	6885	18.699μs	[CUDA memcpy DtoD]
2.03	3105	37.561μs	caffe::ScaleBiasForward
1.88	270	400.74μs	sgemm_32x32x32_NN
1.56	135	665.44μs	sgemm_128x128x8_TN_vec
1.49	135	635.60us	gemmSN_NN_kernel
1.33	945	80.781μs	caffe::PermuteKernel
1.32	3105	24.371μs	caffe::div_kernel
1.06	9990	6.1050us	caffe::Concat
1.01	1350	42.850us	caffe::MaxPoolForward
0.76	31860	1.3690us	axpy_kernel_val
0.43	405	61.110us	caffe::kernel_channel_div
0.38	405	54.119μs	caffe::kernel_channel_subtract
0.37	270	79.064μs	sgemm_32x32x32_TN_vec
0.33	135	139.38μs	thrust::system::cuda::detail::bulk_::detail::launch_by_value
0.30	405	42.425μs	caffe::kernel_exp
0.24	6480	2.1730us	scal_kernel_val
0.23	8505	1.5780us	caffe::Slice
0.19	405	27.239μs	caffe::kernel_channel_max
0.19	405	27.091μs	caffe::kernel_channel_sum
0.18	405	25.707μs	maxwell_sgemm_128x64_raggedMn_nt_splitK
0.18	6210	1.6340us	copy_kernel
0.17	270	37.049μs	apollo::perception::resize_linear_kernel
0.16	21563	419ns	[CUDA memset]
0.15	3105	2.6910us	caffe::powx_kernel
0.08	135	34.784μs	apollo::perception::get_object_kernel
0.08	3105	1.4520us	caffe::add_scalar_kernel
0.04	135	17.092μs	thrust::system::cuda::detail::bulk_::detail::launch_by_value
0.01	26	23.938μs	caffe::ROIPoolForward
0.01	135	3.0720us	caffe::SigmoidForward
0.00	26	3.3440us	apollo::perception::compute_overlapped_by_idx_kernel

Table 4.2: CUDA kernels executed by *Camera process* node.

Time (%)	Time	Calls	Avg	Name
47.23	4.6013ms	708	6.4990us	[CUDA memcpy HtoD]
8.40	818.78μs	392	2.0880us	void gemmk1_kernel
8.40	817.99μs	83	9.8550us	sgemm_32x32x32_NN_vec
5.07	493.48μs	801	616ns	[CUDA memset]
3.96	386.06μs	226	1.7080us	[CUDA memcpy DtoD]
2.71	264.49μs	43	6.1500us	void caffe::im2col_gpu_kernel
2.67	259.72μs	81	3.2060us	void caffe::ScaleBiasForward
2.45	238.54μs	174	1.3700us	void copy_kernel
2.28	221.67μs	182	1.2170us	void scal_kernel_val
2.06	200.55μs	2	100.28μs	sgemm_128x128x8_TN_vec
2.05	200.04μs	87	2.2990us	void caffe::powx_kernel
1.84	179.11μs	87	2.0580us	void caffe::div_kernel
1.55	151.07μs	89	1.6970us	void caffe::ReLUForward
1.31	127.53μs	5	25.505μs	maxwell_sgemm_128x64_raggedMn_nn_splitK
1.18	115.24μs	87	1.3240us	void caffe::add_scalar_kernel
1.08	105.67μs	6	17.611μs	sgemm_32x32x32_NN
1.01	97.921μs	42	2.3310us	void caffe::Concat
0.93	90.562μs	32	2.8300us	void axpy_kernel_val
0.82	80.324μs	2	40.162μs	void caffe::col2im_gpu_kernel
0.75	72.964μs	6	12.160us	[CUDA memcpy DtoH]
0.47	45.889μs	2	22.944μs	void caffe::PSROIPoolingForward
0.46	45.218μs	7	6.4590us	void caffe::MaxPoolForward
0.39	38.082μs	3	12.694μs	void caffe::AvePoolForward
0.19	18.593μs	2	9.2960us	void caffe::kernel_CropBlob
0.15	14.752μs	6	2.4580us	void caffe::ScaleForward
0.10	9.6650us	1	9.6650us	void caffe::kernel_ResizeBlob
0.09	8.4480us	3	2.8160us	void caffe::kernel_channel_max
0.09	8.3840us	3	2.7940us	void caffe::kernel_channel_sum
0.08	8.0970us	3	2.6990us	void caffe::kernel_channel_div
0.07	7.1680us	3	2.3890us	void caffe::kernel_channel_subtract
0.06	5.7920us	2	2.8960us	void gemv2T_kernel_val
0.06	5.4410us	3	1.8130us	void caffe::kernel_exp
0.03	3.2960us	2	1.6480us	void caffe::set_kernel

Table 4.3: CUDA kernels executed by *Traffic Light process* node.

Finally, we show the type of results that *En-Route* provides in terms of memory usage. We obtain memory usage per node or per function, and also the memory requirements over time, as depicted in figure 4.7 for the CPU for input data using either LiDAR or camera configurations. As figure 4.7a shows, CPU memory usage grows up to 1.3 GB in around 100 ms. Such memory usage remains quite constant over time for the remaining 200 ms of execution, but also after that point until the end of the execution. Similarly, as figure 4.7b shows, CPU memory usage grows up to 300 MB in around 20 ms and remains constant for the rest of the execution.

The main reason that the memory usages for the two configurations are different is related to the design of the AD software. For instance, both LiDAR and camera configurations use neural networks with different architectures and, therefore, have different memory usages.

In both cases, *En-Route* allows assessing how much memory is used by each different node, thus facilitating the validation process. For instance, we observe that CUDA support requires around 100MB for both configurations (note the different scale of the plots).

For GPU Platforms

As explained in section 4.2.2, to obtain memory usage information from the GPU, *En-Route* builds upon *GDB* and *nvidia-smi*, and requires user intervention. For this reason, *En-Route* is not able to collect at which time the program uses or frees memory, but it allows to measure the amount of memory allocated at different points of the execution, at the code level. Therefore, the results we got with *En-Route* regarding the GPU memory usage need code knowledge to be fully understood.

However, for both configurations, we found the execution points where data was allocated in memory, and how much of it occupied. In both cases, we did not detect data that was freed from memory until the end of the program. Moreover, the GPU's memory usage of Perception is directly related to the different neural networks it is using and, hence, with Caffe. In particular, Perception running with the camera configuration reached the peak of 2975MB in 7 steps (183MB, 2MB, 6MB, 1504MB, 6MB, 4MB and 1270MB), while with the LiDAR configuration occupied 2295MB in 5 steps (183MB, 6MB, 6MB, 200MB and 1900MB).

4.4.3 Summary

Overall, as shown, *En-Route* allows collecting detailed information in terms of resource usage for complex AD frameworks. We have shown that results allow assessing both, total resource usage as well as resource usage over time, thus enabling a wide variety of assessments for the V&V of AD frameworks.

Chapter 5

Related Work

The timing validation of automotive systems has been customarily based on the combination of dynamic measurements with a system-level timing model [13,49], often extended to capture CAN or network-based communication between engine control units [48]. Some works have also reported on industrial experience in applying STA to automotive software [32,37]. These works, however, focus on the timing characterisation 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 [82] advocates stochastic analysis for the characterisation of end-to-end latencies, thus focusing on system-level aspects rather than timing characterisation. Meanwhile, for the localization module, authors in [43] report large execution time variability, but do not analyse it as we have done in this Thesis for Apollo modules and stages of the *Camera process* node.

In [54], 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 characterise the timing variability and analyse the timing behavior of an entire AD framework, far beyond the sole Perception module.

Regarding resource usage tests, they have been often regarded as an engineering problem, being the main challenge how to create stressful tests. Moreover, since those tests are neither needed for the design of the system itself, nor for the validation of the safety requirements, no explicit safety requirements need to be fulfilled by those tests. Still, tool qualification may be required in accordance with ISO 26262 part 8 [33], since those tools are part of the development of safety-related elements. However, the advent of AD frameworks with software constructs far more complex than those used so far in automotive systems, challenge current resource usage testing practice, thus calling for new solutions. This Thesis tackles this challenge by presenting *En-Route*, a set of guidelines able to handle the complexity of AD frameworks to perform a wide variety of resource usage tests.

Tools for CPU profiling, such as *Valgrind*, *Google Performance Tools* (GPT) or *Perf*, pose also a number of limitations to resource usage testing since they also clash with the dynamic behavior of AD frameworks. Thus, new solutions are needed matching their needs.

Tools for GPU profiling, such as the NVIDIA Visual Profiler and *nvprof* [51], pose a number of limitations related to the dynamic behavior of AD frameworks and the fact that they are intended to run continuously, thus never ending the profiling process. Therefore, appropriate utilization of these tools is needed, as performed by *En-Route*.

The set of guidelines proposed in this Thesis, *En-Route*, overcomes these limitations. While we identified that some additional tool support for an enhanced automation of the process would be convenient, the solutions we provided already enable resource usage testing for complex AD frameworks, subject to the qualification of these tools (or equivalent ones) to fully adhere to the requirements of a safety-related development.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The slant towards AD 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 Thesis, we present an analysis of the timing variability of Apollo, an industrial-quality AD framework, showing that Apollo modules and nodes exhibit a highly variable timing behavior and arbitrary distributions. We analyse 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 analysing the execution time of Apollo modules and single stages of the *Camera process* of the Perception module when running on a real board.

Furthermore, the advent of AD frameworks also challenges current practice to perform resource usage tests due to the complexity of those software frameworks and the hardware platforms that need to be used, which include CPUs and GPUs. Those tests are a requirement for safety-related automotive systems, as indicated in ISO 26262. In this Thesis, we present *En-Route*, a set of remedies and guidelines to enable resource usage testing on complex AD frameworks. We assess *En-Route* with Apollo, illustrating the main difficulties to use existing tools and how those difficulties can be defeated, leading to a wide variety of results for execution time and memory requirements. In particular, those results allow breaking down resource usage across functions and assessing usage over time, thus facilitating the duties of system integrators to validate that resource usage is within expected bounds.

While our timing analysis and *En-Route* guidelines are applied on Apollo, findings of this applied research work can be naturally extended to other AD frameworks (e.g. Autoware) or analogous frameworks in other domains (e.g. in the robotics domain).

6.2 Future Work

On the one hand, we presented the first steps towards a complete and efficient timing analysis solution for AD software frameworks like Apollo, which means it will still require long-term efforts by the community to achieve such an overwhelming objective.

On the other hand, although we were able to enable resource usage testing on Apollo, we faced several problems with existing tools, which led to rely on engineering work, together with our knowledge of the framework, to achieve it. Basically, we found no tool providing appropriate support in this domain. Thus, as we said in section 4.2.1, there is a business opportunity for software vendors to develop proper tools for resource usage testing of complex CPU code.

Chapter 7

Publications

Based on the work done in this Thesis, two papers has been published under the following format:

- **En-Route: on enabling resource usage testing for autonomous driving frameworks [5].** Miguel Alcon, Hamid Tabani, Jaume Abella, Leonidas Kosmidis, and Francisco J. Cazorla. 2020. In Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20), Brno, Czech Republic, March 30 - April 3, 2020.
- **Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions [6].** Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella and Francisco J. Cazorla. 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Sydney, Australia, 21-24 April 2020.

Bibliography

[1] 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.

[2] J. Abella et al. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.

[3] 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.

[4] Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. Safety-related challenges and opportunities for gpus in the automotive domain. *IEEE Micro*, 38(6):46–55, 2018.

[5] Miguel Alcon, Hamid Tabani, Jaume Abella, Leonidas Kosmidis, and Francisco J. Cazorla. En-Route: On Enabling Resource Usage Testing for Autonomous Driving Frameworks. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 1953–1962, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco 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)*, pages 267–280, 2020.

[7] Amazon Web Service. ADAS and Autonomous Driving. `https://aws.amazon.com/automotive/autonomous-driving/`, 2020.

[8] Apollo. Apollo Hardware Development Platform. `https://apollo.auto/platform/hardware.html`, 2020.

[9] ApolloAuto. Apollo 3.0 Software Architecture. `https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_3.0_Software_Architecture.md`, 2018.

[10] ApolloAuto. How to do performance profiling. `https://github.com/ApolloAuto/apollo/blob/r3.0.0/docs/howto/how_to_do_performance_profiling.md`, 2018.

[11] ApolloAuto. Perception. `https://github.com/ApolloAuto/apollo/blob/r3.0.0/docs/specs/perception_apollo_3.0.md`, 2018.

[12] 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.

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

[14] Baidu. Apollo, an open autonomous driving platform. `http://apollo.auto/`, 2018.

[15] Baidu. How coronavirus is accelerating a future with autonomous vehicles. `https://www.technologyreview.com/2020/05/18/1001760/how-coronavirus-is-accelerating-autonomous-vehicles/`, 2020.

[16] 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.

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

[18] 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.

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

[20] S. Chattopadhyay et al. A unified WCET analysis framework for multi-core platforms. In *RTAS*, 2012.

[21] 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.

[22] 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.

[23] Daimler. Autonomous Driving. `https://www.daimler.com/innovation/product-innovation/autonomous-driving/`, 2020.

[24] 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.

[25] GDB Developers. Gdb: The gnu project debugger. `https://www.gnu.org/software/gdb/`, 2017.

[26] GDB Developers. Gdb: The gnu project debugger. `https://developer.nvidia.com/nvidia-system-management-interface`, 2019.

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

[28] J. Redmon et al. Yolo9000: better, faster, stronger. *arXiv preprint*, 2017.

[29] 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.

[30] The Autoware Foundation. Autoware. An open autonomous driving platform. `https://github.com/CPFL/Autoware/`, 2016.

[31] A. Furda and L. Vlacic. Enabling safe autonomous driving in real-world city traffic using multiple criteria decision making. *IEEE Intelligent Transportation Systems Magazine*, 3(1):4–17, 2011.

[32] 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.

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

[34] International Organization for Standardization. *ISO/IEC/IEEE 24765. Systems and software engineering – Vocabulary*, 2017.

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

[36] Y. Jia et al. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[37] 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.

[38] R. Kirner and P. Puschner. Classification of WCET analysis techniques. In *ISORC*, 2005.

[39] 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, 2008.

[40] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

[41] H. Li et al. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In *RTNS*, 2014.

[42] Velodyne Lidar. Velodyne Lidar. `https://velodynelidar.com/`, 2020.

[43] 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.

[44] Markus Maibach, Christoph Schreyer, Daniel Sutter, HP Van Essen, BH Boon, Richard Smokers, Arno Schroten, C Doll, Barbara Pawlowska, and Monika Bak. Handbook on estimation of external costs in the transport sector. *Ce Delft*, 336, 2008.

[45] Fabio Mazzocchetti, Pedro Benedicte, Hamid Tabani, Leonidas Kosmidis, Jaume Abella, and Francisco J Cazorla. Performance analysis and optimization of automotive gpus. In *Proceedings of the 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) 2019*. IEEE, 2019.

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

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

[48] 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.

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

[50] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

[51] NVIDIA. Profiler :: CUDA toolkit documentation. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview`, 2019.

[52] NVIDIA. Driving Innovation. `https://www.nvidia.com/en-us/self-driving-cars/`, 2020.

[53] World Health Organization. Road traffic injuries. `https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries`, 2020.

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

[55] European Parliament. Co2 emissions from cars: facts and figures (infographics). `https://www.europarl.europa.eu/news/en/headlines/society/20190313STO31218/co2-emissions-from-cars-facts-and-figures-infographics`, 2019.

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

[57] Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[58] 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.

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

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

[61] D. Shapiro. Introducing xavier, the nvidia ai supercomputer for the future of autonomous transportation. *NVIDIA blog*, 2016.

[62] Google Open Source. gflags. `https://opensource.google.com/projects/gflags`, 2019.

[63] Google Open Source. google-glog: Application Level Logging . `https://opensource.googleblog.com/2008/10/google-glog-application-level-logging.html`, 2019.

[64] Google Open Source. Google Performance Tools. `https://github.com/gperftools/gperftools/wiki`, 2019.

[65] J. Souyris et al. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *WCET workshop*, 2007.

[66] 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.

[67] Synced. Beijing self-driving vehicle road tests topped one million km in 2019. `https://syncedreview.com/2020/03/14/beijing-self-driving-vehicle-road-tests-topped-one-million-km-in-2019/`, 2020.

[68] Rapita Systems. On-target software verification solutions. `www.rapitasystems.com`, 2020.

[69] Hamid Tabani, Matteo Fusi, Leonidas Kosmidis, Jaume Abella, and Francisco J Cazorla. Intpred: flexible, fast, and accurate object detection for autonomous driving systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 564–571, 2020.

[70] Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Francisco J Cazorla, and Guillem Bernat. Assessing the adherence of an industrial autonomous driving framework to iso 26262 software guidelines. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 9. ACM, 2019.

[71] Araz Taeihagh and Hazel Si Min Lim. Governing autonomous vehicles: emerging responses for safety, liability, privacy, cybersecurity, and industry risks. *CoRR*, abs/1807.05720, 2018.

[72] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 303–314, New York, NY, USA, 2018. Association for Computing Machinery.

[73] Toyota Motor Corporation. Toyota News Release. `https://pressroom.toyota.com/toyota-establish-artificial-intelligence-research-development-company/`, 2015.

[74] Valgrind Developers. Massif: a heap profiler. `http://valgrind.org/docs/manual/ms-manual.html`, 2019.

[75] Valgrind Developers. Valgrind. `http://valgrind.org/`, 2019.

[76] Sergi Vilardell, Isabel Serra, Hamid Tabani, Jaume Abella, Joan Del Castillo, and Francisco J Cazorla. Cleanet: enabling timing validation for complex automotive systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 554–563, 2020.

[77] Waymo. Waymo. `https://waymo.com/`, 2020.

[78] WikiChip. Tegra xavier - nvidia. `https://en.wikichip.org/wiki/nvidia/tegra/xavier#Board`, 2019.

[79] 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.

[80] 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.

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

[82] 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.