The final publication is available at ACM via: https://doi.org/10.1145/3341105.3373938

En-Route: On Enabling Resource Usage Testing for Autonomous Driving Frameworks

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

***Abstract*—Software resource usage testing, including execution time bounds and memory, is a mandatory validation step during the integration of safety-related real-time systems. However, the inherent complexity of Autonomous Driving (AD) systems challenges current practice for resource usage testing. This paper exposes the difficulties to perform resource usage testing for AD frameworks by analyzing a complex and critical module of an AD framework, and 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.**

I. Introduction

Automotive safety-related systems must undergo a development process with exhaustive verification and validation steps, where each item is proven to adhere to its safety requirements with the degree of rigor dictated by safety standards [11]. In safety-related real-time systems, 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) [30], [1], [14], [13], [26], [7], [17], [15]. 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. 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 verification and validation [21]. These techniques rely on the ability to collect information on the execution of the tasks under analysis. This is challenged by forthcoming Autonomous Driving (AD) systems, increasingly considered for adoption by automotive industry. 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. This is, for instance, the case for Baidu's Apollo AD framework, the largest AD project with more than 120 OEMs, Tier1, Tier2, AI and tech companies, and car manufacturers [6]. On the other hand, however, complex software constructs 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.

This paper addresses this challenge by 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 [11]. In particular, our contributions are the following:

1) An analysis of the difficulties and roadblocks to collect timing and memory utilization measurements of an AD software framework, using Apollo framework in general, and its Perception module in particular, as a representative software module for guiding the discussion.
2) 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.
3) An 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.

Although the work in this paper is applied to Apollo as an illustrative example, results and conclusions apply to other AD frameworks as well as autonomous systems in domains such as robotics, since challenges posed by Apollo relate to, first, its integration with the Robotics Operating System (ROS) [20], and second, to the use of abstractions to ease software maintainability such as Docker containers. Both such characteristics are common across autonomous system frameworks in general, and AD frameworks in particular (e.g. Autoware [10]).

The rest of the paper is organized as follows. Section II provides background on the safety-related software development process and on the Apollo AD framework. Section III analyzes the difficulties and roadblocks to perform resource usage testing on AD frameworks in general, building on Apollo as an

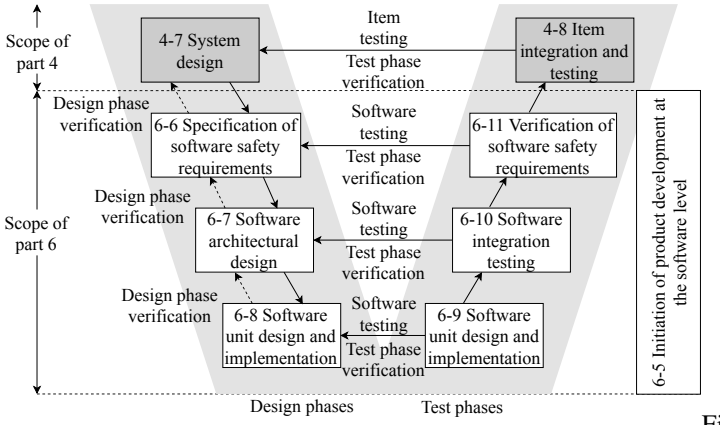


Fig. 1: Software development process as described in ISO 26262 (picture taken from ISO 26262 Part 6 [11]).

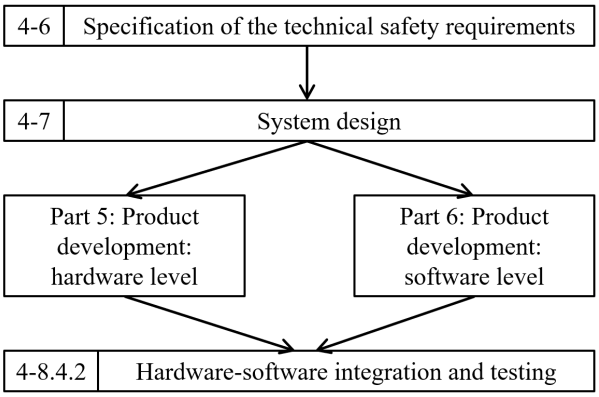


Fig. 2: System development process as described in ISO 26262 (picture elaborated from ISO 26262 Part 4 [11]).

example. Section IV presents *En-Route* guidelines for resource usage testing of AD frameworks. *En-Route* is then evaluated in Section V. Finally, Section VI provides some related work, and Section VII concludes this paper.

II. Background

In this section, we provide some background on the development process of automotive systems as stipulated in ISO 26262 [11] safety standard, with emphasis on the software part, as well as on Apollo AD framework [6], [27], [2].

A. 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 are identified as well as safety requirements 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, verification and validation 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 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 verification and validation 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 verification and validation phase[1].

System-wise, see Figure 2, after the specification and system design, hardware and software product development occurs, where software development is as shown in Figure 1. Software and hardware items are then integrated to form a subsystem and, as indicated before, some testing is performed. 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.

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

[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".

		PLANNED					MEASURED				
ITEM	TASK PERIOD	CPU	DFLASH	PFLASH0	PFLASH1	RAM	CPU	DFLASH	PFLASH0	PFLASH1	RAM
Pos mngmt	10ms	3%	280	120	0	80	4%	308	96	0	104
Angle	5ms	9%	36	0	768	24	9%	36	0	614	24
Torque monitoring	40ms	1%	16	16	0	0	1%	11	21	0	0
Accel monitoring	40ms	1%	16	16	0	0	1%	18	10	0	0
Power mode	2ms	4%	46	376	0	240	2%	41	414	0	336
Torque_CTRL1	20ms	29%	2240	0	880	540	26%	3360	0	440	486
Torque_CTRL2	20ms	28%	2048	0	860	512	14%	2253	0	516	563

Fig. 3: Example of output of resource usage tests. Memory occupancy is given in KBs.

tables such as that in Figure 3. In particular, for different software items of a hypothetical combustion engine, Figure 3 shows the measured and budgeted (planned) CPU and memory usage (DFLASH, PFLASH, and, RAM) in an Infineon AURIX CPU.

B. Apollo AD Framework

Apollo [6] is an open software autonomous driving platform released by Baidu. 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 [3], [19] (see Figure 4). Each of the modules implements a crucial functionality of autonomous vehicles. The main modules of Apollo are:

- **Perception:** identifies the area surrounding the autonomous vehicle 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 autonomous vehicle is located, using various information sources such as GPS, LiDAR and IMU. State-of-the-art localization algorithms, including the one in Apollo, are capable of localizing the position of the vehicle at centimeter-level accuracy.
- **Prediction:** anticipates the future motion trajectories of the perceived obstacles.
- **Routing:** tells the autonomous vehicle how to reach its destination via a series of lanes or roads.
- **Planning:** plans the spatiotemporal trajectory for the autonomous vehicle to take.
- **Control:** executes the planned spatiotemporal trajectory by generating control commands such as accelerate, brake, and steering.
- **CanBus:** is the interface that passes control commands to the vehicle hardware. It also passes chassis information to the software system.
- **HD-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.

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 5. As it can be seen, Apollo uses three libraries for different purposes:

- The Google Logging Library (glog) [24], 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) [23], 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) [20] is a set of software libraries and tools that help building robot applications. 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

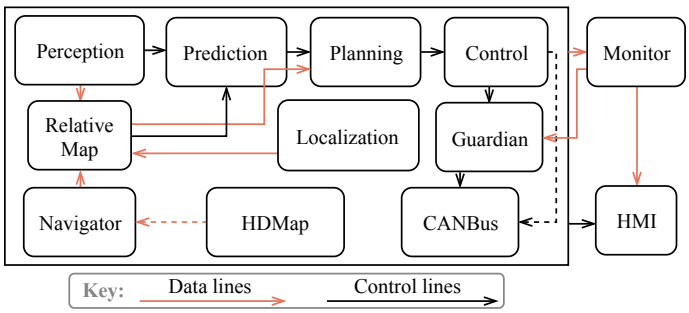


Fig. 4: Interaction between Apollo's modules.

```
#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;
}
```

Fig. 5: Main function of an *ApolloApp*

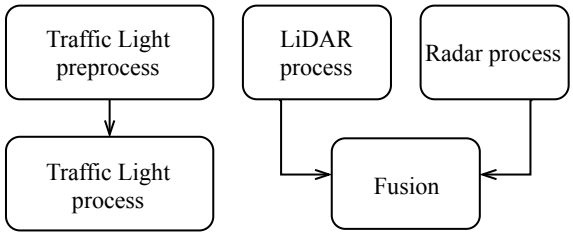


Fig. 6: DAG of the LiDAR configuration.

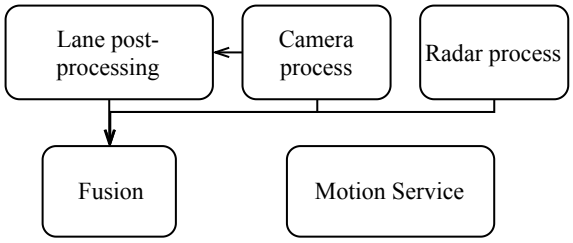


Fig. 7: DAG of the camera configuration.

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.

Perception module. The Perception module [5] 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 Figures 6 and 7, as they are the ones that we could execute with the data (ROS bag[2] files) that Apollo provides.

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 7, *Fusion* requires the output data of *Lane post-processing*, *Camera process*, and *Radar process*.

Beyond Apollo. In this paper we study Apollo as a representative and well-known AD framework. There are other well-known AD systems such as Autoware [10] with similar software architecture design, using ROS and Docker containers, thus facing similar challenges to the ones explored in this paper. ROS is a popular operating system for autonomous frameworks, and it 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 paper. In this paper, we focus on Apollo without lack of generality, and our contributions and findings can be naturally extended to other domains and frameworks.

[2]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.

III. Roadblocks for Resource Usage Testing on AD 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 [16]. This is the case for Apollo in general, and its Perception module in particular [5], 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.

A. 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* [18] and NVIDIA Visual Profiler which uses *nvprof* for visualizing the profiled information.

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 a Robotics Operating System (ROS) [20] embedded in Apollo. 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. Only when we disabled the option to profile the application from start (`profile-from-start off`) execution progressed as expected. However, 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. 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 [12], 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.

B. 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* [29], *Google Perfor-*

mance Tools (GPT) [25] 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 [4].

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 6), 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 8, 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 print 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.

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.

IV. EN-ROUTE

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

A. En-Route for GPUs

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

4.1.1 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 9 and 10 show an example of how to use them.

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.

4.1.2. 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

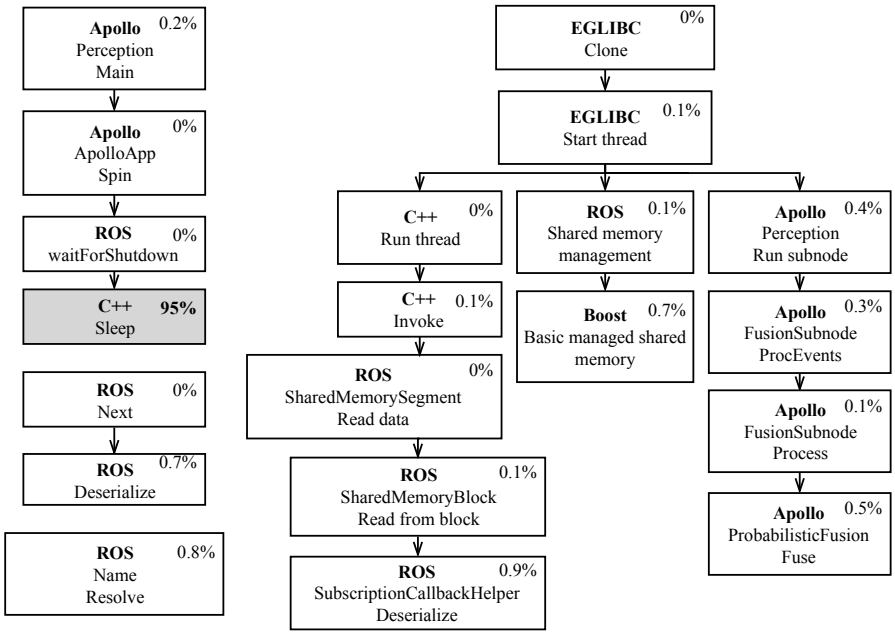


Fig. 8: Call tree of the execution of the Perception module.

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

Fig. 9: Bash command to perform the profiling in selected sections.

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

  // Section you want to profile

  cudaProfilerStop();
  ...
}
```

Fig. 10: Example of C++ code that selects the section of code to be profiled.

of all nodes of Perception (or Apollo if we aim at analyzing the whole framework) 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.

4.1.3. 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 [8] (*GDB*) and the NVIDIA System Management Interface [9] (*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.

B. En-Route for CPUs

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 *Massif* tool (part of *Valgrind*) [28]. In any case, CPU code has low

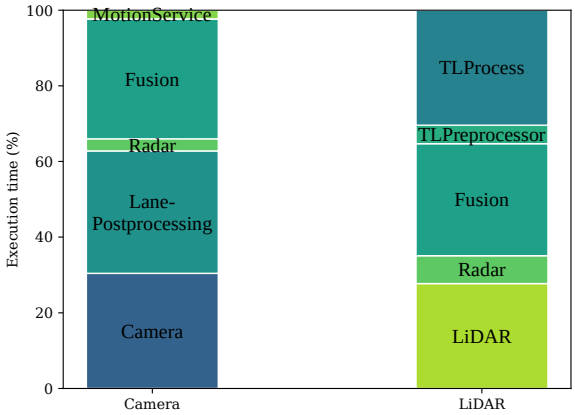


Fig. 11: Execution time breakdown for both camera and LiDAR configurations. TL stands for Traffic Light.

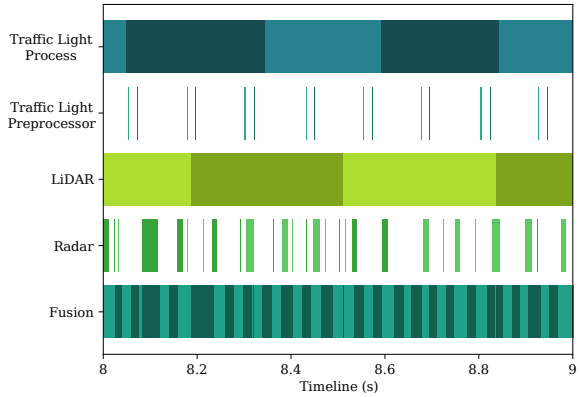


Fig. 12: Excerpt timeline of the execution of Perception with the LiDAR and radar input data. The x-axis is shown in seconds.

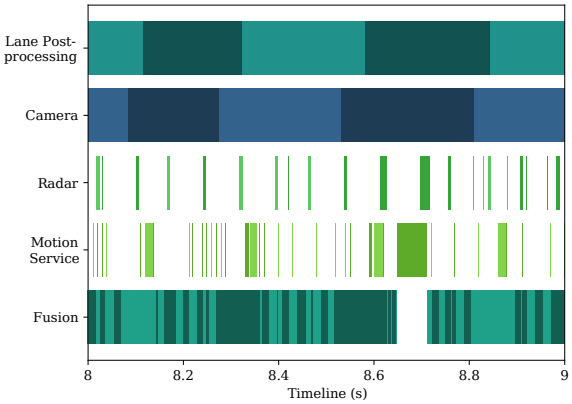


Fig. 13: Excerpt timeline of the execution of Perception with camera and radar input data. The x-axis is shown in seconds.

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.

V. Evaluation

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.

Measurements have been obtained on top of an NVIDIA Jetson Xavier board intended for automotive systems [22]. It includes an 8-core CPU based on Carmel ARM V8 64-bit architecture, and an NVIDIA GPU with 512 CUDA cores based on the Volta architecture.

A. CPU Execution Time Usage Tests

We have collected execution times for the different nodes in Figures 6 and 7. 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 11, 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 analyzed. 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.

In order to dig more into this behavior, we analyze the timelines for both input sets. Excerpts of those timelines, obtained with the *En-Route* guidelines, are shown in Figures 12 and 13 for the 8-9 seconds time frame. Different color depth is used to indicate different jobs of the same task (node).

As shown, the different nodes run concurrently on the CPU and GPU of the NVIDIA Jetson Xavier platform. In particular, the three nodes dominating the execution time for each input set (see Figure 11) 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.

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 I: CUDA kernels executed by *LiDAR process* Sub-node.

B. GPU Execution Time Usage Tests

In order to illustrate the results of *En-Route* to test execution times on the GPU, we report in Table I 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.

C. Memory Usage Tests

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 14 (a) and (b) for the CPU for input data with configurations using LiDAR and camera respectively. As Figure 14 (a) 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 14 (b) 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).

D. 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 verification and validation of AD frameworks.

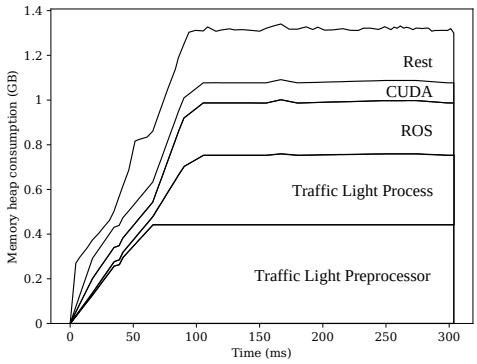
VI. Related Work

Resource usage tests 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 [11], 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 paper 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.

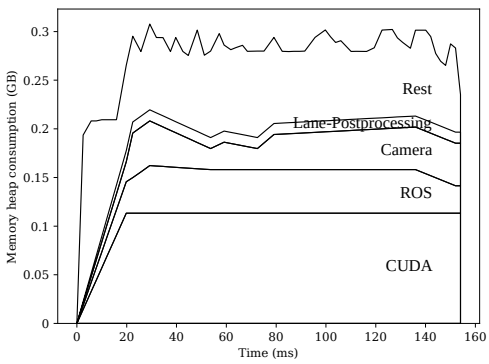
Existing tools for AD frameworks, such as the NVIDIA Visual Profiler and *nvprof* [18], pose a number of limitations related to the dynamic behavior of AD frameworks and the fact that they are intended to run continuously, thus never ending the profiling process. Thus, appropriate utilization of these tools is needed, as performed by *En-Route*.

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 the needs of AD frameworks.

The set of guidelines proposed in this paper, *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 provided in this paper already enable resource usage testing for complex AD frameworks, subject to the qualification of these tools (or



(a) Memory usage for configuration with LiDAR



(b) Memory usage for configuration with camera

Fig. 14: Memory usage over time for the provided bag files.

equivalent ones) to fully adhere to the requirements of a safety-related development.

VII. Conclusions

Resource usage tests are a requirement for safety-related automotive systems, as indicated in ISO 26262. The advent of AD frameworks challenges current practice to perform those tests due to the complexity of those software frameworks and the hardware platforms that need to be used, which include CPUs and GPUs.

In this paper, we present *En-Route*, a set of remedies and guidelines to enable resource usage testing on complex AD frameworks. We assess *En-Route* with the Apollo AD framework, a popular commercial AD framework, 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 *En-Route* is 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).

Acknowledgements

This work has been partially supported by 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. MINECO partially supported Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717) and Leonidas Kosmidis under Juan de la Cierva-Formación postdoctoral fellowship (FJCI-2017-34095).

References

[1] J. Abella et al. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.
[2] 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.
[3] ApolloAuto. Apollo 3.0 Software Architecture, 2018.
[4] ApolloAuto. How to do performance profiling, 2018.
[5] ApolloAuto. Perception, 2018.
[6] Baidu. Apollo, an open autonomous driving platform. http://apollo.auto/, 2018.
[7] S. Chattopadhyay et al. A unified WCET analysis framework for multi-core platforms. In *RTAS*, 2012.
[8] GDB Developers. Gdb: The gnu project debugger, 2017.
[9] GDB Developers. Gdb: The gnu project debugger, 2019.
[10] The Autoware Foundation. Autoware. An open autonomous driving platform. https://github.com/CPFL/Autoware/, 2016.
[11] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[12] Y. Jia et al. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.
[13] R. Kirner and P. Puschner. Classification of WCET analysis techniques. In *ISORC*, 2005.
[14] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.
[15] H. Li et al. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In *RTNS*, 2014.
[16] 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.
[17] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.
[18] NVIDIA. Profiler :: CUDA toolkit documentation, 2019.
[19] 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.
[20] M. Quigley et al. ROS: an open-source Robot Operating System. *ICRA Workshop on Open Source Software*, 2009.
[21] RapiTime. *www.rapitasystems.com*, 2008.
[22] D. Shapiro. Introducing xavier, the nvidia ai supercomputer for the future of autonomous transportation. *NVIDIA blog*, 2016.
[23] Google Open Source. gflags, 2019.
[24] Google Open Source. google-glog: Application Level Logging , 2019.
[25] Google Open Source. Google Performance Tools, 2019.
[26] J. Souyris et al. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *WCET workshop*, 2007.

[27] 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.
[28] Valgrind Developers. Massif: a heap profiler, 2019.
[29] Valgrind Developers. Valgrind, 2019.
[30] 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.