A Multi-Domain Software Architecture for Safe and Secure Autonomous Driving

Luca Belluardo*, Andrea Stevanato*, Daniel Casini*†,
Giorgiomaria Cicero*, Alessandro Biondi*†, and Giorgio Buttazzo*†
*TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
†Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

***Abstract*—This work aims at making Apollo, a popular autonomous driving framework, safer and more secure by designing a *multi-domain architecture*, where its components are split between a feature-rich domain running Linux and a critical domain running a real-time operating system (RTOS). The two domains are isolated by a hypervisor. We implemented a prototype where the control component has been ported from Linux to the Erika automotive-grade RTOS, and we discuss a number of challenges that have been faced in moving the component to Erika. The proposed solution has been experimentally evaluated by measuring the latencies involving processing paths passing through the control component.**

***Index Terms*—multi-domain architecture, autonomous driving, safety, hypervisor**

I. Introduction

In recent years, autonomous driving (AD) and advanced driver-assistance systems (ADASes) attracted increasing attention in the automotive landscape. An autonomous vehicle can sense the surrounding environment thanks to various sensors and move safely with no human intervention.

Implementing AD/ADASes requires dealing with several non-trivial tasks, such as perception, prediction, planning, localization, and control.

These tasks are typically managed by different components within *autonomous driving frameworks*, such as Autoware [1] or Apollo [2], which provide facilities to run them. Furthermore, these frameworks also provide the required inter-component communication mechanisms usually by leveraging, in turn, other middleware layers implementing the publish-subscribe paradigm, such as ROS [3] or CyberRT [4].

AD frameworks typically require a considerable interaction with device drivers, e.g., to acquire data from sensors, and software stacks, e.g., to run inference of deep neural networks on hardware accelerators [5]. As a consequence, a feature-rich OS as Linux is a convenient candidate to run them. On the other hand, Linux is a complex operating system with a potentially large attack surface for security threats and several safety-related issues. On the contrary, AD systems need to ensure safety and security, thus making a trusted real-time operating system a more commendable choice for implementing the most critical functionalities.

A possible solution to take the best of both worlds is to split the AD system into two parts:

- a highly-critical one, comprising functionalities such as the computation of control actions, supervision of the vehicle navigation, fall-back control, and CAN-based communication, which are highly safety-related and do not generally require interaction with complex software stacks and device drivers; and
- a less critical one that generally involves components that are limitedly relevant for safety and not suitable for certification. For example, the latter category may include perception tasks based on complex convolutional neural networks [6], which are very hard to certify [7].

Each part of the system can then run on the most convenient operating system for its purpose as part of a separate domain, i.e., virtual machine (VM), managed by a hypervisor, thus realizing a *multi-domain architecture*.

In this paper, we focus on the Apollo framework [2] (version 5.0) for AD developed by Baidu [8]. Apollo runs on Linux, which provides a large number of device drivers and software stacks that would not be available on an RTOS. On the other hand, it exposes Apollo to security and safety threats.

Contribution. This paper presents the design and implementation of a prototype multi-domain architecture for Apollo. To this end, a deep analysis of the Apollo framework has been conducted to individuate the most safety-related components to be moved on the Erika RTOS [9], an OSEK/VDX certified RTOS used by various companies in the automotive market, including Magneti Marelli PowerTrain and Cobra Automotive Technologies [10]. In the proposed solution, Erika and Linux run into two different domains. To the end of realizing a technology demonstrator, the KVM hypervisor is used to run the domains upon the same platform, orchestrating the accesses to the hardware components. In this way, Apollo can leverage the benefits provided by the two operating systems: for example, it can run the perception component on Linux, benefiting from the pre-existing NVIDIA software stack for accelerating deep networks on Graphics Processing Units (GPUs), and the control component (i.e., the one performing actuation) on Erika with a higher degree of safety and security. However, porting an Apollo component from Linux to Erika requires dealing with several not-trivial shortcomings that are discussed in this paper. Furthermore, the communication between the components running on Linux and Erika needs to be restored, involving the adoption of a proper inter-VM communication mechanism provided by the hypervisor.

Paper Structure. The rest of this paper is organized as follows. Section II provides the needed background. Section III presents the proposed multi-domain architecture. Section IV reports on the main challenges in implementing a multi-domain version of Apollo, discussing the solutions adopted in our prototype implementation. Section V illustrates our experimental evaluation. Section VI discusses the related work. Finally, Section VII concludes the paper and discusses the future work.

II. Preliminaries

This section reviews the three pillars required by our multi-domain architecture, namely, the Apollo AD framework (Section II-A), the Erika RTOS (Section II-B), and the KVM hypervisor with the `IVSHMEM` mechanism to provide inter-VM communication (Section II-C).

A. Apollo

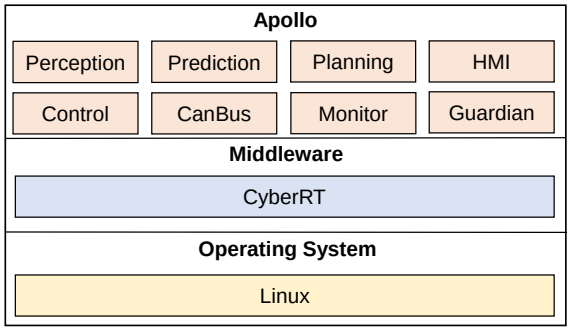


Fig. 1: The Apollo software architecture.

Together with Autoware [1], Apollo is one of the most popular open-source frameworks for autonomous driving. It covers all the functionalities of an autonomous car, from the perception of the surrounding environment to the decisions to take for actually leading the passengers to the destination. It leverages multiple sensors and actuators installed on the vehicle. The Apollo code can be found on a GitHub open repository [11]. The project was started in 2017 by Baidu, and it is the only company to obtain the first batch of road test licenses in China. The software architecture of an Apollo-based system is shown in Fig. 1.

Apollo officially supports Intel x86 platforms only and runs on `PREEMPT_RT` Linux, a variant of Linux which allows achieving under-millisecond scheduling latencies in most cases [12], and, in particular, on Docker. Docker is a set of "platform as a service" products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files.

Above Linux, the Apollo ecosystem provides CyberRT. CyberRT is a framework that manages Apollo's modules and let them communicate using the publish/subscribe (pub/sub) paradigm. Communication channels are called *cyber-channels* or *topics*. Under the pub/sub paradigm, some software components act as *subscribers*, "subscribing" to topics and getting notified when other computational activities, called *publishers*, "publish"' on one of them.

CyberRT was not present in the first versions of Apollo: up to version 3.0, Apollo used a customized version of ROS, which included an optimized communication based on shared memory. From Apollo 3.5, the pub/sub communication has been managed by CyberRT, still developed by Baidu, to better implement the specific requirements individuated by Apollo's developers.

On top of CyberRT, Apollo implements several software components. The most relevant ones are localization, perception, prediction, CAN bus, planning, control, and the human-machine interface (HMI), discussed below:

- *Localization:* provides the localization services, based on GPS, IMU, and LiDAR sensors;
- *Planning:* provides functionalities to plan the path towards the destination;
- *Perception:* allows detecting and classifying obstacles, as well as determine the status of traffic lights;
- *Prediction:* tries to predict the future trajectories of the obstacles detected by the perception module;
- *CAN bus:* forwards the control commands and receives the car's chassis status to the control module;
- *Control:* generates the actuation command based on the data received from the other modules;
- *HMI:* is a module for viewing the status and controlling the functioning of the vehicle.

The Apollo architecture is designed to run on a car conforming with the Apollo's *reference vehicle platform*, which specifies the compatible sensors and the requirements for the industrial computer that needs to be stored inside the trunk.

From a safety point of view, it is interesting to note also the presence of the *monitor* and *guardian* components. The two modules act in synergy. The first one checks the status of the running modules, the data integrity, and the system health (in terms of CPU, memory, and disk usage). If it detects a failure, it informs the guardian that is in charge of performing an action to bring the car to a safe state.

B. Erika RTOS

Erika Enterprise [9] is an RTOS developed by Evidence Srl [13]. Erika is certified OSEK/VDX, a standard for an open-ended architecture for distributed control units in vehicles. Erika has multiple versions, and the newest is version 3.

It provides several excellent features to foster predictability, and hence safety, such as a fixed-priority scheduler, the support for synchronization protocols, and a rich API that allows creating tasks, events, alarms, resources, semaphores, etc. Furthermore, Erika has a small footprint, which translates to a small surface for safety threats and cyber-attacks, thus resulting in being a good candidate to run the most safety-critical parts of Apollo.

Erika comes with a development tool called RT-Druid. RT-Druid is based on Eclipse and allows to write, compile, and analyze applications. An Erika-based system is *statically* configured by means of an *OIL* file, which includes all the

information required to define it. For example, tasks, resource semaphores, counters, and alarms (used to trigger real-time tasks periodically) are static and defined in the OIL file, and hence known at compile time.

C. KVM and IVSHMEM

A hypervisor is the standard solution to integrate different domains on the same hardware platform by managing the accesses to each component (e.g., cache memories [14], interconnects [15], or I/O devices [16]). Domains are implemented by virtual machines, handled by the hypervisor itself. More VMs can co-exist on the same hardware: the hypervisor also provides isolation among them (e.g., by means of reservation-based mechanisms [17]–[19]).

The hardware machine used by the hypervisor is said to be the *host*; instead, the VM that uses the resources granted by the hypervisor is said to be the *guest*. Hypervisors are categorized in two types: type-1 (also called bare-metal) hypervisors, which run directly on the host hardware, and type-2 hypervisors (also called hosted), which execute on an operating system as an application. In this paper, we focus on a type-2 hypervisor, namely, QEMU/KVM [20] (Quick EMUlator/Kernel-based Virtual Machine), as it is a convenient solution for developing our prototype. Indeed, KVM/QEMU executes on Linux and allows running the Linux-based domain of Apollo directly on the host machine, creating a single guest domain (for Erika). It is composed of two parts: KVM and QEMU. Specifically, KVM allows using some specific hardware features to virtualize the processor cores, while QEMU acts on the Linux side and creates a thread for each virtual processor of the VM [18]. For brevity, this configuration will be referred to simply as "KVM" in this paper.

Allowing Linux and Erika to communicate requires dealing with an *inter-domain* communication mechanisms provided by the hypervisor. In our prototype, we used *IVSHMEM* [21] (Inter-VM Shared Memory), which exposes the shared memory as a PCI device with three base address registers (called *BARs*), where: `BAR0` stores the device registers, `BAR1` provides inter-VM interruption functionalities, and `BAR2` maps the shared memory object.

III. A Multi-domain Software Architecture for Autonomous Driving

Fig. 2 illustrates the proposed multi-domain software architecture for Apollo-based autonomous driving systems. The system is divided into two domains: a *critical domain* and a *non-critical* domain.

The first domain runs on Erika and includes all most critical functionalities for the car, e.g., the control, CAN bus, and guardian components. Conversely, the second domain runs on Linux, which conveniently provides the software stacks to interact with I/O devices such as lidars and cameras and perform hardware acceleration of deep neural networks on GPUs. Therefore, the non-critical domain includes all the advanced functionalities with such needs, e.g., artificial-intelligence-based perception, localization, planning, and prediction. Low

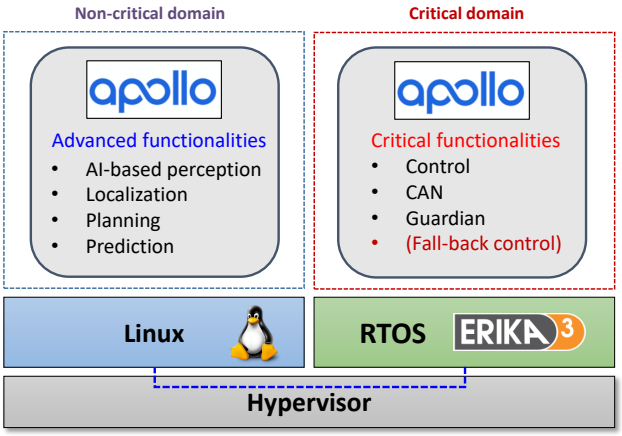


Fig. 2: Multi-domain architecture for Apollo.

critical functionalities as the HMI are also implemented in this domain.

With some components remaining on Linux and some moved to Erika, the CyberRT-based communication needs to be restored.

To this end, the components moved to Erika are replaced by corresponding *bridging* components in the Linux domain, which act as a proxy and interact with the inter-domain communication mechanism provided by the hypervisor.

The general guidelines to select which components to include in the critical domain are reported in the following list:

GL1 The component needs to be highly safety-critical;
GL2 The component should be suitable for certification;
GL3 The component should not require complex software stacks and device drivers that are not available on an RTOS.

These guidelines determined the high-level components-to-domain assignment illustrated in Fig. 2: control, CAN-based communication, and the monitor/guardian are well-suited to run in the critical domain, as they conform to all the guidelines. On the contrary, AI-based perception, localization, planning, and prediction are kept on the Linux domain, as they all either rely on hardware accelerators (e.g., AI-based perception or prediction that requires a GPU and the NVIDIA software stack), or are not suitable for certification (e.g., AI-based perception or perception, which rely on deep neural network and other techniques based on artificial intelligence that may be subject to adversarial attacks [7]), or require other software stacks, libraries, and device drivers that are not available or portable to an RTOS.

The critical domain should also guarantee to bring the car in a safe state when misbehavior is detected. Currently, this is handled in Apollo by the guardian module, which provides only minimal actions in correspondence of a fault. More in detail, it implements a "hard-brake" if the ultrasonic sensors are not working or if they are working and an obstacle is detected; otherwise, it triggers a "soft-stop".

Nevertheless, a safe autonomous driving architecture needs to implement a more robust reaction to faults [22]. In our architecture, this is implemented by a *fall-back controller*, which is in charge of implementing control functionalities based on minimal perception features, e.g., on data provided by legacy sensors only (e.g., radars) that can be fully managed by the Erika domain without the intervention of Linux. However, this feature is not present in Apollo yet, and it requires further studies that will be addressed in future work.

IV. Design and Implementation in Apollo

After presenting the multi-domain software architecture, we discuss several shortcomings we faced in moving safety-critical components of Apollo, the control component, from Linux to Erika, the corresponding solutions, and how the communication has been restored using the `IVSHMEM` inter-domain communication mechanism provided by KVM.

A. Overview

First, we discuss the setup we considered in implementing our multi-domain prototype. Given the current unavailability to us of an actual car to develop the system and the risk of developing a prototype on a car, we considered an HW-in-the-loop simulation environment based on the LGSVL Simulator [23]. Our simulation environment consists of two different physical machines, one for LGSVL, and for run Apollo, where the latter runs exactly the same software that would run on an actual car for almost all enabled components.

However, when running in simulation, Apollo does not enable all the components: the CAN bus component and the Guardian are not used (i.e., the CAN bus is handled directly by LGSVL [24]).

Consequently, we decide to implement our prototype by only porting the control component on the Erika domain. Nonetheless, porting a single and apparently simple component revealed to be a hard task: indeed, the control component leverages several libraries that are not trivial to move outside Linux, as extensively discussed later in this section.

Furthermore, the control component interacts with several other modules: Fig. 3 illustrates its main communication relations.

The control component receives data from the planning, localization, CAN Bus, and HMI components regarding the trajectory, localization estimate, chassis status, and data for the HMI. Furthermore, it writes the control command on another topic to be transmitted on the CAN bus by the corresponding component.

The messages exchanged with the HMI are not relevant in the context of this paper: indeed, they cannot be generated in simulation.

In a simulated environment, since no CAN device is available, the actual CAN bus component is not started, and another software module runs in place of it.

The control component consists of a periodic activity triggered every 10 ms. From a functional perspective, it consists of two main functions calls: **(i)** `init()`, which initializes the module and loads the configuration parameters for the control algorithms; and **(ii)** `proc()`, which acquires the data from the CyberRT channels, computes the control commands, and sends them to the CAN bus component.

B. Pub/Sub communication with CyberRT

As a first step, we performed a deep inspection of the Apollo and CyberRT codebase to understand them in detail. In particular, we highlight some aspects related to the publish/subscribe communication implemented by CyberRT that resulted in being helpful to know when developing our multi-domain solution.

The control module uses the `NodeChannelImpl` class to communicate. The message types to exchange are defined by protobuf[1] files. When the `NodeChannelImpl` class for a node is created, it is notified to a *nodes manager*, which keeps track of all the `NodeChannelImpl` classes created. To use the publish/subscribe paradigm, each module needs to create `reader` and `writer` objects. Specifically, to subscribe to a topic, the module needs to create a `reader` object for such a topic.

When a `reader` is created, optionally, it is possible to associate to it a callback function. Then, when a new message is received, it is stored in the reader class and, if a callback function has been registered, it is activated. Similarly, if a module wants to publish on a specific topic, it needs to create a `writer` object.

C. Porting the required libraries to Erika

As a preliminary, a bare-metal ERIKA3 RTOS image for x86-64 has been created to run Erika with the KVM hypervisor. This configuration requires a specific bare-metal toolchain to compile the program [25], which includes a compiler, linker, and libraries to provide interfaces to the operating system, and a debugger.

Before porting the control component on the Erika RTOS, the first step involved isolating the component from the Apollo codebase. Then, the control module has been taken out from Apollo, with all the necessary files to compile it, i.e., the files of the control module sub-folder and the protobuf files of the components that communicate with the control module.

Then, we compiled it on Linux but outside Docker. This step helped in identifying the library dependencies of the control component:

- `qpOASES` [26]: is an open-source C++ implementation of the Online Active Set Strategy using the quadratic programming (qp). It is used by the controller to compute the control commands.
- `protobuf` [27]: is the library used by Apollo to standardize the format of configurations and messages exchanged among components.
- `gflags` [28]: is a library to specify flags and use them during the program execution. An example of a flag is the

[1]Google's Protocol buffers (protobuf) is a mechanism for serializing structure data, similar to XML files.

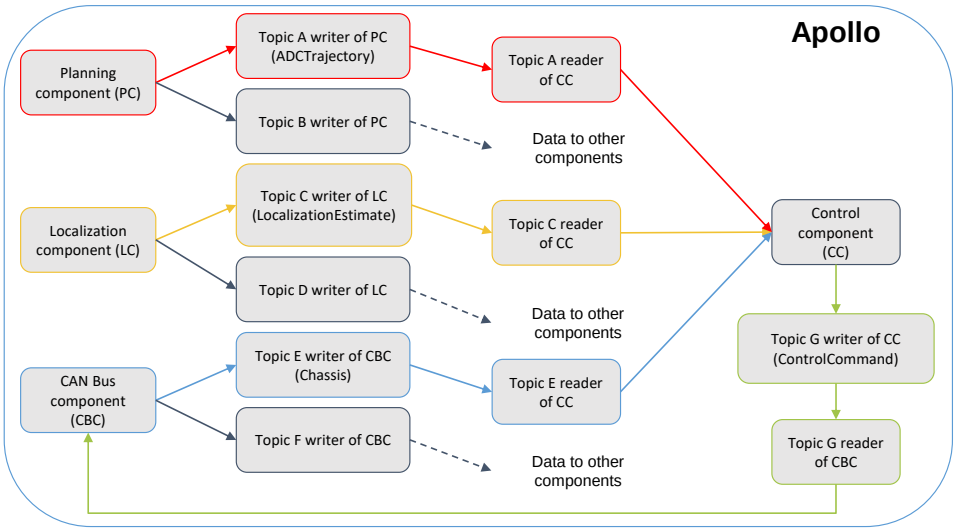


Fig. 3: Communication relations of the control component. Different data flows are highlighted with different colors.

path of the configuration file of the module or the size of the reader message queue.

- `glog` [29]: implements application-level logging. The library provides logging APIs based on C++ style streams and various helper macros.

Then, the component needs to be compiled for Erika. However, these libraries subtly rely on different features of Linux. Since Erika uses a different compiler (i.e., `x86_64-unknown-elf-g++`), it is necessary to re-compile the libraries used by the control component (listed above).

To reduce the memory usage of Erika, the logging functionality is disabled, and hence the glog library is removed.

The other libraries, gflags, qpAOSES, and protobuf, are therefore modified to be compiled with the bare-metal toolchain.

In the case of the `qpOASES` library, the modifications concern the change of the compiler, the addition of the paths of the include files, and the paths of the libraries to link.

Conversely, running `gflags` on Erika required to properly configure CMake, the build solution used for this library, to use the bare-metal compiler required for Erika.

Finally, porting the `protobuf` library still required similar efforts (e.g., properly configuring the bare-metal compiler in the Makefile), but it also required resolving a dependency with respect to the `pthread` library, used by protobuf. Indeed, `pthread` is not available on Erika; therefore the library has been modified to avoid any dependency on it. This modification has been particularly challenging.

D. File system and configuration files

When the control component is initialized, it needs to read several configuration files containing multiple configuration options of the control algorithms, e.g., coefficients of PID controllers and more sophisticated configurations of more complex control strategies. To this end, a file system is needed, but the standard configuration of Erika does not provide it. However, Erika supports `FatFS` [30], a tiny library that provides FAT/exFAT support for small embedded systems, written in compliance with ANSI C (C89). Therefore, `FatFS` has been used to develop the Erika domain.

Furthermore, it is necessary to provide the Erika domain with all the needed files before the system's startup to allow the control module to read them during the initialization.

The control component never writes data to the file system but only reads from it. Therefore, we create a static file system of 1 MB with the following Linux commands:

```
$ dd if=/dev/zero of=file.fs bs=1024 count=1024
# mkfs.fat file.fs
# mount file.fs /mnt/tmp/
```

The first command creates the binary file, *file.fs*, with a given size; the second command creates a FAT file system, and the last command mounts the file system to allow adding the configuration files.

To use this file system in Erika, we converted it from a binary file to a C/C++ header file using the *bin2header* tool [31]. In this way, the header file stores a character array, thus allowing to pass the address of the file system to the FatFs library and mounting it on Erika using the function provided by the FatFs library.

Apollo's control module configuration files are managed by the `protobuf` library, which is based on a POSIX API. For example, the API for reading and writing data from/to the file systems are:

```
int read(int fd, void *buf, int count)
int write(int fd, const void *buf, int count)
```

where `fd` is the *file descriptor*, a non-negative number representing a file in a POSIX system.

Instead, the API of the `FatFs` library provides a much different API, thus not allowing seamless integration with the `protobuf` library. For example, the API for reading and writing data with `FatFs` is:

```
FRESULT f_read(FIL* f, void* buf, UINT btr, UINT* b)
FRESULT f_write(FIL* f, void* buf, UINT btw, UINT* b)
```

To allow `protobuf` using `FatFs`, a POSIX-compliant wrapper has been applied to it to provide the same interface of a POSIX system.

Moreover, Erika already has some of the functions provided by the POSIX API, i.e., `write`, `read`, `close`, `fstat`, `lseek` and `isatty` functions, which are also needed in the POSIX wrap. Erika uses these functions for the serial communication. Therefore, these functions are modified to keep their original functionality and serve as wrappers for the FatFs functions. To this end, for example, at the beginning of the read and write operations, the file descriptor is checked to distinguish serial (UART) communication (i.e., C++ standard input and output, `stdin` and `stdout`) from operations on files: in the first case, the code provided within Erika is run; otherwise, the FatFs function executes.

E. The control component on Erika

Next, we discuss how to move the control component on Erika. Since it is a periodic activity, it has been implemented by using the standard facilities provided by Erika for activating a periodic task, i.e., by properly configuring a counter, an alarm, and task object in the `OIL` file. The counter counts 1 ms for every tick, and the alarm triggers the task when the counter reaches 10.

However, to integrate the control component in Erika some changes are required. First, in the header file, as well as in the `input` and `proc` functions, it is required to remove any interaction with the Cyber RT channels and substituting them with an interaction with the shared memory provided by KVM by means of `IVSHMEM`. The details of the adopted inter-domain communication mechanism are discussed next in Section IV-G.

To make the solution working, many problems needed to be addressed. Indeed, in the beginning, Erika was not able to boot. Using the `gdb` debugger of the toolchain, it has been found out that the size of the application was too large for Erika (the compiled control component resulted to be about 6 MB and the memory areas critical for the startup of the RTOS were overwritten). Accurate modifications to Erika were then required. They were related to the memory layout and, in particular, to the size of the various memory regions.

F. Bridging component on Linux

On the Apollo side, it is necessary to manage the interaction between the inter-VM shared memory and the planning, localization, and CANBus modules. So, the control component inside Apollo has been used to act as a "bridge", thus managing the communication between the actual control component (called "external control component", running on Erika) and the remainder of Apollo (running on Linux). The bridging component **(i)** writes the received messages from the localization, CANBus, and planning modules in the shared memories; and **(ii)** reads the output of the external control component, still from the shared memory, and forward it to the CANBus (remember that the CANbus module in our prototype setting is just used to forward data to the simulator — in a complete setting of the multi-domain architecture the CANBus module should be moved to the critical domain).

A straightforward solution to implement the bridging component would have been to leave it periodic with 10 ms period, as the Apollo's control component. However, this may introduce additional delays in the worst-case: since the bridge component on Linux and the external control component on Erika are not synchronized, the latter may read the shared memory just ϵ (with $\epsilon > 0$ arbitrary small) times unit before new data becomes available, therefore introducing a worst-case sampling delay of one period [32] (i.e., 10 ms).

To avoid introducing additional delays in implementing our solution, we leveraged the features offered by CyberRT to execute event-driven computations. Specifically, a callback function has been added to the `reader` object of each topic to write the message in shared memory as soon it arrives, i.e., in a *synchronous* way.

Since callbacks are triggered upon message arrival on a topic, this solution cannot be used to handle the data flows starting in the external control component and ending in the CANBus module.

Indeed, the external control component does not interact directly with CyberRT, and it has to write in the inter-VM shared memory. Therefore, a specific thread has been created to check for the arrival of new messages. The thread constantly checks the shared memory and, when the external control component makes new control commands available, it reads the commands and notifies the message to a `write` object used by the bridging component to interact with the control command topic.

In summary, the bridging component is implemented in place of the original control component of Apollo, to: **(i)** initialize the shared memories and the needed pointers; **(ii)** create `reader` and `writer` objects for the topic, associating a callback where needed; and **(iii)** create the thread to handle the control command message.

Fig. 4 shows an overview of the implemented communication mechanism, where the differences with respect to the original Apollo have been highlighted with a bold border.

G. Using `IVSHMEM` to restore communication

To restore the communication and receive and send the messages to the other modules in Apollo, we leveraged the `IVSHMEM` inter-domain communication mechanism provided by KVM.

`IVSHMEM` offers the possibility to create a shared memory device between virtual machines and the host. In this work, we created a shared memory device with a size of 1 MB, which is large enough to store the four messages exchanged by the two domains.

Four shared memory regions have been used to communicate with Apollo: one to receive planning messages, one for localization messages; one for chassis messages; and one to send the control commands. Fig. 5 illustrates the internal

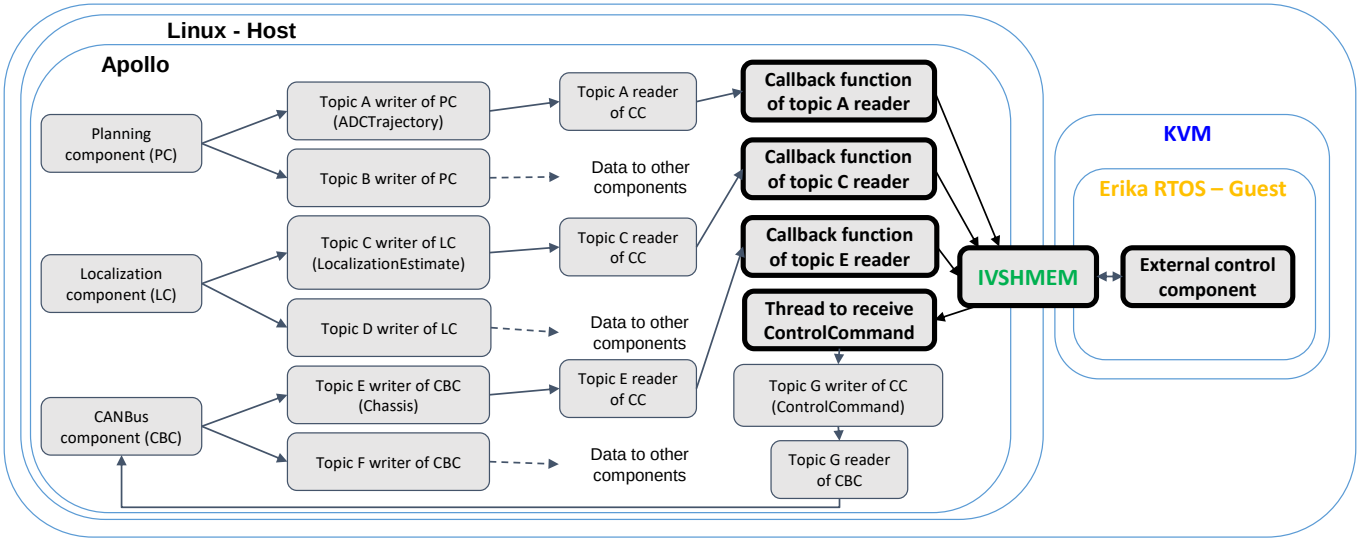


Fig. 4: Architecture of the multi-domain Apollo prototype.

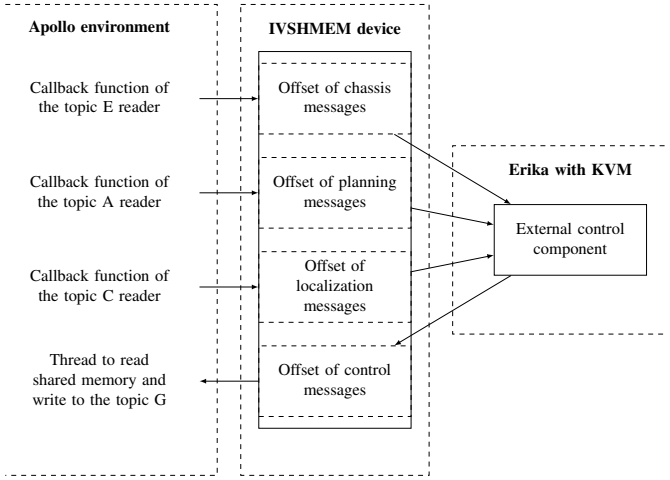


Fig. 5: Layout of the shared memory based on `IVSHMEM`.

scheme of the shared memory device. Although the `reader` object stores messages in a queue of multiple elements, by analyzing the behavior of Apollo and CyberRT we noted that the new message is always stored in front of the queue, and the control component always gets the most recent message, i.e., it only accesses the top of the queue. This has also been confirmed by the Apollo developers [33]. Therefore, for each topic, a buffer has been created in shared memory to store the most recent message.

To write or read a specific message in the shared memory device, the offset where that message is located must be known.

On the Linux side, `IVSHMEM` exposes a PCI device with three base address registers discussed in Section II-C, where `BAR2` maps the shared memory. Conversely, Erika does not have a device manager, and therefore it cannot access the shared memory as a PCI device. Therefore, it has been necessary to discover the address where KVM loaded the device, specifically the address of `BAR2`, and map this address at the Erika initialization.

To obtain this information, the `qemu monitor` has been used. Then, to map the `BAR2` in the address space seen by Erika applications and let the memory region be accessible by the control task, we leveraged the `osEE_x86_64_map_range` function offered by Erika. Once mapped the device address, the shared memory becomes accessible using a pointer, and pointer arithmetic is used to switch among the four messages.

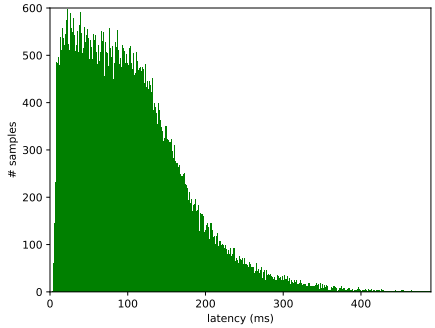
V. Evaluation

The system described in Section IV has been implemented in Apollo 5.0. Two Intel x86_64 machines have been used in the evaluation, one to run Apollo and one for the LGSVL simulator.

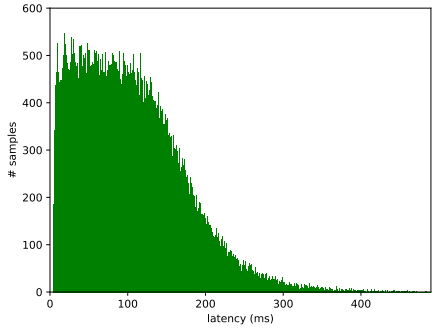
The Apollo machine is equipped with an Intel Core i7-6700K with 8 logical cores running at 4.00 Ghz, an NVIDIA GeForce GTX 1080-Ti, and 32GB of RAM. The LGSVL machine is equipped with an Intel Core i9-9900 with 16 logical cores running at 3.10 Ghz, an NVIDIA GeForce RTX 2080 Super, and 32GB of RAM.

In the evaluation, we compared the latencies of three paths passing through the control component: the chassis to control command message (`chassis2command`), the localization to control command message (`loc2command`), and the planning to control command message (`planning2command`).

The latencies are computed as follows. First, a timestamp is taken when the reader of the three topics, i.e., `Chassis`, `LocalizationEstimate`, and `ADCTrajectory` (see Fig. 4) arrives at the `reader` object of the control (bridge) component. Then, the messages pass through the control component, and a second timestamp is taken when the ControlCommand message is ready to be sent by the corresponding

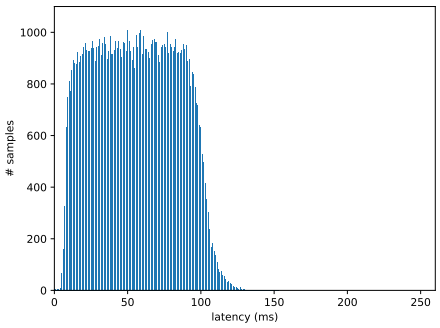


(a) Apollo Standard

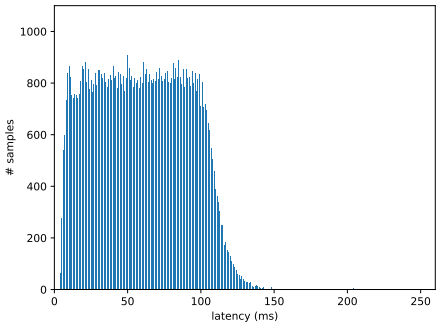


(b) Proposed Approach

Fig. 6: Histograms of the `planning2command` latencies.



(a) Apollo Standard



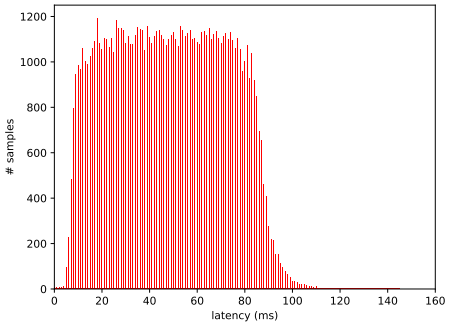
(b) Proposed Approach

Fig. 7: Histograms of the `chassis2command` latencies.

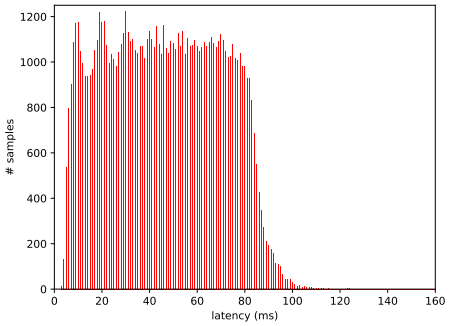
`writer` object. The timestamps are then used to compute the latencies. Fig. 6, Fig. 7, and Fig. 8 report the distributions of the observed delays for the three topics. For each of them, inset (a) targets measurements obtained with the standard version of Apollo 5.0, while inset (b) considers the multi-domain Apollo prototype discussed in this paper. In each chart, the y-axis reports the number of samples for which a given delay on the x-axis has been observed. The samples have been obtained by running the two configurations, i.e., standard and multi-domain Apollo, for 15 minutes each. For all the three latencies `chassis2command`, `loc2command`, and `planning2command`, the distribution of the delays is very similar between standard Apollo and the proposed solution, thus showing that the adoption of our multi-domain approach does not introduce considerable delays. Indeed, thanks to the usage of the callback mechanism provided by CyberRT, each new message is synchronously placed in the shared memory buffer provided by `IVSHMEM` as soon it becomes available to the `reader` object of each topic, as discussed in Section IV-F.

VI. Related Work

Recent work [22] proposed a safe, secure, and predictable software architecture for adopting deep learning in safety-critical systems. While our work shares the multi-domain architecture with it, the work in [22] is not explicitly designed for autonomous driving, and it provides no implementation. The recent work by Alcon et al. [34] discussed the challenges



(a) Apollo Standard



(b) Proposed Approach

Fig. 8: Histograms of the `loc2command` latencies.

in analyzing the timing of Apollo and studied the execution time variability observed when running it on a GPU-based platform.

Other papers targeted the Autoware autonomous driving framework [1]. Bateni et al. [35] implemented a predictable data-driven resource manager in Autoware. In another work, Bateni et al. [36] proposed a method to co-optimizing performance and memory footprint by jointly managing the CPU and GPU memory management, showing the benefits of their approach on Autoware.

Other works focused on the predictability of middleware frameworks [37]–[41]. Most relevant to us are those regarding ROS [41]–[46], which implements the publish-subscribe paradigm in a similar way as CyberRT does.

Other works studied methods to improve the reliability of autonomous driving from an orthogonal perspective, i.e., by improving the reliability of deep neural networks. Several results aimed at addressing the problem of adversarial attacks: for example, Nesti et al. [47] proposed a method to detect them via input transformation, defense perturbations, and voting. A complete overview on the topic would require much more than the space provided with this paper: therefore, we leave the interested reader to the surveys by Chakraborty et al. [48] and Zhang and Chen [7].

To improve the safety in autonomous driving, Kelly et al. [49] proposed a tool to verify the safety of trajectories of autonomous cars.

Finally, it is worth mentioning the Elisa project [50] launched by the Linux Foundation, which aims at enabling Linux for safety-critical applications, thus being of potentially great interest for autonomous driving.

VII. Conclusions and Future Work

This paper presented a multi-domain software architecture for autonomous driving based on the Apollo framework. First, we discussed the proposed software architecture, which allows leveraging multiple domains with different levels of criticality. We debated several non-trivial shortcomings that need to be addressed when implementing a prototype of such an architecture, targeting the specific case in which the critical domains handle the control module. In particular, we illustrated the library dependencies of the control component and how support them on the Erika RTOS, although they are originally designed for Linux. Then, we discussed how to restore the communication between components on the critical and non-critical domain by leveraging the inter-domain communication mechanism provided by the hypervisor, making sure that no significant additional delays are introduced due to sampling issues. Finally, our evaluation showed how the proposed system can be efficiently implemented, introducing almost negligible additional latencies.

While the proposed implementation shows a practical example of how to instantiate a multi-domain architecture for autonomous driving in a real (but simulated) system, much further work is required to realize a proper multi-domain architecture for autonomous driving.

Some of the future works include:

Time-predictable and isolated acceleration of DNNs. Another interesting research direction includes making Apollo more predictable by acting on the perception module, which strongly relies on deep neural networks. Currently, Apollo accelerates them through NVIDIA GPUs, which scheduling behavior is hard to predict due to many due to the many internal details undisclosed by NVIDIA [51]. Ongoing work is targeting the usage of way more predictable FPGA-based accelerators [52], which can also enable a proper isolation of the memory traffic generated by DNN inference. Another interesting future research direction involves using more predictable GPU designs recently proposed in the research community [53], or the usage of AMD GPUs [54].

Fail-safe controller and safe perception module. In Section III, we discussed the possibility of extending the Guardian module of Apollo to switch the system to use a safe controller providing minimal perception features based on legacy sensors, thus not relying on non-critical modules that can potentially be compromised or faulty. Future work may investigate methods to implement such a controller starting from the existing literature (e.g., [55, 56]), to bring the system into a safe state or do the best to avoid accidents/collisions when a failure is detected. Even more ambitious future research can develop a minimal safe perception module (based on verified/safe artificial intelligence techniques) in the critical domain.

End-to-end analysis of CyberRT. Timing predictability is a key feature for making a system safer by guaranteeing that all the tasks are completed within their deadlines. In the context of complex automotive software characterized by chains of tasks, end-to-end latency analysis techniques are usually developed to predict worst-case latencies. However, when dealing with autonomous driving frameworks, the situation is much more complicated: a proper analysis should consider the joint effect of the operating system and middleware frameworks, which can substantially affect the application timing behavior. For example, this has been demonstrated in the context of ROS 2, used by Autoware, for which end-to-end analyses have been derived [41]. Similarly, deriving an end-to-end analysis of CyberRT would be highly beneficial for analytically bounding the latencies in Apollo. Another interesting research direction involves modifying the scheduling policy provided by the middleware layer to favor predictability [44].

Porting to an embedded platform. Currently, the Apollo prototypical cars embed an industrial PC: to address the so-called SWaP problem (space, weight, and power), it would be advisable to move Apollo to embedded platforms. To this end, modern heterogeneous platforms equipped with hardware accelerators are good candidates. Due to the high-computational requirements of AD software, a multi-SoC (system on a chip) solution may be required [57]. At the time we are writing this paper, there are ongoing efforts in our group for porting Apollo on a Xilinx Ultrascale+ platform.

ACKNOWLEDGMENTS

This work has been partially supported by Huawei and the Department of Excellence in Robotics and Artificial Intelligence of Scuola Superiore Sant'Anna, Pisa, Italy. The authors would like to thank Bruno Morelli and Claudio Scordino from Evidence/Hauwei, Pisa, for their support in configuring the Erika RTOS for applications with large memory requirements.

REFERENCES

[1] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 287–296.
[2] [Online]. Available: https://apollo.auto
[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.
[4] [Online]. Available: https://cyber-rt.readthedocs.io/en/latest/
[5] TensortRT. Https://developer.nvidia.com/tensorrt.
[6] Z. Peng, J. Yang, T.-H. P. Chen, and L. Ma, "A first look at the integration of machine learning models in complex autonomous driving systems: A case study on apollo," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 1240–1250.
[7] J. Zhang and C. Li, "Adversarial examples: Opportunities and challenges," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 7, pp. 2578–2593, 2020.
[8] [Online]. Available: https://www.baidu.com
[9] [Online]. Available: http://www.erika-enterprise.com
[10] [Online]. Available: http://www.evidence.eu.com/products/erika-enterprise.html
[11] [Online]. Available: https://github.com/ApolloAuto/apollo
[12] D. B. de Oliveira, D. Casini, R. S. de Oliveira, and T. Cucinotta, "Demystifying the Real-Time Linux Scheduling Latency," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, 2020.
[13] [Online]. Available: http://www.evidence.eu.com
[14] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, 2018, pp. 1651–1657.
[15] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "Axi hyperconnect: A predictable, hypervisor-level interconnect for hardware accelerators in fpga soc," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
[16] D. Casini, A. Biondi, G. Cicero, and G. Buttazzo, "Latency Analysis of I/O Virtualization Techniques in Hypervisor-Based Real-Time Systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 306–319.
[17] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013.
[18] L. Abeni, A. Biondi, and E. Bini, "Hierarchical scheduling of real-time tasks over linux-based virtual machines," *Journal of Systems and Software*, vol. 149, pp. 234–249, 2019.
[19] D. Casini, L. Abeni, A. Biondi, T. Cucinotta, and G. Buttazzo, "Constant bandwidth servers with constrained deadlines," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17, 2017.
[20] Kvm. [Online]. Available: https://www.linux-kvm.org/page/Main_Page
[21] Nahanni: a shared memory interface for kvm. [Online]. Available: https://www.linux-kvm.org/images/e/e8/0.11.Nahanni-CamMacdonell.pdf
[22] A. Biondi, F. Nesti, G. Cicero, D. Casini, and G. Buttazzo, "A safe, secure, and predictable software architecture for deep learning in safety-critical systems," *IEEE Embedded Systems Letters*, vol. 12, no. 3, pp. 78–82, 2020.
[23] LGSVL simulator. [Online]. Available: https://github.com/lgsvl/apollo-5.0
[24] [Online]. Available: https://github.com/ApolloAuto/apollo/issues/13109
[25] [Online]. Available: http://www.erika-enterprise.com/wiki/index.php?title=Bare-metal_x86-64_image
[26] [Online]. Available: https://github.com/coin-or/qpOASES
[27] [Online]. Available: https://developers.google.com/protocol-buffers
[28] [Online]. Available: https://github.com/gflags/gflags
[29] [Online]. Available: https://github.com/google/glog
[30] [Online]. Available: http://elm-chan.org/fsw/ff/00index_e.html
[31] [Online]. Available: https://antumdeluge.github.io/bin2header
[32] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007.
[33] [Online]. Available: https://github.com/ApolloAuto/apollo/issues/13265
[34] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, "Timing of autonomous driving software: Problem analysis and prospects for future solutions," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 267–280.
[35] S. Bateni and C. Liu, "Predictable data-driven resource management: an implementation using autoware on autonomous platforms," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 339–352.
[36] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, "Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.
[37] D. Casini, A. Biondi, and G. Buttazzo, "Analyzing parallel real-time tasks implemented with thread pools," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, 2019.
[38] R. Vargas, E. Quinones, and A. Marongiu, "Openmp and timing predictability: A possible union?" in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 617–620.
[39] D. Casini, A. Biondi, and G. Buttazzo, "Timing isolation and improved scheduling of deep neural networks for real-time systems," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1760–1777, 2020.
[40] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of openmp4 tasking model," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.
[41] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, 2019.
[42] Y. Tang, F. Zhiwei, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors," *Proceedings of the 41st IEEE Real-Time Systems Symposium (RTSS)*, 2020.
[43] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems," in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.
[44] H. Choi, Y. Xiang, and H. Kim, "PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2," in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.
[45] Y. Suzuki, T. Azumi, S. Kato, and N. Nishio, "Real-Time ROS Extension on Transparent CPU/GPU Coordination Mechanism," in *Proceedings of the 21st IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, 2018.
[46] Y. Saito, T. Azumi, S. Kato, and N. Nishio, "Priority and Synchronization Support for ROS," in *4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications)*, 2016.
[47] F. Nesti, A. Biondi, and G. Buttazzo, "Detecting Adversarial Examples by Input Transformations, Defense Perturbations, and Voting," *IEEE Transactions on Neural Networks and Learning Systems*.
[48] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, "Adversarial Attacks and Defences: A Survey," *arXiv e-prints*, p. arXiv:1810.00069, Sep. 2018.
[49] M. O'Kelly, H. Abbas, S. Gao, S. Shiraishi, S. Kato, and R. Mangharam, "Apex: Autonomous vehicle plan verification and execution," 2016.
[50] Elisa project (Linux Foundation).
[51] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2017.
[52] B. B. Seyoum, M. Pagani, A. Biondi, S. Balleri, and G. Buttazzo, "Spatio-temporal optimization of deep neural networks for reconfigurable fpga socs," *IEEE Transactions on Computers*, 2020.
[53] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for gpu with preemption support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 119–130.
[54] N. Otterness and J. H. Anderson, "AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 165, 2020, pp. 10:1–10:23.
[55] K. Angelopoulos, A. V. Papadopoulos, V. E. S. Souza, and J. Mylopoulos, "Engineering self-adaptive software systems: From requirements to model predictive control," vol. 13, no. 1, 2018.
[56] S. Shevtsov, D. Weyns, and M. Maggio, "Simca*: A control-theoretic approach to handle uncertainty in self-adaptive systems with guarantees," vol. 13, no. 4, 2019.
[57] A. Biondi *et al.*, "SPHERE: A Multi-SoC Architecture for Next-Generation Cyber-Physical Systems Based on Heterogeneous Platforms," *IEEE Access*, vol. 9, pp. 75 446–75 459, 2021.