Autonomous-driving frameworks and predictability: challenges and open problems

Daniel Casini
Assistant Professor, Scuola Superiore Sant’Anna, Pisa, Italy

Invited Talk at Workshop CAPITAL 2022: sCalable And PrecIse Timing AnaLysis for multicore platforms, Friday, June 3th, 2022. Grenoble
This Talk

1. Introduction to Frameworks for Autonomous Driving and their challenges for time-predictability
   - Apollo – based on Cyber RT
   - Autoware – based on ROS 2

2. Apollo – contributions and open problems

3. ROS 2 – contributions and open problems
Autonomous Driving

Autonomous driving is the next step in the history of human transportation

Potential future benefits with respect to a human driver:

- More reliable driving
- Easier parking
- No unexperienced driver on the roads
The Society of Automotive Engineers (SAE) identifies six levels of driving automation:

First 3 levels: The human driver is still the core of the driving system

**LEVEL 0: No driving automation**
- Vehicles are manually controlled (steering wheel, throttle, brakes, and everything)

**LEVEL 1: Driver Assistance**
- Cruise Control and Adaptive Cruise Control. The driver is assisted in basic operations such as steering or accelerating

**LEVEL 2: Partial Driving Automation**
- Advanced driver assistance systems (ADAS). Examples are Tesla Autopilot and Cadillac Super Cruise Systems. The vehicle controls steering, deceleration, and acceleration. Automatic lane keeping. Some actions may be not supported by the autonomous car. The driver needs to be ready to take the control at any time.
LEVEL 3: Conditional Automation

- Vehicles have environmental detection capabilities, and drives autonomously. The driver needs to take over when necessary.

LEVEL 4: High Automation

- Vehicles controls every aspect of driving. The vehicle can intervene even if there is a system failure. No expectation for a human intervention, but the human has still the option to drive manually. There are legal limitations for the areas in which these vehicles can operate in self-driving (geofencing).

LEVEL 5: Full Automation

- The vehicles is autonomous, without any geographic limitation. No steering wheel or driving pedals!
How Self-Driving Car Works

Usage of images from cameras to perceive the world

Integration of data from other sensors for better understanding the surrounding environment

Computer Vision

Sensor Fusion

Localization → Path Planning → Control
How Self-Driving Car Works

Use data from **computer vision** and **sensors** to understand the position

Derive a path toward where we want to go

How to **brake**, **throttle**, and **steer** to realize the trajectory
Frameworks for Autonomous Driving

Two main open-source projects: Apollo and Autoware

Autoware

- Started in **2015** by Shinpei Kato at Nagoya University (Japan).
- 2300+ stars on GitHub and 500+ accounts on Slack (10/2018).
- More than 100 companies and runs on more than 30 vehicles in more than 20 different countries.

Baidu

- Started in **2017** by Baidu (China).
- The only company to obtain the first batch of road test licenses in China.
- Recently, started the Robotaxi project in China.
- The first Autoware project based on ROS 1
- Released as a research and development platform for autonomous driving

- Second version of Autoware, based on ROS 2
- Based on a redesigned architecture with better software engineering practices
The Apollo Autonomous Driving Framework
The Apollo Stack

Four layers

1. **Reference Vehicle Platform**

2. **Reference Hardware Platform**

3. **Open Software Platform**

4. **Cloud Service Platform**

---

<table>
<thead>
<tr>
<th>Cloud Service Platform</th>
<th>HD Map</th>
<th>Simulation</th>
<th>Mass Production Service Component</th>
<th>Security</th>
<th>OTA</th>
<th>DuerOS</th>
<th>V2X Roadside Service</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Pipeline</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open Software Platform</td>
<td>Map Engine</td>
<td>Localization</td>
<td>Perception</td>
<td>Prediction</td>
<td>Planning</td>
<td>Control</td>
<td>HMI</td>
</tr>
<tr>
<td></td>
<td>RTOS</td>
<td></td>
<td>Apollo Cyber RT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Development Platform</td>
<td>Computing Unit</td>
<td>GPS/IMU</td>
<td>Camera</td>
<td>LiDAR</td>
<td>Radar</td>
<td>Ultrasonic Sensor</td>
<td>HMI Device</td>
</tr>
<tr>
<td>Open Vehicle Certification Platform</td>
<td>Certified Apollo Compatible Drive-by-wire Vehicle</td>
<td>Open Vehicle Interface Standard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Apollo GitHub.
# The Apollo Stack

## Four layers

1. **Reference Vehicle Platform**
2. **Reference Hardware Platform**
3. **Open Software Platform**
4. **Cloud Service Platform**

### Cloud Service Platform
- HD Map
- Simulation
- Mass Production Service Component
- Security
- OTA
- DuerOS
- V2X Roadside Service

### Open Software Platform
- Map Engine
- Localization
- Perception
- Prediction
- Planning
- Control
- HMI
- Apollo Cyber RT
- RTOS

### Hardware Development Platform
- Computing Unit
- GPS/IMU
- Camera
- LiDAR
- Radar
- Ultrasonic Sensor
- HMI Device
- Black Box
- ASU
- AXU
- V2X OBU

### Open Vehicle Certification Platform
- Certified Apollo Compatible Drive-by-wire Vehicle
- Open Vehicle Interface Standard

Apollo GitHub.
Software Components

- Apollo
  - Perception
  - Prediction
  - Planning
  - HMI
  - Control
  - CanBus
  - Monitor
  - Guardian
  - the core of Apollo

- Middleware
  - CyberRT
  - handles publish/subscribe communication

- Operating System
  - Linux
  - Linux equipped with PREEMPT_RT
Modules

- **Localization**: to understand precisely where the car is in the world
- **Perception**: capability of detecting and classifying obstacles
- **Prediction**: studies and predicts the behavior of all the obstacles
- **Planning**: to plan a path towards the destination
- **Control**: actual commands to drive the car (how to turn the steering wheel, how much to hit the throttle, etc.)
- **HMI**: is a module for viewing the status and controlling the functioning of the vehicle
- **CanBus**: handles communication through CAN
- **Monitor**: checks the system’s status and detects possible fault conditions
- **Guardian**: takes actions based on the monitor’s results
Middleware

- This is the operating environment of Apollo
- Up to version 3.0, it was a customized version of ROS
  - The changes implemented by Apollo included optimization of shared-memory communication and the usage of protobuf instead of ROS message
- From version 3.5 on, Apollo is adopting CyberRT
- Communication among modules is performed with a publish/subscribe paradigm
**Topics in Apollo: an Example**

*Inputs (and cyber channels):*

- LiDAR data
- Radar data
- Image data
- velocity and angular velocity
- sensor calibration data
- obstacle data
- traffic light data

```
components {
    class_name: "RadarDetectionComponent"
    config {
        name: "FrontRadarDetection"
        config_file_path: "../front_radar_component_conf.pb.txt"
        readers: [{channel: "/apollo/sensor/radar/front"}]
    }
}
```
Threats to Predictability and Challenges
Threats to Predictability and Challenges
Threats to Predictability and Challenges

A very complex autonomous driving application that needs to satisfy timing constraints.
Threats to Predictability and Challenges

ROS 2 and CyberRT are used to conveniently implement autonomous driving, thanks to the pub/sub infrastructure and the large set of pre-implemented functionalities they provide.
Threats to Predictability and Challenges

The DDS manages the communication among callbacks, with a complex multi-thread software that needs to be properly scheduled by the OS.
Threats to Predictability and Challenges

Application-level, ROS2-level, and DDS-level threads need to be properly scheduled by the OS to meet timing constraints.
Threats to Predictability and Challenges

A hypervisor, if present, can greatly help in implementing temporal and spatial isolation between virtual machines (VM), but it needs to be properly configured.
The underlying hardware platform can be very complex. Usually is equipped with heterogeneous cores and hardware accelerators and it has complex memory hierarchies, introducing further challenges.
What we did to address these threats?
Several contributions:

This Talk

A Multi-Domain Architecture for Autonomous Driving (IEEE ESL, RTCSA 2020)

Response-Time Analysis of ROS 2 (ECRTS 2019, RTSS 2021)

Real-Time Analysis of the QNX Adaptive Partitioning Scheduler (RTAS 2022)

Many others, not covered in the presentation (timing predictability in Tensorflow, QoS regulators for mitigating I/O-related interference, I/O Virtualization, etc.)
Multi-Domain Architecture


Multi-Domain Designs

- A multi-domain architecture used to switch the system to a ‘safe controller’ running on Erika that uses minimal perception features based on legacy sensors (e.g., radars), extending what already done by the Apollo Guardian component.

- An alarm signal can be sent to the driver to take back the control, as mandated by SAE's Level 3 specifications (Conditional Driving Automation, a.k.a., "eyes off").
Multi-Domain Architecture for Apollo

OBJECTIVE
Making Apollo safer and more secure by designing a multi-domain architecture

HOW?

1. Separating Apollo’s modules between a non-critical and a critical domain, running different OSes
2. Using a hypervisor to separate them
3. Restoring the pub/sub CyberRT-based communication

Advanced functionalities:
- HD Map
- AI-based perception
- Localization
- Planning & prediction

Critical functionalities:
- Control
- CAN
- Guardian
- (Fall-back control)

The multi-domain design provides **all critical functionalities** of Apollo in a **separate domain** running **Erika3**.

Advanced Apollo functionalities (e.g., those requiring drivers of software stacks that not available in a RTOS) on a Linux domain and the CyberRT-based communication (based on the pub/sub paradigm) with those moved to the **Erika3** domain must be restored.

The components moved to the Erika3 domain are replaced by “bridging” components in the Linux domain, i.e., they act as a bridge to control inter-domain communication.

A fall-back controller (not present in Apollo) may be realized in the future to perform minimal safe control without the advanced Apollo functionalities.
Prototype Implementation
**HW-in-the-loop Simulation**

- **HW-in-the-loop simulation environment** to work with Apollo

**LGSVL Simulator**

[https://www.lgsvlsimulator.com/](https://www.lgsvlsimulator.com/)

on a powerful desktop computer

**X86 machine with Linux** to run Apollo

**Gigabit Ethernet**

**4 GHz Quad-core CPU, 32GB of RAM, Nvidia GTX 2080 16GB, Win 10 64-bit**
How to split components among VMs?

- In our implementation, we moved the **control component** from Linux to Erika.

- Candidates to run in Erika.

- Not used in simulation.

- Handled by the LGSVL simulator in the HW-in-the-loop simulation.
Why the Control Component?

- It is a highly safety-critical component;
- It does not require complex software stacks or device drivers that are not available on Erika (e.g., for interacting with NVIDIA GPUs, as the perception component does);
- The Control Component is characterized by the following communication relationships (showed in the picture and extracted during our in-depth analysis of the Apollo code).
Apollo officially supports Intel x86 platforms only.

Realized a multi-domain design on Intel x86 using KVM and Erika3 for x86.
The following libraries have been ported to Erika:

1. **qpOASES**: it is an open-source C++ implementation of the Online Active Set Strategy using the quadratic programming. It is used by the controller to compute the control commands; (see https://github.com/coin-or qpOASES)

2. **gflags**: it is a library to manage command-line flags in applications with multiple files. (see https://gflags.github.io/gflags/#intro)

3. **protobuf**: the library to standardize the exchange messages between components. (see https://developers.google.com/protocol-buffers)

4. A **POSIX-FatFS wrapper** to avoid modifying the Apollo code for accessing the file system.
Evaluation

Latency of the planning messages

Standard version

Multi-domain version

Similar latencies have been observed
Other works with Apollo at the RETIS lab
Porting Apollo on an Embedded Platform

Rear view. Apollo GitHub.

- How to replace it with one (or more) embedded platforms?

**Target**: Xilinx Ultrascale+ FPGA-based SoC, high predictability, low power consumption

<table>
<thead>
<tr>
<th>apollo::planning::PlanningComponent::Proc</th>
<th>zcu_mod_testing</th>
<th>pc_mod_testing</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>906.26</td>
<td>105.42</td>
<td>8.6</td>
</tr>
<tr>
<td>Max</td>
<td>3900</td>
<td>216</td>
<td>18.06</td>
</tr>
</tbody>
</table>

Very challenging problem due to huge slow down due to slower cores

X86 machine with Linux to run Apollo

4 GHz Quad-core CPU, 32GB of RAM, Nvidia GTX 2080 16GB, Win 10 64-bit
Optimized Acceleration of DNNs

- **Optimized acceleration** of the Apollo Deep Neural Networks for deployment on embedded platforms
- Quantization & network pruning
  - **Current target**: Xilinx Ultrascale+ FPGA-based SoC, high predictability, low power consumption

### Apollo Deep Neural Networks

- Denseline lane detector
- DarkSCNN
- Traffic Light Detection
- Traffic Light Recognition
- Yolo - Object Detection
- Lidar (Velodyne 16)
Inter-domain communication

- Transparent, shared-memory communication between ROS 2, running in a Linux-based VM and micro-ROS, running on FreeRTOS
- Based on the CLARE hypervisor, developed by Accelerat, a spin-off company of the Scuola Superiore Sant’Anna

![Diagram showing inter-domain communication and hardware accelerators](image)

- ROS 2
  - Low-criticality domain
  - Sensing, Human Interface, Connectivity, Guidance Algorithm
  - CLARE Middleware
  - CLARE-Hypervisor
  - Real-time Linux
  - Sensors
- micro-ROS
  - High-criticality domain
  - Actuation, Emergency Interface, Fall-back Node
  - RTOS
  - Health monitoring
  - Actuators
  - Sensors

Everything on a single platform
Apollo – Open Problems

- Analysis of processing chains under CyberRT
- Fail-safe controller and safe perception module
- Multi-domain architecture with a type-1 hypervisor on an embedded platform
- And many more, e.g., on the AI and hardware acceleration side
Response-Time Analysis of Processing Chains in ROS 2


Why the scheduling in ROS is complex?

Two levels of scheduling:

- ROS processes are scheduled by the Linux operating system.
- Callbacks (e.g., C++ functions) are in turn scheduled by the ROS executor implemented by ROS.

Even more complex when thinking to the complete software stack.
Why the scheduling in ROS is complex?

Two levels of scheduling:

- ROS processes are scheduled by the Linux operating system
- Callbacks (e.g., C++ functions) are in turn scheduled by the ROS executor implemented by ROS

ROS-level analysis accounting for the ROS’s quirks
Based on CPA [1] (Compositional Performance Analysis)

Linux-level scheduling using reservations (e.g., SCHED_DEADLINE)

---

Modeling

- ROS Systems are distributed networks of callbacks

* Slides adapted from the original presentation by Tobias Blass
.Callbacks are assigned to executor threads.

ROS-Level Scheduling

Linux-Level Scheduling

Thread 1

Thread 2
The ROS documentation does not specify the execution order of callbacks
A Real-Time Model for ROS 2

**Timer callback**
- WCET bound $e_i$
- Priority $\pi_i$
- Event arrival curve $\eta_i^a$

**Subscription callback**
- WCET bound $e_i$
- Priority $\pi_i$

**Communication Delay** $\delta_{i,j}$

**Event source external to ROS**
- Event arrival curve $\eta_i^a$

**Processing chains of interest**
The operating system’s view

- **Groups of callbacks**
- **Linux Scheduler**
  - Controls executor scheduling
- **OS Process**
  - **Node**
  - **Executor Thread**
  - **IPC Layer**
  - **Timers**
- **Executor Thread**
- **IPC Layer**
- **Topics**
- **Services**
- **Node**
The operating system’s view

ROS-specific, undocumented

Linux Scheduler

well understood
(SCHED_DEADLINE)

OS Process

Node

Executor Thread

Executor Thread

IPC Layer

Timers

Topics
Services

Executor Thread

IPC Layer

Timers

Node

ROS-specific, undocumented

OS Process

Node
The Executor’s Algorithm

readySet := ready sources in IPC Layer

timer ready?

yes
no

topic in readySet?

yes
no

service in readySet?

yes
no

service reply in readySet?

execute highest-priority callback
remove callback from readySet

no

no

no
The Executor’s Algorithm

- It significantly differs from usual schedulers
- ROS maintains a set (i.e., at most one instance per time) of ready callbacks
- When the set is empty (polling point), it queries the IPC layer for new activations
  - This can create priority inversion effects
- Except timers, up to ROS 2 “Dashing”
- It creates two levels of priority:
  - The first level of priority is given by the callback type (timer, subscription, service, service reply, in this order)
  - The second level of priority is given by the callback registration order in the ROS program
Compositional Performance Analysis

Per-Task Analysis
Computes per-task response times given event arrival curves

Arrival-Curve Propagation
Computes event arrival curves given per-task response times

The CPA approach fits ROS well
The CPA approach fits ROS well

Per-Callback Analysis

Needs to account for ROS’s quirks

Fixed-point search

Arrival-Curve Propagation

ROS-specific improvements in the RTSS 2021 paper
Response-Time Analysis – ECRTS 2019

\[
sbf_k(A + R_i^k(A)) = rbf_i(A + 1) + RBF(hp_k(c_i), A + R_i^k(A) - e_i + 1) + B_i
\]

\[
sbf_k(A + R_i^k(A)) = rbf_i(A + 1) + RBF(\{C_k \setminus c_i\}, A + R_i^k(A) - e_i + 1)
\]

Dedicated analyses for timers and polling-point-based callbacks

Response-Time Analysis that accounts for the ROS’s peculiarities

Optimization for intra-executor chains
We **improve** upon existing response-time analyses with three techniques.

1. **Address large execution-time variance** over time

2. **Exploit starvation-freedom** in the callback scheduler

At most one instance of each callback can run in a processing window

3. **Improve activation-curve propagation within executors**

![Activation curve diagram](image)

\[
\eta_1(\Delta) \quad \eta_1(\Delta + R_1) \quad \eta_1(\Delta + R_1 + R_2)
\]
Evaluation

- **Turtlebot 3** “Burger” controlled by a **Raspberry Pi 4B**

- Running various **ROS packages**
  - Navigation 2 packages
  - Turtlebot 3 drivers

- Callback graph extracted from measurements
  - See Blass et al., “Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems”, RTAS 2021
Evaluation
Evaluation

High-variance callback ⇒ Large gains from execution-time curves
Evaluation

Shares executor with bursty callback
⇒ Large gains from exploiting starvation-freeness
Evaluation

Gains over baseline thanks to improved arrival curve propagation
ROS 2 – Open Problems

- Real-Time Analysis of the multi-threaded executor of ROS 2
- Real-Time Analysis of the (many) custom executors that have been developed over the last years (see https://www.apex.ai/roscon-21)
- Scheduling of ROS 2 and DDS threads in a coordinate fashion (e.g., using QNX APS)
- Development of new special-purpose and real-time friendly executors
- And many more...
The opportunity of using QNX

- Preferred base operating system by many automotive OEMs
- ISO-26262 certified at the highest level of assurance (ASIL-D)
- POSIX Compliant, commercial, proven: AUTOSAR Adaptive needs POSIX
- Commercial OS supporting CPU reservations!

Interestingly, the QNX reservation-based scheduler supports multiple threads in the same reservation, thus offering opportunities for the coordinate scheduling of ROS and DDS threads.

However, it is first necessary to understand the behavior of the QNX reservation-based scheduling algorithm.
QNX Adaptive Partitioning Scheduler

How does APS work?

➢ To describe the behavior of APS partitions, we presented a set of rules
  ➢ They have been derived by relying both on the QNX documentation, and by performing proper validation experiments to corroborate our findings with empirical evidence (more about this later in the presentation)
  ➢ Validation results have been checked by collecting scheduling traces

➢ We considered two cases:
  ➢ Budget reclaiming is disabled, which can be configured with the SCHED_APS_SCHEDPOL_LIMIT_CPU_USAGE
  ➢ Budget reclaiming is enabled and idle time is distributed in priority order, which is the default option in QNX (SCHED_APS_SCHEDPOL_DEFAULT)

➢ Several rules have been derived, to define how budget is initialized, decremented, incremented, etc.
To analyze APS, we provided three contributions:

1. A supply bound function which lower-bounds the minimum service provided by a partition in any time interval of length $\Delta$.

2. A response-time analysis for the threads running in each partition.

3. A condition to guarantee that each partition can correctly deliver the supply to pending tasks, i.e., to guarantee that the core is not overloaded.

More details in the paper.
Experiments

- Setup on the real platform
  - QNX Software Development Platform 7.1
  - Raspberry Pi 4b (4 cores, 4GB RAM)
- Recording Traces
  - QNX Tracelogger to record scheduling events
  - Custom logger to record APS trace
    - Granularity 1ms
    - Assigned to dedicated core
- Custom tool to evaluate the recorded traces
- Analysis implemented on top of PyCPA

![Diagram showing experiment setup and process](Image)
Many, as usual, but let’s discuss a particular one:

**The design of tools for the design-space exploration of the system parameters using this analysis.**

**QNX APS Configuration Algorithm**

**INPUT:**
- Default System Model Params, End-to-end deadline

**Other constraints**
- (e.g., thread task must be allocated on core 2)

**Timing Constraints Satisfied?**
- YES
- NO

**Feasible solution?**
- YES
- NO

**Derive new parameters**
- (budgets, periods, thread to partition assignment, etc)

**OUTPUT:**
- Feasible System Model Parameters
Conclusions
Conclusions

- Autonomous driving is incredibly complex – much work still need to be done

- This is due to unpredictability issues occurring at multiple levels – from the application to the hardware

- Redesigning such systems to be time-predictable would be the best, but it is not an easily viable option because they are already widely used and provides a large amount of pre-implemented functionalities

We should try our best to improve their time-predictability by acting at different levels.
Are you interested in these topics?

- Join the RAGE 2022 workshop at DAC 2022! (see https://rage2022.github.io/)
- Advanced rate registration ends June 10th
- Organized by me, Dakshina Dasari (Bosch), and Matthias Becker (KTH)
Thank you!

Daniel Casini
daniel.casini@santannapisa.it