AutonomROS: A ReconROS-based Autonomous Driving Unit

Christian Lienen*, Mathis Brede†, Daniel Karger‡, Kevin Koch§, Dalisha Logan¶,
Janet Mazur‖, Alexander Philipp Nowosad**, Alexander Schnelle††, Mohness Waizy‡‡, and Marco Platzner[x]
Department of Computer Science
Paderborn University
Germany
Email: *christian.lienen@upb.de, †mbrede@mail.uni-paderborn.de, ‡dkarger@mail.uni-paderborn.de,
§kevink2@mail.uni-paderborn.de, ¶dalisha@mail.uni-paderborn.de, ‖mazurj@campus.uni-paderborn.de,
**anowosad@mail.uni-paderborn.de, ††aschnell@mail.uni-paderborn.de, ‡‡waizy@mail.uni-paderborn.de, [x]platzner@upb.de

***Abstract*—Autonomous driving has become an important research area in recent years, and the corresponding system creates an enormous demand for computations. Heterogeneous computing platforms such as systems-on-chip that combine CPUs with reprogrammable hardware offer both computational performance and flexibility and are thus interesting targets for autonomous driving architectures. The de-facto software architecture standard in robotics, including autonomous driving systems, is ROS 2. ReconROS is a framework for creating robotics applications that extends ROS 2 with the possibility of mapping compute-intense functions to hardware.**

This paper presents AutonomROS, an autonomous driving unit based on the ReconROS framework. AutonomROS serves as a blueprint for a larger robotics application developed with ReconROS and demonstrates its suitability and extendability. The application integrates the ROS 2 package Navigation 2 with custom-developed software and hardware-accelerated functions for point cloud generation, obstacle detection, and lane detection. In addition, we detail a new communication middleware for shared memory communication between software and hardware functions. We evaluate AutonomROS and show the advantage of hardware acceleration and the new communication middleware for improving turnaround times, achievable frame rates, and, most importantly, reducing CPU load.

***Index Terms*—Robotics, FPGA, ROS 2, ReconROS**

I. Introduction

Autonomous driving is a rapidly evolving and important area of research [1], which has the potential to fundamentally impact our transportation systems by, for example, significantly reducing traffic accidents [2]–[5], congestion [4]–[6], and carbon dioxide emissions [4], [7], [8].

One of the major technical challenges that could impede the widespread adoption of autonomous driving is the high computational cost involved [9], [10]. The computing demands for autonomous driving are exceptionally rigorous. Vehicles must continuously sense, process, and analyze their environment in real-time, make decisions, and execute them reliably and precisely. Providing the required computational performance is expensive, and the resulting costs could make autonomous vehicles unaffordable for many people, especially as the technology is still in its early stages of development [4]. Additionally, the necessary electrical power needed for the computations can be substantial [10]. This poses a particular problem for electric vehicles and can result in reduced ranges or larger and heavier batteries. Moreover, the more complex autonomous driving systems become, the less reliable they become, and system failures may occur [11], [12].

One promising technology that addresses the computational cost challenge of autonomous driving is reconfigurable hardware. Reconfigurable hardware devices such as field-programmable gate arrays (FPGAs) comprise programmable logic and memory blocks and allow for exploiting parallelism in computations on several levels. Many modern FPGAs are systems-on-chip, including dedicated processors and even embedded GPUs. Compared to microprocessors, FPGAs exhibit massive parallelism, making them ideal for computationally intensive functions such as those seen in autonomous driving, e.g., [13], [14], object detection [15], [16] or localization [17]. Moreover, FPGAs can be highly energy-efficient [18]–[20]. Compared to application-specific integrated circuits (ASICs), the key advantage of FPGAs is their flexibility, as they can be reprogrammed for different functions.

Our work focuses on making hardware acceleration based on FPGAs available for robotics systems. Supporting widely used software architectures and programming abstractions is of utmost importance for the successful adoption of reconfigurable hardware technology. The standard software framework for modern robotics applications is ROS, the robot operating system, or ROS 2 [21]. At the architecture level, ROS foresees the functional decomposition of an application into nodes, which are then linked via many-to-many publish-subscribe communication with ROS topics or one-to-one communication paradigms leveraged for ROS services and ROS actions.

In the last years, several approaches for integrating reconfigurable hardware into ROS-based applications have been proposed [22]–[26]. One of these approaches is our own development ReconROS [27], which allows for mapping complete ROS nodes to hardware while preserving the software programming model for both hardware and software nodes.

This paper presents AutonomROS[1], an autonomous driving

[1]https://github.com/Lien182/AutonomROS

unit based on ReconROS. AutonomROS serves as a blueprint for a more extensive ReconROS-based application combining state-of-the-art open-source ROS 2 packages, custom-developed ROS 2 software nodes, and ROS 2 nodes completely mapped to hardware. In particular, we make the following novel contributions:

- We present a new zero-copy communication middleware for ReconROS, based on Iceoryx, that greatly improves the performance of shared-memory inter-process communication between hardware and software nodes.
- We demonstrate hardware acceleration for the functions point cloud computation, obstacle detection, and adoption lane following. We show that AutonomROS is infeasible on the selected system-on-chip without hardware acceleration.
- We show the suitability of ReconROS for developing larger robotics applications comprising existing software packages, ROS 2 software nodes, and nodes accelerated in hardware.

The remainder of the paper is structured as follows: Section II provides background about ROS 2-based architectures for autonomous driving and ReconROS. Section III presents our new and efficient shared memory communication layer for ReconROS. In Section IV, we elaborate on the architecture of AutonomROS and its hardware-accelerated components. Section V presents an experimental evaluation, and Section VI concludes the paper and sketches ideas for feature work.

II. Background

In recent years, there has been much research in the field of architectures for autonomous driving. In this section, we first briefly introduce some relevant projects for autonomous driving and then provide background on ReconROS, the framework used by AutonomROS.

A. Autonomous Driving Architectures

In 2015, the *Autoware Foundation* started the *Autoware.ai* project, which aimed to propose a functional software architecture for autonomous driving based on ROS 1 as an open-source project. Later, *Autoware.ai* was transferred to the *Autoware.auto* project [28], now based on ROS 2, that provides a complete system stack for autonomous driving, including, e.g., functions for perception, planning, and control. Currently, *Autoware.auto* focuses on parking scenarios. The integration of FPGAs and the switch to a heterogeneous compute platform was shown in a follow-up project started by one of the *Autoware.ai* founders [29]. Besides the architecture, the follow-up project also includes the specification and development of compilers, the operating system, middleware, and applications.

In another work, Reke et al. [30] proposed a self-driving architecture on ROS 2 that focuses on safe and reliable real-time behavior while preserving the main advantages of ROS, e.g., standardized message formats and distributed architecture. However, the proposed architecture does not include hardware acceleration and requires a desktop-class CPU for execution.

Several FPGA-based architectures for autonomous driving were presented in, for example, [15], [31]–[35]. All these architectures have in common that they are designed for a rather limited scenario and do not rely on ROS 2 and state-of-the-art packages for autonomous driving, such as Navigation 2.

Additionally, there is work dealing with implementing and optimizing individual components of an autonomous driving architecture. Examples are provided in [36], [37]. In [36], the authors present a lane line detection component for autonomous driving running on an embedded GPU. After color space transformation and filtering, the street lane is approximated by polynomial regression. The work is limited to the vision system of the autonomous driving architecture. The authors of [37] describe a hardware-accelerated implementation of a lane detection algorithm on a Zynq-7000 System-on-Chip. The algorithm relies on edge detection on the input camera image, followed by a Hough transformation. Again, this work only considers the lane detection component.

B. ReconROS

ReconROS [27], [38] is a framework for implementing ROS 2 applications on heterogeneous compute platforms comprising a multi-core CPU and a reconfigurable hardware fabric and accelerating robotics application components on the reconfigurable hardware. ReconROS combines the ReconOS reconfigurable hardware operating system [39] with ROS 2.

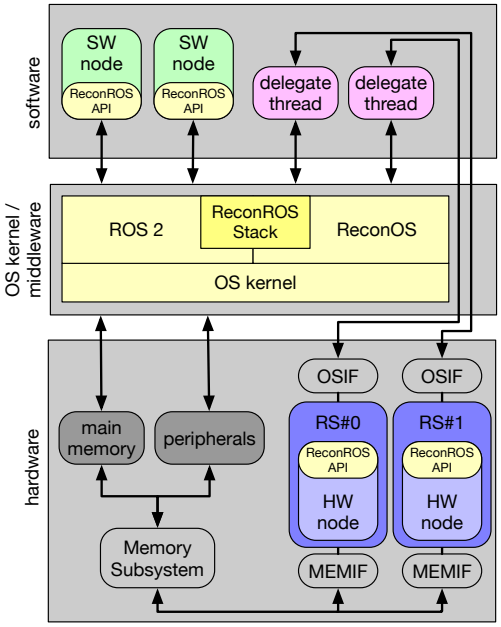

Fig. 1. ReconROS Architecture (from [40])

Figure 1 sketches the hardware/software architecture of ReconROS. A ROS 2 application comprises a set of hardware nodes (HW node) and software nodes (SW node). HW nodes are assigned to specific rectangular regions on the reconfigurable fabric denoted as reconfigurable slots (RS). Each reconfigurable slot connects to the operating system kernel

via an operating system interface (OSIF) and a lightweight software thread, the delegate thread. The delegate carries out operating system calls on behalf of the hardware node. Further, each reconfigurable slot connects to shared memory via a memory interface (MEMIF). This way, hardware nodes can access the shared virtual memory space, including any memory-mapped peripherals.

ReconROS extends the Linux operating system and enables the execution of ROS 2 nodes in hardware and software through the ReconROS stack and the ReconROS API. The ReconROS stack expands the capabilities of ReconOS by introducing ROS-related objects, such as ROS publishers or subscribers. Therefore, hardware and software nodes can interact with other ROS 2 nodes leveraging regular ROS 2 communication paradigms. Hence, communication is standardized, and integrating hardware-accelerated components into an existing ROS 2 application is greatly facilitated.

The ReconROS API is available for both software and hardware nodes and provides a consistent programming model across the software/hardware boundary. Hardware nodes can even be assigned to specific reconfigurable slots at runtime, allowing for dynamic mapping of a ROS application to nodes running in software and hardware.

When a hardware node wants to leverage ROS 2 communication, it starts the interaction by sending a corresponding command via its OSIF to its delegate thread. Examples of such commands are a subscription to a ROS 2 topic or a request for a ROS 2 service. The delegate thread relays the command to the ReconROS stack and blocks until lower ROS 2 layers complete the processing of the requested command. When a message is received, the delegate is unblocked and transmits the location of the message in the main memory through the OSIF to the hardware node. The hardware node can then access the message through its MEMIF.

III. ReconROS Shared-Memory Communication

A critical improvement of ROS 2 over ROS 1 was the introduction of an exchangeable communication layer based on well-established data distribution services (DDS). A developer can select between various available DDS implementations with different properties. Several DDS implementations rely on standard sockets as the default communication mechanism. Sockets are the most flexible option and enable both intra-platform and inter-platform communication. When intra-platform or even inter-process communication is required, a loopback adapter is employed to transfer data to the same or other processes. This flexibility is paid for with lowered performance since the loopback mechanism results in overheads due to several data copy operations involved.

To mitigate such overheads, the Iceoryx [41] communication middleware for ROS 2 was introduced. Iceoryx is an intra-process communication middleware enabling zero-copy data transmission between processes on the same platform. The disadvantage of Iceoryx, however, is that it comes with significant limitations, e.g., there is no support for ROS 2 services and actions. Since many larger software packages for ROS 2, e.g., Navigation 2, rely on these communication paradigms, the field of application for Iceoryx would be limited. Fortunately, Iceoryx is also part of the CycloneDDS middleware that allows for simultaneously using socket-based and shared-memory-based communication. When selecting shared memory for communication, the topic to which the ROS 2 nodes publish and subscribe has to adhere to the following constraints: (i) the message has a fixed length, (ii) a suitable QoS configuration is selected, (iii) the topic has at most 127 subscriptions, and (iv) a publisher has at most eight loaned messages simultaneously.

We have chosen the CycloneDDS middleware, with the integrated Iceoryx, as the basis for AutonomROS. Since Iceoryx requires a slightly different programming model than standard ROS 2 communication, we had to extend ReconROS to support the zero-copy communication scheme of Iceoryx. When using Iceoryx, a publishing node must first request a memory chunk from the middleware for communication with other nodes. The publishing node can write its message into the received memory chunk and execute the corresponding publishing function call if successful. Similar to standard ROS 2 subscribers, the subscribing node can block for a new message. However, after receiving it, the subscriber returns the message to the middleware to enable the re-usage of the message chunk.

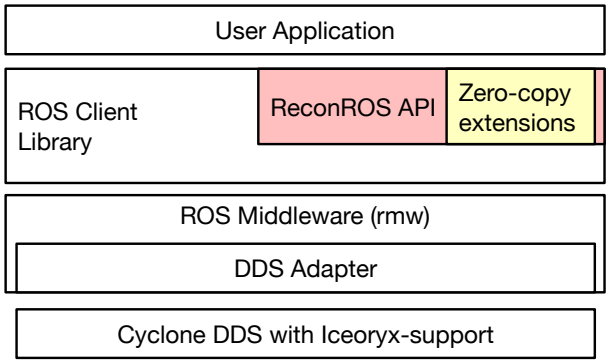

Fig. 2. ReconROS API (red) extended by operations for zero-copy data transfers between nodes (yellow)

Figure 2 shows the resulting extensions in the ReconROS API. Overall, we extended ReconROS by four function calls: `ROS_BORROW` requests a memory chunk from Iceoryx, `ROS_PUBLISH_LOANED` publishes the message after the message has been written to the message chunk, `ROS_SUBSCRIBE_TAKE_LOANED` tries to read a message, and `ROS_SUBSCRIBE_RETURN_LOANED` returns the read message to Iceoryx so that the memory chunk can be re-used.

IV. Autonomous Driving Unit Architecture

Figure 3 shows the top-level overview of the architecture of the AutonomROS autonomous driving unit. Besides providing its indented functionality, the architecture also aims to demonstrate that our ReconROS framework for creating robotics applications enables efficient hardware acceleration while maintaining the programming abstractions of ROS 2 and the ability to integrate larger ROS 2 software packages.

AutonomROS comprises seven main components: The *Obstacle Detection* component and its preprocessing component *Point Cloud Generation* detect obstacles in front of the car, and the *Lane Detection* analyzes lanes. The *Navigation Stack* subsequently uses this information, which sets commands for steering control and the desired speed. The *Localization* component fuses different external sensors, e.g., inertial measurement units or wheel encoders, that provide information about the actual movement and the current position based on a static map. The *Vehicle Communication* component handles communication with the infrastructure around the car, e.g., a traffic light controller. It uses information about the vehicle's actual position from the *Localization* component to determine if the car is approaching an intersection. An entrance request is sent if the vehicle wants to enter an intersection. Eventually, the vehicle is allowed to enter the intersection. This permission is provided to the *Navigation Stack* component. The *Cruise Control* component controls the car's speed in a control loop leveraging a PID controller. The reference value of the control loop is the desired speed from the *Navigation Stack*. The difference between this reference value and the actual speed from the *Localization* component serves as the measured error for the PID controller. The output of the controller is forwarded to the engine of the vehicle.

Three of the components, *Point Cloud Generation*, *Obstacle Detection*, *Lane Detection*, show high computational demands with significant amounts of data processed and are thus suitable for hardware acceleration. We discuss these components in more detail on the algorithmic level in Subsections IV-B, IV-C, and IV-D. These components are developed in C/C++ and can either be compiled using GCC for software execution, or synthesized with Xilinx Vitis HLS for hardware execution. Except for the communication with the traffic light, all communication is realized using standard ROS 2 publish-subscribe communication. The communication between the car and traffic lights relies on MQTT (Message Queuing Telemetry Transport).

Three more components, *Localization*, *Vehicle Communication*, and *Cruise Control*, are custom-designed for AutonomROS and mapped to ROS 2 software nodes. Finally, the component *Navigation Stack* is based on Nav2 (Navigation 2), an open-source ROS 2 package. We elaborate on this component in Subsection IV-A.

A. Navigation Stack

Figure 4 details the *Navigation Stack* component. It consists of the components *Behavior*, *Planner*, *Global Map*, *Controller*, and *Local Obstacle Map*. The whole component uses localization information for position estimation, which is crucial for all its sub-components. The *Behavior* component defines how the car should behave in different situations. It receives a goal position, which represents to which location the vehicle should drive. Additionally, it uses the intersection information to decide whether the car must stop in front of an intersection. These decisions result in different control commands for the *Planner* and *Controller*. Among others, the

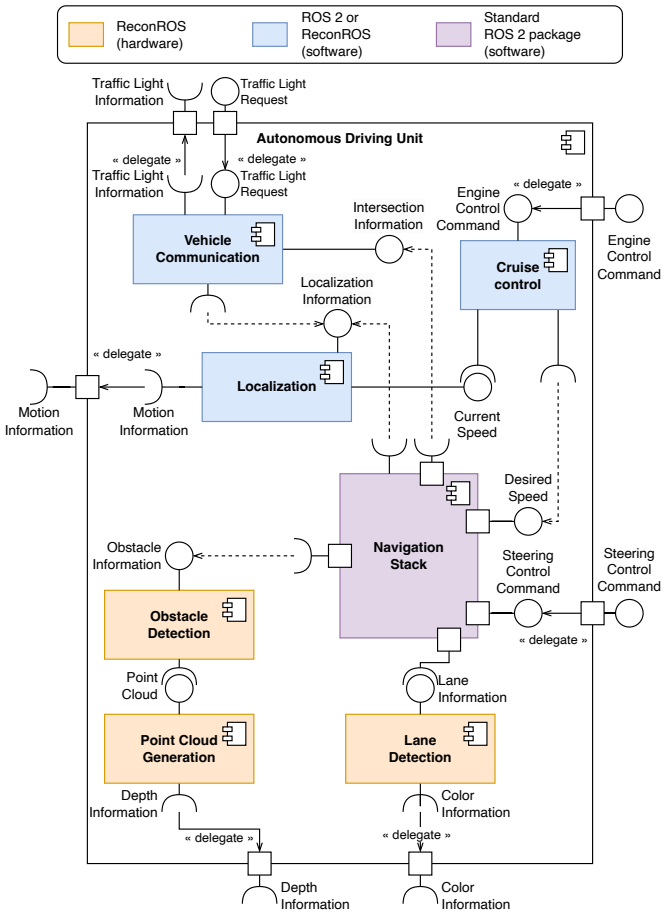

Fig. 3. Architecture of the AutonomROS autonomous driving unit. Hardware-accelerated components are highlighted in blue color

Planner receives a goal position that the car should reach from the *Behavior* component. The *Planner* calculates, based on static map information from the *Global Map* and the car's current position, a path from the current position to the goal position. This path is then handed to the *Controller*. The *Controller* component also receives control commands from the *Behavior* component. These commands include whether the car should follow the path, lane, or stop because an intersection is blocked. The component calculates a steering control command and the desired speed based on the path or lane information. Additionally, the *Controller* uses information from a local map from the *Local Obstacle Map* to check whether an obstacle is in the way, and if so, stops the car.

B. Point Cloud Generation

The *Point Cloud Generation* component receives depth information from an external sensor, e.g., a 3D camera, and calculates a point cloud from the depth and corresponding color images. The *Obstacle Detection* component uses the resulting point cloud to detect obstacles in front of the car. Calculating a 3D point cloud involves analyzing and processing a depth image to generate a comprehensive representation of a physical object or environment in 3D. For point cloud

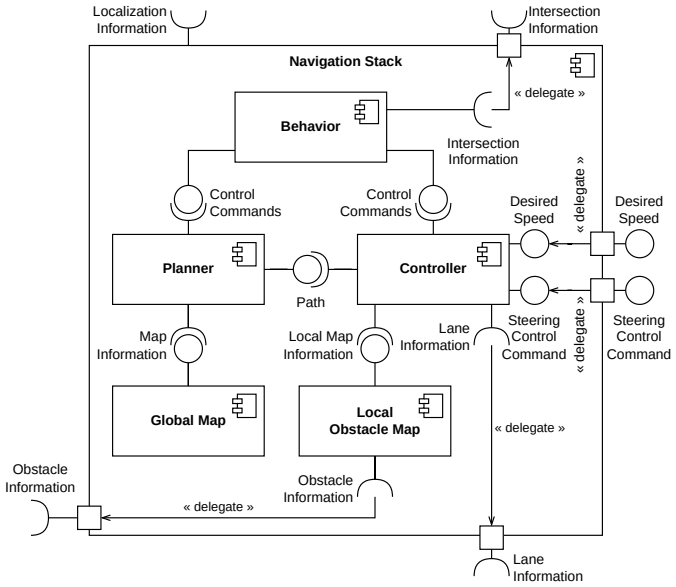

Fig. 4. AutonomROS Navigation Stack

computation, the first step is to merge the depth and the color image. Since pixels in the depth image are independent, this task can be easily parallelized and is ideally suited for hardware acceleration with ReconROS.

Our hardware implementation of the component initially receives the camera's projection matrix P as shown in Equation 1. This matrix is needed to transform pixels from the depth image into the 3D world. It comprises the focal lengths (fx, fy), the principal point (cx, cy), and information about the relative position of the second camera to the first (Tx, Ty) [42]. The matrix is published as a ROS 2 `CameraInfo` message by the camera's wrapper node. We need to gather this matrix only once before the actual runtime loop starts because the matrix does not change as long as the camera is not switched.

$$P = \begin{bmatrix} fx & 0 & cx & Tx \\ 0 & fy & cy & Ty \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1}$$

In the runtime loop, we transform all pixels of an incoming image with their coordinates x, y, and depth w to their 3D world coordinates X, Y, and Z. To this end, we first determine intermediate variables $u = x \cdot w$ and $v = y \cdot w$ and then apply Equations 2 - 4, which are taken from the description of the `CameraInfo` message of ROS [42].

$$X = \frac{u - cx \cdot w - Tx}{fx} \tag{2}$$

$$Y = \frac{v - cy \cdot w - Ty}{fy} \tag{3}$$

$$Z = w \tag{4}$$

C. Obstacle Collision

Obstacle detection is typically done by computing a cost map layer based on the generated point cloud. Within the Navigation 2 package, a so-called Voxel Layer constitutes the default cost map layer. However, the Voxel Layer sequentially iterates over every point in the point cloud, which is slow. Thus, we have decided to replace the Voxel Layer by (i) processing the obstacle detection in hardware and (ii) handling the resulting data in the Navigation 2 package by a customized cost map layer.

The process of converting the point cloud into an obstacle grid is split into the following four steps: First, we transform the image from the camera's coordinate system into the car's base coordinate system based on a fixed transformation matrix. Second, we select all points in a predefined volume in front of the car and consider only the points in this "obstacle box" in the following steps. Since points higher than the car and points far away from the front or the sides of the vehicle need not be considered, this selection helps save on computations. Third, we project the points within the obstacle box to the ground, i.e., to the xy-plane. Finally, we discretize the obstacle box to a grid and assign to each grid cell the number of points of the original point cloud that map to the cell. The discretization reduces the required memory for representing obstacles from 4.7 MB for the point cloud to 234 Byte for the grid. In addition, the discretization reduces noise from the camera's depth sensor, which could result in false-positive detections of obstacles.

Most of the involved processing is performed pixel-parallel. The final grid is published to a ROS 2 topic to make it usable for the custom cost map layer in the Navigation 2 package that runs in the software.

D. Lane Detection

The *Lane Detection* component includes multiple computationally expensive image processing steps, e.g., the perspective and color thresholding transformation. Our component implementation involves several steps and relies on the open-source Vitis Image processing library[2].

The first step transforms the incoming image from the RGB to the HSV color space. The transformation supports processing different color values independent of the environment's saturation or lighting. The next step applies color thresholding to identify the image's white and yellow areas. Thresholding for both colors is processed in parallel, resulting in one grayscale image representing yellow and white pixels in the original image. The following step performs a warp transformation to the grayscale image based on a fixed transformation matrix to get a bird's view of the represented scene. After warp transformation, a decision is taken considering the number of pixels for each of the two colors, whether to follow the street's white or yellow lane. Subsequent processing focuses on the selected color.

[2]https://github.com/Xilinx/Vitis_Libraries.git

Then, we perform a polynomial least-squares regression on the lane to eliminate interfering falsely detected pixels and establish an equation for the lane marking. Based on the N pixels with coordinates (x_n, y_n) that represent the lane and k as the polynomial order to fit, we compute the desired polynomial coefficients $\vec{a} = [a_0 \dots a_k]^T$ by solving the equations system as shown in Equation 5.

$$\begin{bmatrix} N & \dots & \sum_{k=0}^{N} x_n^k \\ \vdots & \ddots & \vdots \\ \sum_{n=0}^{N} x_n^k & \dots & \sum_{k=0}^{N} x_n^{2k} \end{bmatrix} \cdot \vec{a} = \begin{bmatrix} \sum_{n=0}^{N} y_n \\ \vdots \\ \sum_{n=0}^{N} y_n x_n^k \end{bmatrix} \quad (5)$$

For implementing the *Lane Detection* component, we have determined that the second-order polynomial shown in Equation 6 is suitable.

$$f_l(x) = a_2 \cdot x^2 + a_1 \cdot x + a_0 \quad (6)$$

To estimate the final trajectory, the resulting polynomial has to be shifted to the middle of the image and transformed into the car's base coordinate system.

For more efficient computation of the final trajectory, we use 30 equally-distributed points along the height of the image ($x_i = i \cdot 480/30$, $i = 0..29$) and compute 30 function values $y_i = f_l(x_i)$ (Equation 6).

Using the resulting 30 coordinates (y_i, x_i), the equation system 5 is solved again for a target polynomial function $f_t(x)$ (Equation 7).

$$f_t(x) = a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0 \quad (7)$$

V. Evaluation

In this section, we first report on the evaluation setup, including a real-world model car used for driving experiments to test the functionality of the AutonomROS driving unit. Then, we present architecture exploration experiments to evaluate the performance of different DDS versions and the hardware acceleration.

A. Evaluation Setup

We execute AutonomROS on a Zynq UltraScale+ MPSoC ZCU104 evaluation board. The board contains a system-on-chip architecture with a quad-core ARM Cortex-A53, a dual-core Cortex-R5 real-time processor, a Mali-400 MP2 embedded graphics processing unit, and programmable logic (PL). For our evaluations, we used the quad-core CPU and the programmable logic. The board runs Ubuntu 20.04 in combination with ROS 2 galactic and ReconROS.

We have mounted the evaluation board on a model car platform shown in Figure 5. The model car platform is based on a modified commercial remote-controlled car in which the control and sensor systems have been replaced. The actuators for the steering and the drive were preserved. Regarding sensors, the platform includes two cameras, one for color information and one 3D camera providing depth and color information, an inertial measurement unit (IMU) for measuring acceleration data, and a wheel encoder for gaining speed data. Regarding actuators, the platform exhibits interfaces to drive the engine and the car's steering. Further, the evaluation board is equipped with a wireless LAN interface for data exchange with other vehicles and the infrastructure, e.g., the traffic lights controller. We have set up a $5m \times 5m$ grid of streets with two intersections to mimic real-world environmental test conditions. Additionally, we have set up a central infrastructure server that acts as a traffic lights controller and handles requests for crossing intersections. We have built two model cars to evaluate the AutonomROS functionality involving multiple vehicles.

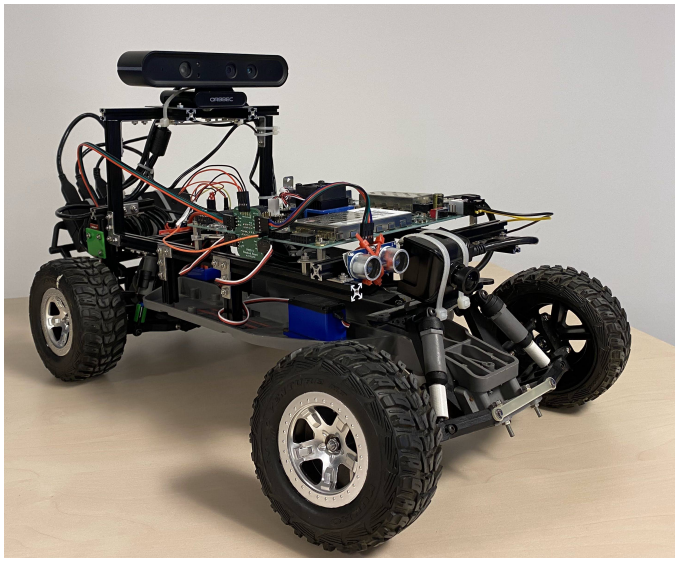

Fig. 5. Model car platform

B. Performance Evaluation

Table I summarizes the results of the performance measurements for the hardware-accelerated components *Point Cloud Generation*, *Obstacle Detection*, and *Lane Detection*. For these processing components, the test setup includes two ROS 2 nodes, one node publishing camera data, and one node including the actual processing. The table lists different architecture configurations, the total CPU load in % of four cores, the achieved frames per second (FPS), and the turnaround time. The turnaround time represents the node's raw computation time, which, in contrast to the FPS metric, is not limited by the input signal rate.

Point Cloud Generation: Two are two main observations: First, using the zero-copy Iceoryx middleware is highly beneficial, as expected, for both software and hardware mappings of the component. The CPU load and the achieved FPS are improved by roughly $2\times$, and the turnaround time decreased by $7.5\times$ compared to standard ROS 2 communication. Adding hardware acceleration without a zero-copy communication middleware gives a minimal advantage. Second, using Iceoryx and hardware acceleration reduces the CPU load further, while the achieved FPS is bound by the camera's maximum frame

Component	Architecture configuration	CPU[a]	FPS	Turnaround time (ms)
Point Cloud Generation @640x480 depth image	ReconROS SW without Iceoryx	180–220	9–12	83–111
	ReconROS HW without Iceoryx	180–210	13–14	71–76
	ReconROS SW with Iceoryx	90–110	29–30	11–15
	ReconROS HW with Iceoryx	64–78	29–30	17–18
Obstacle Detection	ReconROS SW with Iceoryx	50–60	29–30	15–17
	ReconROS HW with Iceoryx	4–8	29–30	9–11
Lane Detection	ROS 2 SW with Iceoryx	170–190	29–30	27–31
	ReconROS HW with Iceoryx	24–38	29–30	9–18

[a]CPU load in % of 4 cores

TABLE I
PERFORMANCE MEASUREMENTS FOR THE HARDWARE-ACCELERATED COMPONENTS OF AUTONOMROS. THE TABLE SHOWS MIN/MAX VALUES COLLECTED IN MULTIPLE MEASUREMENTS.

rate of around 30 frames per second. The turnaround time is slightly higher because of overheads for data transmission into the programmable logic.

Obstacle Detection: Here, we compare the software and hardware configurations with Iceoryx. Both configurations achieve the maximum FPS. For the hardware-mapped node, the CPU utilization of the component decreases significantly by a factor of $12.5\times$, and the turnaround time is about $1.6\times$ lower compared to the implementation in software.

Lane Detection: We compare a ROS 2 software implementation with a ReconROS hardware implementation. Compared to ReconROS SW implementation, this node relies on a standard ROS 2 C++ implementation, including the ROS executor enabling the event-driven programming model. Again, hardware acceleration results in a significant reduction of the CPU load and turnaround times.

The main conclusion of the measurements reported in Table I is that for intra-platform communication, using a zero-copy communication middleware such as Iceoryx is of utmost importance to maintain performance. Moreover, effective hardware acceleration relies on such a middleware. The reported measurements are only for two ROS 2 nodes. Mapping the overall AutonomROS unit of Figure 3 entirely to software would exceed the maximum CPU utilization of 400 %. Thus, running the presented AutonomROS on the chosen system-on-chip platform is only possible with hardware acceleration.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented AutonomROS, a ReconROS-based unit for autonomous driving. AutonomROS integrates a ROS 2 open-source package for navigation with custom-developed software nodes for localization, cruise control, and vehicle communication and hardware-accelerated nodes for point cloud generation, obstacle detection, and lane detection. We have also introduced a new communication middleware for ReconROS that boosts the performance of shared memory communication between software and hardware nodes. Hardware acceleration combined with the new communication middleware results in significant performance improvements, measurable turnaround times, and achievable frame rates, but most pronounced in reduced CPU utilization with up to $12.5\times$ for selected ROS 2 nodes.

In feature work, we plan to extend the AutonomROS driving unit with more advanced functionality, such as ORB-SLAM and GPS for localization or Advanced Driver Assistance Systems (ADAS). The advanced functionality should increase the potential for hardware acceleration.

REFERENCES

[1] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE access*, vol. 8, pp. 58 443–58 469, 2020.

[2] M. Bertoncello and D. Wee, "Ten ways autonomous driving could redefine the automotive world," *McKinsey & Company*, vol. 6, 2015.

[3] I. Barabas, A. Todoruţ, N. Cordoş, and A. Molea, "Current challenges in autonomous driving," *IOP Conference Series: Materials Science and Engineering*, vol. 252, no. 1, p. 012096, 2017.

[4] I. Yaqoob, L. U. Khan, S. A. Kazmi, M. Imran, N. Guizani, and C. S. Hong, "Autonomous driving cars in smart cities: Recent advances, requirements, and challenges," *IEEE Network*, vol. 34, no. 1, pp. 174–181, 2019.

[5] S. Parida, M. Franz, S. Abanteriba, and S. Mallavarapu, "Autonomous driving cars: future prospects, obstacles, user acceptance and public opinion," in *Advances in Human Aspects of Transportation: Proceedings of the AHFE 2018 International Conference on Human Factors in Transportation, July 21-25, 2018, Loews Sapphire Falls Resort at Universal Studios, Orlando, Florida, USA 9*. Springer, 2019, pp. 318–328.

[6] P. Sun and A. Boukerche, "Challenges and potential solutions for designing a practical pedestrian detection framework for supporting autonomous driving," in *Proceedings of the 18th ACM Symposium on Mobility Management and Wireless Access*, 2020, pp. 75–82.

[7] U. Montanaro, S. Dixit, S. Fallah, M. Dianati, A. Stevens, D. Oxtoby, and A. Mouzakitis, "Towards connected autonomous driving: review of use-cases," *Vehicle system dynamics*, vol. 57, no. 6, pp. 779–814, 2019.

[8] P. Kopelias, E. Demiridi, K. Vogiatzis, A. Skabardonis, and V. Zafiropoulou, "Connected & autonomous vehicles–environmental impacts–a review," *Science of the total environment*, vol. 712, p. 135237, 2020.

[9] X. Zhao, P. Sun, Z. Xu, H. Min, and H. Yu, "Fusion of 3d lidar and camera data for object detection in autonomous vehicle applications," *IEEE Sensors Journal*, vol. 20, no. 9, pp. 4901–4913, 2020.

[10] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[11] J. Chen, B. Yuan, and M. Tomizuka, "Model-free deep reinforcement learning for urban autonomous driving," in *2019 IEEE intelligent transportation systems conference (ITSC)*. IEEE, 2019, pp. 2765–2771.

[12] H. Lu, Q. Liu, D. Tian, Y. Li, H. Kim, and S. Serikawa, "The cognitive internet of vehicles for autonomous driving," *IEEE Network*, vol. 33, no. 3, pp. 65–73, 2019.

[13] Y. Li, S. E. Li, X. Jia, S. Zeng, and Y. Wang, "Fpga accelerated model predictive control for autonomous driving," *Journal of Intelligent and Connected Vehicles*, vol. 5, no. 2, pp. 63–71, 2022.

[14] S. Du, T. Huang, J. Hou, S. Song, and Y. Song, "Fpga based acceleration of game theory algorithm in edge computing for autonomous driving," *Journal of Systems Architecture*, vol. 93, pp. 33–39, 2019.

[15] A. Kojima, "Autonomous driving system implemented on robot car using soc fpga," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2021, pp. 1–4.

[16] C. Hao, A. Sarwari, Z. Jin, H. Abu-Haimed, D. Sew, Y. Li, X. Liu, B. Wu, D. Fu, J. Gu *et al.*, "A hybrid gpu+ fpga system design for autonomous driving cars," in *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2019, pp. 121–126.

[17] Q. Liu, Z. Wan, B. Yu, W. Liu, S. Liu, and A. Raychowdhury, "An energy-efficient and runtime-reconfigurable fpga-based accelerator for robotic localization systems," in *2022 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 2022, pp. 01–02.

[18] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels," in *2019 IEEE international conference on embedded software and systems (ICESS)*. IEEE, 2019, pp. 1–8.

[19] B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, M. Xie, L. Wan, H. Liu, and C. Ding, "Ftrans: energy-efficient acceleration of transformers using fpga," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 175–180.

[20] D. Haripriya, K. Kumar, A. Shrivastava, H. M. R. Al-Khafaji, V. Moyal, and S. K. Singh, "Energy-efficient uart design on fpga using dynamic voltage scaling for green communication in industrial sector," *Wireless Communications and Mobile Computing*, vol. 2022, 2022.

[21] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, 2022.

[22] Y. Sugata, T. Ohkawa, K. Ootsu, and T. Yokota, "Acceleration of Publish/Subscribe Messaging in ROS-Compliant FPGA Component," in *Proc. of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART2017)*. ACM, 2017.

[23] T. Ohkawa, Y. Sugata, H. Watanabe, N. Ogura, K. Ootsu, and T. Yokota, "High Level Synthesis of ROS Protocol Interpretation and Communication Circuit for FPGA," in *Proc. 2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*, 2019, pp. 33–36.

[24] V. Mayoral-Vilches, S. M. Neuman, B. Plancher, and V. J. Reddi, "RobotCore: An Open Architecture for Hardware Acceleration in ROS 2," *arXiv preprint arXiv:2205.03929*, 2022.

[25] A. Podlubne and D. Göhringer, "FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs," in *Proc. International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019.

[26] M. Eisoldt, S. Hinderink, M. Tassemeier, M. Flottmann, J. Vana, T. Wiemann, J. Gaal, M. Rothmann, and M. Porrmann, "ReconfROS: Running ROS on Reconfigurable SoCs," in *Proc. 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*. ACM, 2021, p. 16–21.

[27] C. Lienen, M. Platzner, and B. Rinner, "ReconROS: Flexible Hardware Acceleration for ROS2 Applications," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 268–276.

[28] "autowautoware - the world's leading open-source software project for autonomous driving," https://github.com/autowarefoundation/autoware, accessed: 2023-08-30.

[29] H. Chishiro, K. Suito, T. Ito, S. Maeda, T. Azumi, K. Funaoka, and S. Kato, "Towards heterogeneous computing platforms for autonomous driving," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, 2019, pp. 1–8.

[30] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis, "A self-driving car architecture in ros2," in *2020 International SAUPEC/RobMech/PRASA Conference*. IEEE, 2020, pp. 1–6.

[31] Y. Nitta, S. Tamura, and H. Takase, "A study on introducing fpga to ros based autonomous driving system," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 421–424.

[32] R. Yamamoto, Y. Izumi, R. Aono, T. Nagahara, T. Tanaka, W. Liao, and Y. Mitsuyama, "Development of autonomous driving system based on image recognition using programmable socs," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2021, pp. 1–4.

[33] T. Tanaka, I. Ikeno, R. Tsuruoka, T. Kuchiba, W. Liao, and Y. Mitsuyama, "Development of autonomous driving system using programmable socs," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 453–456.

[34] E. Jones, K. Pepper, A. Li, S. Li, Y. Zhang, and D. Bailey, "Autonomous driving developed with an fpga design," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 431–434.

[35] T. Wu, W. Liu, and Y. Jin, "An end-to-end solution to autonomous driving based on xilinx fpga," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 427–430.

[36] J. A. B. Palma, M. N. I. Bonilla, and R. E. Grande, "Lane line detection computer vision system applied to a scale autonomos car: Automodelcar," in *2020 17th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, 2020, pp. 1–6.

[37] M. Gopinathan, R. Soundarrakumar, A. Kalaiselvi, A. Mohideen *et al.*, "Implementation of lane detection in autonomous vehicle using fpga," in *2022 International Conference on Emerging Trends in Engineering and Medical Sciences (ICETEMS)*. IEEE, 2022, pp. 141–147.

[38] C. Lienen and M. Platzner, "Design of Distributed Reconfigurable Robotics Systems with ReconROS," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 3, dec 2022.

[39] E. Lübbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, pp. 8:1–8:33, 2009.

[40] C. Lienen and M. Platzner, "Task mapping for hardware-accelerated robotics applications using reconros," in *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, 2022, pp. 148–155.

[41] "iceoryx - true zero-copy inter-process-communication," https://github.com/eclipse-iceoryx/iceoryx, accessed: 2023-08-30.

[42] "Camerainfo raw message definition," http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/CameraInfo.html, accessed: 2023-08-30.