ENABLING RECONFIGURABLE HARDWARE ACCELERATION FOR ROS-BASED ROBOTICS APPLICATIONS

DISSERTATION

A thesis submitted to the
FACULTY FOR COMPUTER SCIENCE, ELECTRICAL ENGINEERING AND MATHEMATICS
of
PADERBORN UNIVERSITY
in partial fulfillment of the requirements
for the degree of *Dr. rer. nat.*

by

CHRISTIAN LIENEN

Paderborn, Germany
Date of submission: December 2023

SUPERVISORS:
Prof. Dr. Marco Platzner

REVIEWERS:
Prof. Dr. Marco Platzner
Prof. Dr. Erdal Kayacan

ORAL EXAMINATION COMMITTEE:
Prof. Dr. Marco Platzner
Prof. Dr.-Ing. Roman Dumitrescu
Prof. Dr. Erdal Kayacan
Prof. Dr. Stefan Sauer
Dr. Tobias Kenter

DATE OF SUBMISSION:
December 2023

ACKNOWLEDGMENTS

This dissertation would not have been possible without the support of many people. I want to mention a few particularly noteworthy supporters at this point. First, my biggest gratitude goes to Prof. Dr. Marco Platzner, without whom I would not have been able to persuade myself to do this project. With his trust, patience, and professional advice, he creates an outstanding working environment that is second to none. In this context, I would also like to thank the other reviewer, namely Prof. Dr. Erdal Kayacan, for the time and effort in reviewing and evaluating this thesis. I would also like to thank Prof. Dr.-Ing. Roman Dumitrescu, Prof. Dr. Stefan Sauer and Dr. Tobias Kenter for their involvement in the oral examination committee.

Furthermore, I would like to thank all members of the computer engineering group at the University of Paderborn, who provided me with professional, administrative, and personal support. Special thanks go to my roommate Tim Hansmeier and Linus Witschen, who brightened up the coffee breaks with their humor and creativity. Additional thanks also go to all students who have made a valuable contribution through their bachelor's or master's project or their participation in the AutonomROS project group. I would like to mention my student research assistants, who have made an important contribution to this work.

From my personal environment, I would first of all like to thank my second half, Helena, who encouraged me to do this project and who had to endure my ups and downs during this project and accompanied me on my conference visits.

Finally, I would like to thank my parents and siblings, who have supported me unconditionally throughout my life and made this path possible for me. Thank you very much for your support in all situations!

ABSTRACT

The Robot Operating System (ROS) has recently become the de-facto standard for developing robotics applications comprising a design paradigm based on decomposition, communication infrastructure, and libraries. For the execution of these robotic applications, modern computing platforms comprise a set of heterogeneous computing platforms, e.g., multi-core CPUs, embedded GPUs, and FPGAs. Due to custom hardware architectures, FPGAs promise faster and more energy-efficient computation for many applications than related computing platforms such as multi-core CPUs and GPUs. Modern reconfigurable system-on-chip architectures combine a multi-core CPU with reconfigurable logic, resulting in a tighter composition of hardware and software and supporting more advanced functionality, such as dynamic partial reconfiguration. Therefore, an important research aspect is the standardized integration of reconfigurable hardware accelerators into existing ROS-based software applications since the complete redevelopment of projects is inefficient and usually not feasible due to their complexity. Other research aspects besides the standardized integration are, e.g., the support for dynamic partial reconfiguration for increased hardware utilization or the reduction of memory transfer overheads into the reconfigurable logic.

This thesis presents ReconROS, a novel approach for integrating reconfigurable hardware accelerators into robotics applications. The contribution of this thesis comprises the ReconROS framework, the ReconROS executor, and fpgaDDS. The ReconROS framework combines ReconOS and ROS 2, enabling robotics developers to implement ROS 2 nodes entirely in hardware. Due to its consistent programming model for hardware and software and full virtual memory access, developers can easily migrate nodes from software to hardware and vice versa. The ReconROS executor extends the concept of event-driven programming using callbacks to hardware. For that, it introduces hardware callbacks executed after a specific event. The ReconROS executor includes infrastructure for dynamic partial reconfiguration in hardware and the scheduling and placement of callbacks. As the third contribution, this thesis proposes fpgaDDS, an intra-FPGA data distribution layer for hardware-mapped ROS 2 nodes. All three contributions achieve speedups in execution time and improved jitter compared to their pure-software counterparts. Lastly, this thesis presents three more extensive application examples benefiting from ReconROS and hardware acceleration.

ZUSAMMENFASSUNG

Das Robot Operating System (ROS) hat sich in letzter Zeit zum de-facto-Standard für die Entwicklung von Software im Bereich Robotik entwickelt und beinhaltet ein Designparadigma basierend auf Dekomposition, einer Infrastruktur zur Kommunikation und Bibliotheken. Für die Ausführung von Robotik Anwendungen bestehen moderne Computerplattformen aus einer Reihe heterogener Plattformen wie z.B. Multi-Core-CPUs, eingebettete GPUs und FPGAs. Durch angepasste Hardwarearchitekturen verspricht die Nutzung von FPGAs eine schnellere und energieeffizientere Berechnung als verwandte Plattformen wie Multi-Core-CPUs und GPUs. Moderne System-on-Chip Architekturen kombinieren eine Multi-Core CPU mit rekonfigurierbarer Logik, was in einer dichteren Komposition von Hardware und Software und die Unterstützung von partieller Rekonfiguration resultiert. Ein wichtiger Aspekt ist daher die standardisierte Integration von Hardwarebeschleunigern in bestehende ROS-basierte Softwareanwendungen, da die komplette Neuentwicklung von Anwendungen ineffizient und aufgrund ihrer Komplexität meist nicht realisierbar ist. Weitere Herausforderungen neben der standardisierten Integration sind die Nutzung einer effizienteren Ressourcennutzung durch dynamische partielle Rekonfiguration und die Verringerung des Overheads durch Datentransfers zwischen Hardware und Software.

In dieser Arbeit wird ReconROS vorgestellt, ein neuartiger Ansatz zur Integration rekonfigurierbarer Hardware in Robotik Anwendungen, der das ReconROS-Framework, den ReconROS Executor und fpgaDDS umfasst. Das ReconROS Framework kombiniert ReconOS und ROS 2 und ermöglicht es Robotik-Entwicklern, ROS 2-Knoten vollständig in Hardware zu implementieren. Dank des konsistenten Programmiermodells für Hardware und Software und des vollständigen Zugriffs auf den virtuellen Speicher können Entwickler ROS 2 Nodes leicht von Software auf Hardware und umgekehrt migrieren. Der ReconROS Executor erweitert das Konzept der ereignisgesteuerten Programmierung mit Callbacks auf die Hardware. Zu diesem Zweck wurden Hardware Callbacks eingeführt, die nach einem bestimmten Ereignis ausgeführt werden. Der ReconROS Executor umfasst eine Infrastruktur für die dynamische partielle Rekonfiguration in der Hardware und die Platzierung, Planung und Platzierung von Callbacks. Als dritten Beitrag schlägt diese Arbeit fpgaDDS vor, eine Intra-FPGA-Datenverteilung für Hardware-gemappte ROS 2-Knoten. Alle drei Teilbeiträge erzielen Geschwindigkeitsvorteile bei mitteleren Ausführungszeit und eine Reduktion des Jitters im Vergleich zu ihren reinen Software-Pendants. Schließlich werden in dieser Arbeit drei ausführlichere Anwendungsbeispiele vorgestellt, die von ReconROS und Hardwarebeschleunigung profitieren.

AUTHOR'S PUBLICATIONS

[1] Lennart Clausing, Zakarya Guetattfi, Paul Kaufmann, Christian Lienen, and Marco Platzner. "On Guaranteeing Schedulability of Periodic Real-time Hardware Tasks under ReconOS64." In: *Proceedings of the 19th International Symposium on Applied Reconfigurable Computing (ARC)*. 2023. DOI: 10.1007/978-3-031-42921-7_17.

[2] Christian Lienen, Mathis Brede, Daniel Karger, Kevin Koch, Dalisha Logan, Janet Mazur, Alexander Philipp Nowosad, Alexander Schnelle, Mohness Waizy, and Marco Platzner. "AutonomROS: A ReconROS-based Autonomonous Driving Unit." In: *2023 Seventh IEEE International Conference on Robotic Computing (IRC) (Accepted for Publication)*. 2023.

[3] Christian Lienen, Sorel Horst Middeke, and Marco Platzner. "FPGADDS: An Intra-FPGA Data Distribution Service for ROS 2 Robotics Applications." In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2023, pages 6261–6266. DOI: 10.1109/IROS55552.2023.10341921.

[4] Christian Lienen, Alexander Philipp Nowosad, and Marco Platzner. "Mapping and Optimizing Communication in ROS 2-based Applications on Configurable System-on-Chip Platforms." In: *Proceedings of the 9th International Conference on Robotics and Artificial Intelligence (ICRAI) (Accepted for Publication)*. 2023.

[5] Christian Lienen and Marco Platzner. "Design of Distributed Reconfigurable Robotics Systems with ReconROS." In: *ACM Transactions on Reconfigurable Technology and Systems* 15.3 (2022). ISSN: 1936-7406. DOI: 10.1145/3494571.

[6] Christian Lienen and Marco Platzner. "Event-Driven Programming of FPGA-accelerated ROS 2 Robotics Applications." In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. 2022, pages 615–623. DOI: 10.1109/DSD57027.2022.00088.

[7] Christian Lienen and Marco Platzner. "Task Mapping for Hardware-Accelerated Robotics Applications using ReconROS." In: *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*. 2022, pages 148–155. DOI: 10.1109/IRC55401.2022.00033.

[8] Christian Lienen, Marco Platzner, and Bernhard Rinner. "ReconROS: Flexible Hardware Acceleration for ROS2 Applications." In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pages 268–276. DOI: 10.1109/ICFPT51103.2020.00046.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ASIC	Application-Specific Integrated Circuit
BRAM	Block Memory
CLB	Configurable Logic Block
CPU	Central Processing Unit
DDS	Data Distribution Service
DMA	Direct Memory Access
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
HiL	Hardware-in-the-Loop
HMN	Hardware-Mapped Node
HMT	Hardware-Mapped Topic
HWC	Hardware Callback
ICAP	Internal Configuration Access Port
LFU	Least Frequently Used
LRU	Least Recently Used
LUT	Look-up Table
LoC	Lines of Code
MEMIF	Memory Interface
MMCM	Mixed-mode Clock Manager
MMU	Memory Management Unit
MPSoC	Multi-Processor System-on-Chip
OSFSM	Operating System Finite State Machine
OSIF	Operating System Interface
OSRF	Open Source Robotic Foundation
PCAP	Processor Configuration Access Port
PWM	Pulse-Width Modulation
QoS	Quality-of-Service
rcl	ROS Client Library
rclcpp	ROS Client Library for C++
rclpy	ROS Client Library for Python

RDK	ReconOS Development Kit
RMW	ROS Middleware
ROS	Robot Operating System
RPC	Remote Procedure Call
RS	Reconfigurable Slot
SLAM	Simultaneous Localization and Mapping
SMN	Software-Mapped Node
SMT	Software-Mapped Topic
SoC	System-on-Chip
SWC	Software Callback
URAM	UltraRAM
VDMA	Video Direct Memory Access
VHDL	Very High Speed Integrated Circuits Hardware Description Language

1

INTRODUCTION

In recent years, robots, and in particular mobile robots, have become ubiquitous in our daily lives. Example application fields for mobile robots can be found as autonomous vacuum cleaner robots at home, as drones for tasks such as finding fawns in agriculture [40], as (partially) autonomous cars in road traffic [134], or even for explorations on foreign planets in our solar system [127]. A common trend for mobile robots is to increase the degree of autonomy [100]. An increased degree of autonomy is usually achieved by observing the robot's environment more precisely using more sensors and the resulting information to make better decisions, such as using machine learning-based algorithms [6, 24]. Another example is the generation of the robots' pose or complete maps of the environment from standard camera images [105]. Cameras are widely used on mobile robots, and advanced computer vision algorithms such as visual simultaneous localization and mapping (SLAM) represent a crucial component for pose estimation [126]. The aim of all of these efforts is to provide a foundation to act more autonomously or *intelligent*.

On the one side, the increased degree of autonomy results in higher computational workloads for the mobile robot processing platform due to more significant amounts of data and advanced algorithms [54]. On the other side, the performance of data computation in mobile robots is crucial since large amounts of data must be processed in real-time [133]. The demand for more processor power is in direct competition with mission time since the processor is usually powered by the same battery, even if the power consumption of the motors of a drone, for example, is orders of magnitude higher. Although modern central processing units (CPUs) have become more potent due to multi-core architectures and vector processing extensions, there is the need for heterogeneous compute platforms [19]. The need is due to the insufficient computing power of CPUs on the one hand and the insufficient energy efficiency of CPUs compared to accelerators on the other hand.

Candidates that are already in use are (embedded) general-purpose graphics processing units (GPUs) [84, 86, 103], application-specific integrated circuits (ASICs) [70, 110], and field programmable gate arrays (FPGAs) [4, 55, 56, 71, 72]. While ASICs are only economically reasonable for (very) large quantities due to their high fixed costs, gpGPUs and FPGAs have become more common in the last years. Modern FPGA-based system-on-chips (SoCs)

platforms often combine programmable logic and dedicated ARM cores, enabling close cooperation in processing through high-performance memory interfaces. With a high degree of parallelism on one side and the ability to adapt the processing unit to the task, FPGAs promise a fast and energy-efficient execution. Several examples in literature report improvements due to the usage of FPGAs compared to GPUs in terms of computation power and energy efficiency, e.g., gradients for robotic dynamics [85] for vision kernels [93], for morphological image processing functions [14], for feature detection and description algorithms [119] and convolutional neural network inference [75, 120].

However, despite the demonstrated advantages of FPGAs, their proliferation into the robotics domain is still hampered by several reasons. The design complexity is the main barrier for software developers to include FPGAs in their robotics architecture. On the one hand, FPGA design and software/hardware co-design are arguably more challenging than embedded software development. Robotics engineers and application developers are typically not trained in FPGA designs or hardware/software co-design. On the other hand, designing an adapted accelerator in a hardware description language is more challenging than programming the same functionality in software in a high-level language. As a result, Podlubne and Goehringer [88] show in a literature analysis that while the research interest in GPUs in robotics is increasing, the relative interest in FPGAs in robotics is decreasing.

One of the approaches to reduce the design complexity of FPGA designs is high-level synthesis (HLS). HLS tools enable the usage of standard C/C++ for describing behavior and (semi-)automatically take such descriptions to FPGA hardware. HLS tools increase productivity and are thus highly useful, but a consistent programming model for implementing software and hardware functions in robotics is still lacking. Porting a robotics application from software to hardware or accelerating parts of the application in hardware requires the creation of suitable interfaces between software and FPGA hardware and often leads to a re-development of substantial parts of the application. However, modern robotics applications are based on complex architectures involving large software packages and thousands of lines of code (LoC). That makes the integration of hardware acceleration even more challenging since the re-development of the overall application is only a valid option for some applications. Therefore, the compatibility of hardware accelerators with standard concepts has to be guaranteed.

In recent years, the robot operating system (ROS) has become the de-facto standard for developing robotics applications. ROS provides an architecture paradigm, a communication infrastructure, and tools for visualization and debugging. The architecture of ROS-based applications relies on so-called nodes, each responsible for a subfunction of the overall application. These nodes can communicate using $n : m$ topic-based publish-subscribe or $1 : 1$ communication schemes. The resulting decomposed architecture is usually represented by a computation graph comprising nodes and edges representing communication.

In summary, although FPGAs has excellent potential for robotics applications, a comprehensive approach to integrating reconfigurable hardware into modern robotics architectures is still lacking. Therefore, the main goal of this thesis is to establish the use of reconfigurable hardware in a standardized form for ROS-based robotic applications. For the standardized integration, this thesis explores a new approach that supports a more extensive set of features compared to related approaches (cf. Section 3.1), allowing better utilization of the capabilities of modern SoCs architectures.

1.1 THESIS CONTRIBUTIONS

The overall contribution of this thesis is to present a novel approach for the computation of subparts of the computation graph or even the complete computation graph using reconfigurable hardware. This effort results in the ReconROS open-source framework, which is publically available[1]. ReconROS allows but is not limited to shifting nodes or graphs completely to reconfigurable hardware by providing a consistent programming model for hardware and software-implemented nodes while being compatible with ROS.

The contributions can be divided into three three parts:

- First, based on a combination of ROS 2 and ReconOS, the **ReconROS** framework allows robotics developers to utilize hardware acceleration for ROS applications as hardware-accelerated ROS nodes or as ROS nodes mapped completely to hardware as so-called hardware-mapped nodes (HMNs). The latter option provides a consistent programming model for ROS applications, independently of mapping ROS nodes to software or hardware. **ReconROS** supports all ROS 2 communication paradigms and preserves the main advantages of ReconOS, such as full memory access for hardware threads or operating system-like synchronization mechanisms for hardware/software co-designed applications.

- Second, the **ReconROS executor** allows ROS 2 developers to easily exchange hardware accelerators during runtime without the need for custom scheduling and dispatching implementations. It, therefore, makes dynamic partial reconfiguration useable for a broad (non-hardware) community. As a result, it allows for a better utilization of the available resources due to time-sharing. The concept of the **ReconROS executor** generalizes the common ROS concept of event-driving programming using callbacks by introducing hardware callbacks. Hardware callbacks are partial FPGA designs loaded during runtime after a particular event (e.g., a new message is received).

- Third, the lean data distribution service **fpgaDDS** aims to reduce communication overheads between HMNs due to the mapping of communication of topics to hardware. **fpgaDDS** relies on a customized

1 https://github.com/Lien182/ReconROS/

and statically generated streaming-based communication architecture enabling intra-FPGA communication between HMNs. In order to preserve the programming model, extensions in the tool flow enable ROS 2 message and topic-based communication through dedicated streaming networks based on automatically generated macros. The gateways concept extends the application area for **fpgaDDS**. Gateways synchronize hardware-mapped and standard ROS 2 topics to minimize communication between the software and hardware domains.

Furthermore, this provides three more advanced example applications that successfully leverage the contributions of this thesis.

1.2 THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

- Chapter 2 provides an overview of the background and related work. The overview includes ReconOS and ROS since this thesis relies heavily on both frameworks. Furthermore, the chapter reports on related approaches for integrating hardware accelerators into the robot operating system.
- Chapter 3 presents the design and implementation of ReconROS. The chapter starts with design considerations for integrating reconfigurable hardware into ROS-based applications before it presents the architecture, design flow, and framework's programming model. Finally, the chapter reports overheads due to hardware acceleration and dynamic partial reconfiguration during runtime.
- Chapter 4 deals with mapping ROS 2 nodes to reconfigurable hardware using the architecture presented in the previous chapter. The mapping includes static mapping, where nodes remain in the execution unit during runtime, and dynamic mapping, where nodes are mapped to execution units dynamically during runtime. For dynamic mapping, the chapter presents the scheduling and replacement strategies used by the ReconROS executor.
- Chapter 5 describes fpgaDDS and its extensions by gateways for the optimization of robotics applications on the computation graph level. Additionally, the chapter presents an approach for optimizing the communication between nodes.
- Chapter 6 reports on three example architectures realized with ReconROS. The set of applications comprises a distributed ball-on-plate architecture, an autonomous driving example able to drive along a street lane and handle traffic lights during the drive, and a more advanced hardware-software co-design for autonomous driving with navigation.

- Chapter 7 concludes the thesis and provides an outlook for future work.

2

BACKGROUND AND RELATED WORK

After the introduction and motivation of the content of this thesis, this chapter provides background related to ReconROS. Since ReconROS started as a combination of ReconOS and ROS 2, both approaches are introduced in the following sections.

ReconROS is not the first approach aiming to integrate reconfigurable hardware into ROS-based applications in a standardized way. There are approaches in the literature presented before and after the introduction of ReconROS. Therefore, we provide an overview of related approaches for integrating reconfigurable hardware acceleration into ROS applications. In Chapter 3, we are going to compare these approaches with ReconROS.

2.1 RECONOS: OPERATING SYSTEM FOR RECONFIGURABLE COMPUTING

Several decades ago, reconfigurable hardware evolved from pure glue-logic devices connecting various integrated circuits and interfaces to powerful computing platforms [114]. This development was made possible mainly by advancements in manufacturing, which resulted in more available resources in a given area and lower costs per chip. Additionally, FPGAs are equipped with advanced dedicated functional elements, e.g., digital signal processors (DSPs) or block memory (BRAM) functional units. Later, FPGAs are combined with (multi-core) processors to SoC architectures with high-performance interfaces between the processing system and programmable logic. These powerful architectures are opening up more and more fields of application. However, the design methods used until then could not keep up with the growing availability of resources.

Therefore, there was a demand for more efficient design processes to increase efficiency during development. Eckart et al. [25] mention two approaches for more efficient development and short design iterations: First, there is the value of high-level synthesis, which allows high-level programming languages such as ANSI C/C++ to be used to develop hardware. The introduction is comparable to the transition of high-level languages for software development instead of assembler in the past [23]. The high-level synthesis relies on a complex toolchain comprising the *compilation* of the developers' specification into a formal model followed by the *allocation* of hardware resources of the FPGA, operations *scheduling*, the *binding* of

operations, variables and bus transfers and the *generation* of the register-transfer-logic [23].

As the second approach, several works in literature deal with the standardization of communication and synchronization between subcomponents. In order to communicate and synchronize with other system components, hardware blocks often use application-specific implementations, resulting in increased development expense and lower interoperability.

Traditionally, regular operating systems already provide well-established mechanisms for execution abstraction, communication, and synchronization. For the abstraction of the execution, modern operating systems concepts distinguish between processes and threads in that processes run on their own virtual hardware provided by the operating system [25]. The thread is a lightweight version of the process, allowing the exploit of thread-level parallelism by running multiple parallel tasks on one shared virtual hardware. For the synchronization between these execution units, the operating system provides several objects, e.g., mutexes, semaphores, conditional variables, and standardized APIs. Operating systems support objects similar to queues for realizing communication between two or more execution units. In order to provide standardized handling and access, many operating systems follow the POSIX standard, aiming to provide interoperability between several operating system types. For the execution modeling, the POSIX standard specifies POSIX threads (pthreads), providing a platform-independent interface for multi-threading.

Following these well-known standardized mechanisms, several works have been done that integrate reconfigurable hardware into the operating system to translate the concepts in the area of reconfigurable hardware. Some important examples are HThreads [3], R3TOS [41], SPREAD [124], or FUSE [39]. For a more complete overview of related approaches, the reader might consider the survey in. [25].

HThreads [3], the hybrid thread programming model for heterogeneous processing units, is an approach for a hardware-software co/designed operating system developed at the University of Kansas. It aims to generalize operating systems abstractions for hardware and software and reduce overheads in terms of execution time and jitter by implementing operating systems functions entirely in hardware. The interface between applications and functions is realized through a memory-mapped interface and, therefore, accessible for hardware and software threads.

Another approach is SPREAD [124], providing a streaming-based reconfigurable architecture both in hardware and software. It allows for mapping streaming-based applications to software and reconfigurable hardware threads. Hardware threads use a standardized hardware thread interface (HWI) comprising an input and output port. Hardware threads can use both ports to connect either to the main memory or to build a streaming channel to another hardware thread. Therefore, SPREAD can chain hardware threads leveraging these point-to-point streaming channels. The communication between software threads or mixed software and hardware threads uses queue

communication based on the shared main memory. From the application's point of view, SPREAD uses the pthread model to abstract hardware threads.

Ismail et al. introduced FUSE [39], an approach for hardware accelerator abstraction aiming to achieve transparency for the applications developer and operating systems support for integrating hardware accelerators. Therefore, FUSE comprises reconfigurable hardware accelerators in the programmable logic, a kernel driver per accelerator instance in the Linux kernel space, and a userspace library providing a programming interface to the software applications. Due to its multi-layered structure, it decouples the hardware acceleration from the software application and enables operating system support to integrate hardware accelerators. The communication between hardware accelerators and the software application is done via shared-memory communication without the support of virtual memory by the hardware accelerator.

The main objective of R3TOS [41] is to provide a reliable operating system tolerating faults in the silicon without additional design costs. For the execution of hardware tasks, R3TOS provides a 2D grid in the reconfigurable logic for placing hardware tasks. During runtime, R3TOS schedules hardware tasks and places them into the grid. The operating system achieves reliability and detects faulty logic by spatial diverse execution of replicated hardware tasks. In the case of detecting faulty logic, the operating system considers affected areas for later scheduling and placement decisions.

ReconOS [99], another approach for an operating system for reconfigurable computing, was first presented in 2007 in [57]. The main idea behind ReconOS is the usage of standard operating system-related principles for abstraction, communication, and synchronization for reconfigurable hardware.

For the transfer of the runtime abstraction concept into the domain of reconfigurable hardware, the design methodology of ReconOS extends the concept of pthreads across the hardware-software boundary and thus enables a heterogeneous multithreaded programming model. As a result, the ReconOS application can comprise a set of software and so-called hardware threads, each processing on a fraction of the overall application. Similar to multi-threading purely in software, all threads of the ReconOS application share one common virtual address space.

Additionally, ReconOS also follows standard OS concepts for communication and synchronization. The application designer can use synchronization primitives, e.g., mutexes and semaphores, for both hardware and software threads using a consistent programming interface. The same holds for queue-based communication via mailboxes.

Combined with a high-level synthesis design approach for hardware threads, the unified usage of synchronization and communication and the shared virtual address space allow for faster design space exploration. For example, algorithms previously developed in a high-level programming language in software can be migrated to a hardware thread through high-level synthesis. Therefore, in contrast to the previously mentioned approaches,

ReconOS generalizes the programming concept of pthreads to hardware threads, including, e.g., system calls and the shared virtual address space.

2.1.1 *ReconOS Architecture*

An example of ReconOS system architecture with two hardware threads and three software threads is shown in Figure 2.1. ReconOS hardware threads are accommodated in so-called reconfigurable slots (RSs), which are areas in the reconfigurable fabric providing access to the operating system interface (OSIF) and a memory interface (MEMIF). In order to map hardware threads to reconfigurable slots, ReconOS supports two options: First, the application designer can statically assign hardware threads to reconfigurable slots during design time. In that case, the application designer must define no explicit areas in the reconfigurable fabric. Second, the application designer can define areas in the reconfigurable fabric that can accommodate hardware threads during runtime. For this option, ReconOS makes use of dynamic partial reconfiguration.

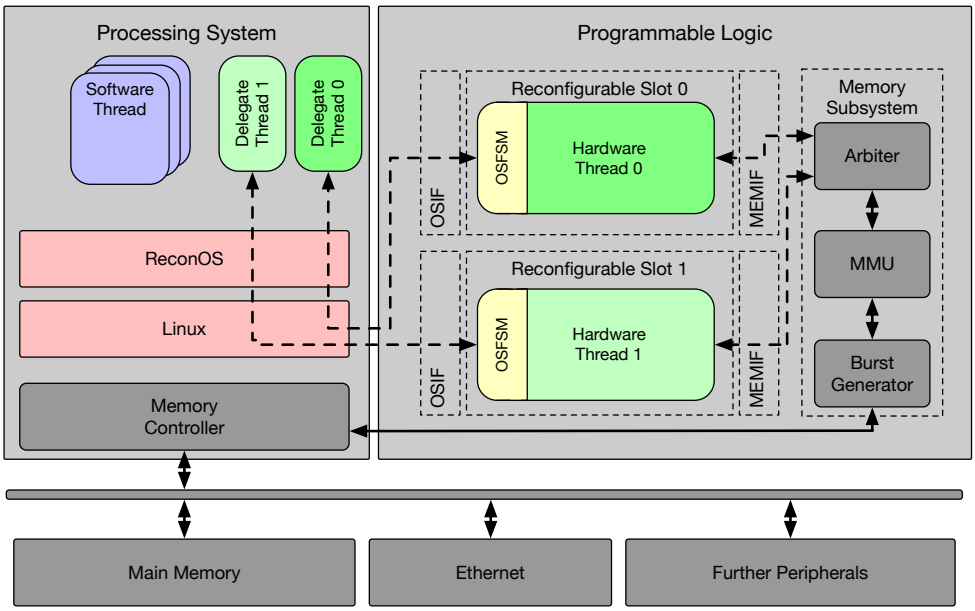

Figure 2.1: ReconOS architecture comprising two hardware threads and three software threads.

For each hardware thread, ReconOS generates an additional software thread, the delegate thread. The delegate thread represents the corresponding hardware thread and executes operating system-related functions on behalf of the hardware thread. Therefore, the concept of delegate threads allows the usage of standard operating system primitives by the hardware thread. If, for example, a hardware thread wants to use a synchronization object (e.g., a mutex), it sends a command and an identifier of the mutex instance to the delegate thread, which interacts with the mutex using a standardized interface. After that, the delegate thread responds to the hardware

thread. From the hardware thread perspective, the communication is done via the OSIF interface, which is controlled by the operating system finite state machine (OSFSM) as part of the hardware thread. The ReconOS Linux Driver handles the interface between the delegated thread and the OSIF on the processing system side. From the application programmer's point of view, the delegate thread is hidden [1].

The shared virtual address space is provided by the memory subsystem comprising an arbiter for the distribution of transfer time, the memory management unit (MMU) for the translation of the virtual address space to the physical, and the burst generator for more efficient interface utilization. Based on the memory translation, hardware threads can access other memory-mapped components and peripherals of the system.

2.1.2 *ReconOS Tooflow*

The ReconOS framework comes with the ReconOS development kit (RDK), a Python-based templating system, and a preprocessing toolflow. The overall build flow of ReconOS is sketched in Figure 2.2.

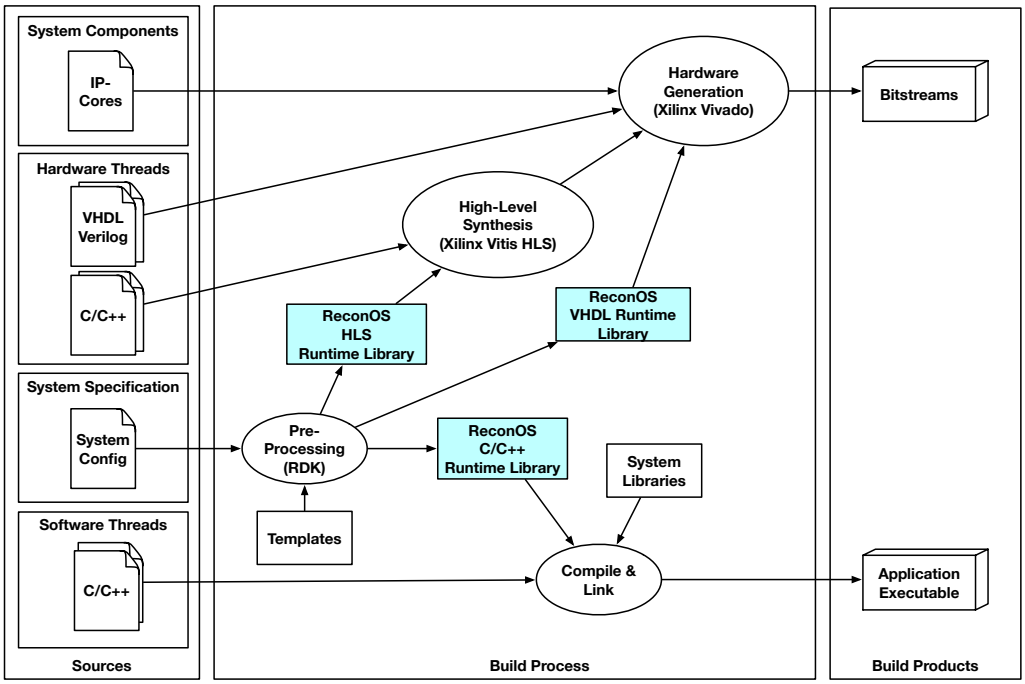

Figure 2.2: ReconOS build flow. Components in blue are (partly) generated by the preprocessing toolchain of ReconOS. Adapted from [1, 21].

A ReconOS project comprises sources for software threads written in C/C++, sources for hardware threads written in a hardware description language (very high speed integrated circuits hardware description language (VHDL) or Verilog), or sources in C/C++ for high-level synthesis. Furthermore, the application's designer can extend the project by additional system components in terms of IP cores for the hardware design. General

information about the project is gathered in the ReconOS-specific system configuration file.

The RDK uses the system configuration as an input file to obtain information about the project. The file comprises information about the used target FPGA platform, the synthesis toolchain, and several implementation-specific elements, e.g., reconfigurable slots, hardware and software threads, synchronization, and communication primitives. The RDK generates runtime libraries for HLS, VHDL, and C/C++ software sources using this information.

The first step of the hardware generation process is the synthesis of high-level hardware specifications into the register-transfer-level using Xilinx Vitis HLS. This step is repeated for each hardware thread designed in C/C++. After that, the high-level synthesis results are used together with the remaining hardware threads written in VHDL extended by the VHDL runtime libraries and eventually further IP-cores to generate the final bitstream. In the case of using dynamic partial reconfiguration, this process generates the partial bistreams as well.

Regarding software, the ReconOS runtime library for software and the software sources are compiled into an application executable for the target platform.

2.1.3 *ReconOS Project*

The ReconOS project was started at Paderborn University in 2006 and has since been continuously expanded and migrated to various CPU and operating system platforms [2].

The first 32-bit version of ReconOS using the eCos operating system [136] and running on PowerPC CPUs was presented by Lübbers and Platzner [57] in 2007. For the second version, FIFO interconnects between hardware threads, and the support of Linux and its virtual address space concept extend ReconOS. In 2013, ReconOS reached its third version. In combination with a minor update to 3.1, ReconOS supports the Xilinx Microblaze and Xilinx Zynq CPU platform and implements a more lightweight and modular design. In the fourth version, ReconOS supports the Xilinx Vivado toolflow and more advanced clock supply functionality for individual hardware threads. Additionally, two master projects migrated ReconOS to freeRTOS [30] running on ARM and to Zephyr [135] running on RISC-V [95] softcore platforms.

The most recent work extends ReconOS to support modern multi-processor system-on-chips (MPSoCs), including 64-bit ARM architectures [21]. The main novelties of ReconOS64 are the following:

- The bit width of both interfaces for hardware threads (OSIF and MEMIF) is increased from 32 bits to 64 bits. This expansion leads to modifications in all involved IP blocks and library functions.
- The memory translation unit was adjusted to the address space of the ARMv8 architecture. The modifications enable mapping the lower 39

bits of the virtual address space to a 48-bit physical address space and a page size of 4 KB.

- So-called reconfigurable slot groups extend the dynamic hardware thread management, which are sets of slots of the same size. Hardware threads can be assigned to one or more of these groups. The tool flow of ReconOS will then automatically generate a partial bitstream for each reconfigurable slot.
- ReconOS64 supports a finely adjustable clock supply for hardware threads. The clock supply IP-core leverages a mixed-mode clock manager (MMCM) tile with a static multiplier and a variable divider and can supply each reconfigurable slot group with a separate clock signal.

The evaluation of ReconOS64 implementation compared to the standard ReconOS implementation for 32-bit systems shows performance improvements for operating system calls and memory accesses for hardware threads [21].

2.2 THE ROBOT OPERATING SYSTEM

The Robot Operating System (ROS) [98] is an open-source framework for robotics applications on top of an operating system that was initially developed by Willow Garage and is now coordinated by the open source robotic foundation (OSRF). ROS comprises a design philosophy, a middleware, several packages including robotics-related algorithms, and development tools [61]. Due to its usage in countless open-source projects, ROS has become the de-facto standard for developing robotics applications in the last few years. Furthermore, ROS is also used in commercial products due to its numerous advantages [59].

The first version of ROS (in the following ROS 1) was introduced in 2009 [94]. The development started in 2007 at Standford University and, after some time, went over to the robotics lab Willow Garage, from where it was transferred to the OSRF in 2013. ROS 1 is still a widely-used software, although it has several weaknesses and limitations [61]. Examples of these weaknesses are limitations in security and reliability, especially in lossy network environments, a single point of failure due to a central name server (roscore), and limitations regarding multi-threading, which prevents an efficient distribution of components to multi-core machines. These and other factors have led companies to build workarounds that undermine the standardization and interoperability of ROS [61]. Therefore, the OSRF (and the community) decided to design a new version (ROS 2) of the robot operating system to tackle the previously mentioned issues. Since this thesis is based on ROS 2, the remainder of this section will focus on ROS 2.

Following the design philosophy of ROS, the overall robotics application is decomposed into subcomponents, so-called nodes. The concept of nodes enables application programmers to split their applications into functionally independent subcomponents and promises code reusability and modularity

for their robot application in general. During runtime, ROS nodes run in their independent context and share data with other nodes using several communication paradigms [61].

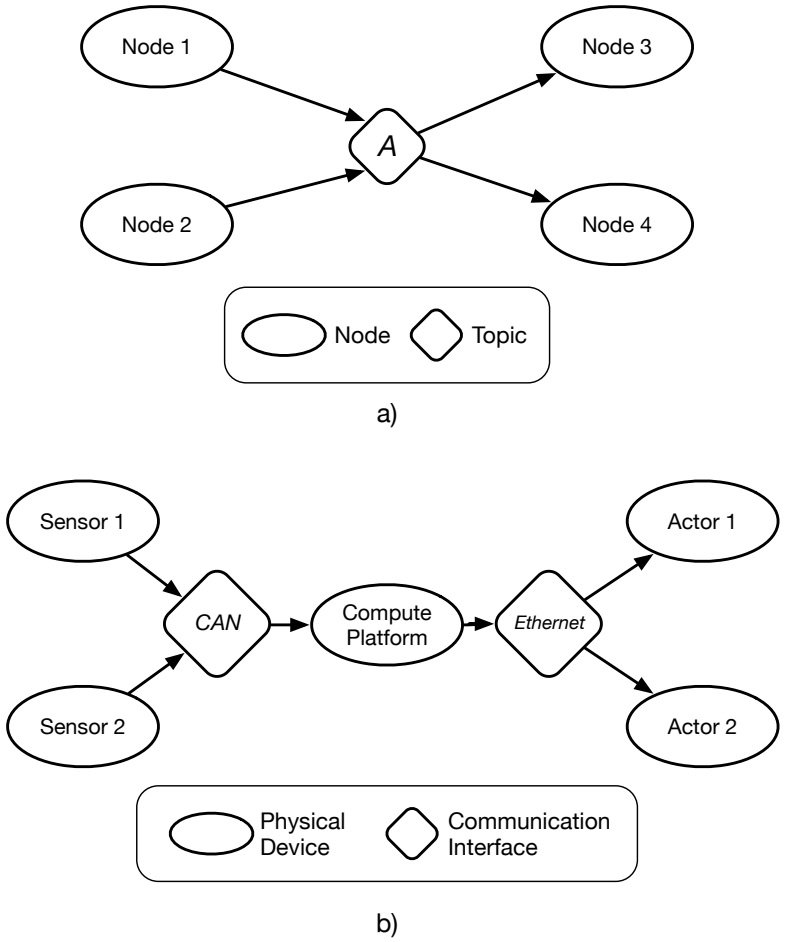

Figure 2.3: Example ROS 2 computation graph (a) comprising two nodes publishing to the topic A and two nodes subscribing data from the topic. The ROS 2 data layer graph (b) shows the physical structure comprising two sensors, two actors, and a computing platform. Adapted from [64].

The resulting decomposed robotics application is represented as a so-called computation graph. An exemplary computation graph is shown in Figure 2.3(a). The computation graph is not formally specified, so several different versions exist. However, in most versions, the ROS nodes are connected via edges, representing communication between involved nodes.

Another ROS-related abstraction is the so-called ROS data layer graph shown in Figure 2.3(b). Mayoral-Vilches and Corradi [64] mention the ROS data layer graph in their work, as it represents physical groupings and connections implementation of the behavior modeled by the computation graph. Therefore, it represents the system from the hardware point of view. Examples of the nodes in this graph are sensors, computing platforms, actors, and edge interfaces such as Ethernet or CAN.

Supported operating systems for ROS 2 are Linux, MacOS, and Windows. However, work has also been done to run ROS 2 on other operating systems, especially to enable low-power embedded systems. An example of such effort is microROS [9], which enables using ROS-related concepts, such as nodes, publishers, and subscribers, onto deeply integrated systems. However, microROS still requires a second platform running standard ROS 2, including an agent acting as a gateway for the embedded platform to communicate with other ROS nodes in the network. The connection between the embedded target platform and the agent can be Ethernet, Bluetooth, or a Serial interface. mROS [112] goes a step further and runs the complete ROS stack on the embedded device. Hence, an embedded device running mROS can interact with other ROS nodes in the network without the need for a dedicated agent platform. mROS is available both for ROS 1 [112] and ROS 2 [139]. ROS-lite [5] is a development framework for embedded multi-core architectures targeting network-on-chip (NoC) communication between cores. ROS-lite is based on the eMCOS real-time operating system for many-core processors. It leverages its message infrastructure for exchanging ROS-like messages between ROS nodes running on separate cores. RT-ROS [125] divides the execution platform into a non-real-time Linux environment and a real-time capable environment running Nuttx.

2.2.1 *Communication Interfaces*

For the communication between nodes, ROS offers different interface types. The set of available communication interfaces comprises (i) a many-to-many publish/subscribe model, which allows to broadcast messages to multiple subscribers but is one-way, (ii) services that follow a client-server model where the server provides data only if requested by the client, basically mimicking a remote procedure call (RPC), and (iii) actions, which make use of services but here the client can receive regular feedback about the server's progress. An example of three nodes leveraging all types of communication is shown in Figure 2.4.

The most common paradigm is the publish-subscribe communication that allows asynchronous data exchange via messages published to topics. Using this communication type, ROS nodes can run with varying rates and execution times independently from other nodes. In the example, node C publishes a message to the topic that nodes A and B subscribe to.

For a synchronous communication similar to a RPC, nodes B and C perform a ROS service communication. In this example, node C is in the role of the service server, whereas node B makes use of the offered service as the service client. Service-based communication is performed in two phases: first, the client sends a request to the server, which is, eventually, after the processing phase, responded by the server.

A unique asynchronous communication pattern is the ROS action that is performed between node A (as action server) and node C (as action client). This communication paradigm comprises three subelements: goal,

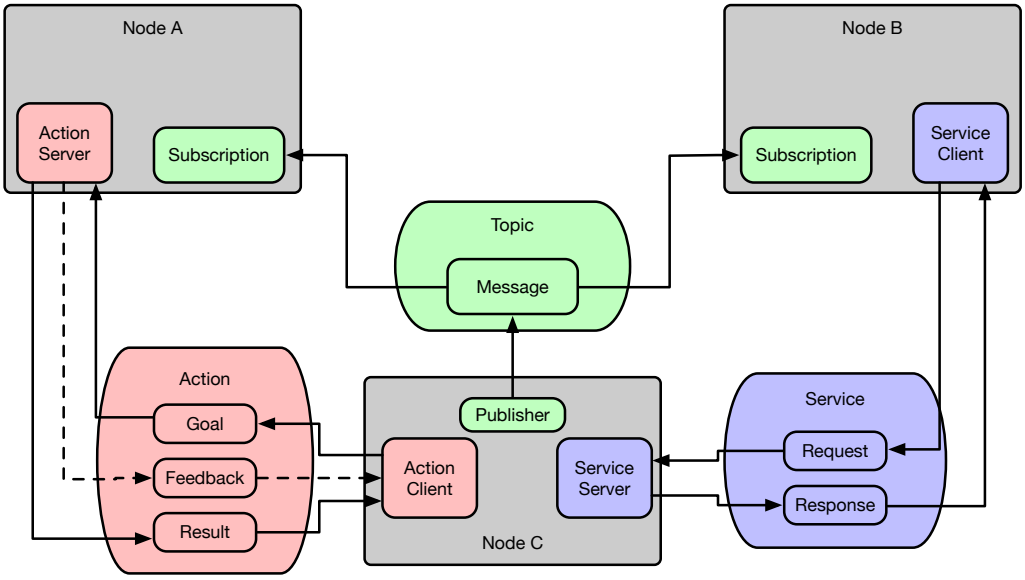

Figure 2.4: Example ROS 2 network comprising the three communication paradigms supported by ROS: Topic-based (green), service-based (blue), and action-based (red). Adapted from [61]

feedback, and result, whereas goal and feedback include request-response communication between server and client and feedback from server to client only. The action-based interaction between two nodes starts with the goal request from the action client. After that, the action server may accept or refuse this request. If accepted, the server eventually starts sending feedback. After receiving an acknowledgment that the server has accepted the goal, the client can send a further request to the server to get the task result. This request is then, possibly after a specific time, answered by the server with a response. Furthermore, the client has the option of canceling the action during processing. Due to its multi-phase structure, ROS 2 actions are well suited for long-running tasks, e.g., navigation or manipulation tasks [61]. Technically, ROS 2 actions are implemented based on ROS 2 publish-subscribe communication via a feedback topic and ROS 2 service communication for goal and result.

All three ROS interfaces rely on semantically specified data formats (messages) that are either pre-defined as part of the ROS installation or custom-designed by the applications developer. These ROS 2 data formats are multi-layered combinations of built-in data types such as integers, floats, and strings. For the specification of interfaces, ROS relies on an interface definition language (IDL). Based on the IDL specification, the tool flow generates the source code required for communication and usage in the user application. Since a fixed length for arrays of datatypes or strings is not mandatory, or the length of a message might vary during runtime, the ROS middleware (RMW) supports dynamic memory allocation for messages.

2.2.2 *Client Library Stack*

One significant improvement of ROS 2 is the exchangeable communication infrastructure, which results in higher scalability, reliability, and durability [62] and performance compared to ROS 1. However, in order to support different data distribution service (DDS) implementations, the use of abstraction levels became necessary. The resulting ROS 2 stack is shown in Figure 2.5.

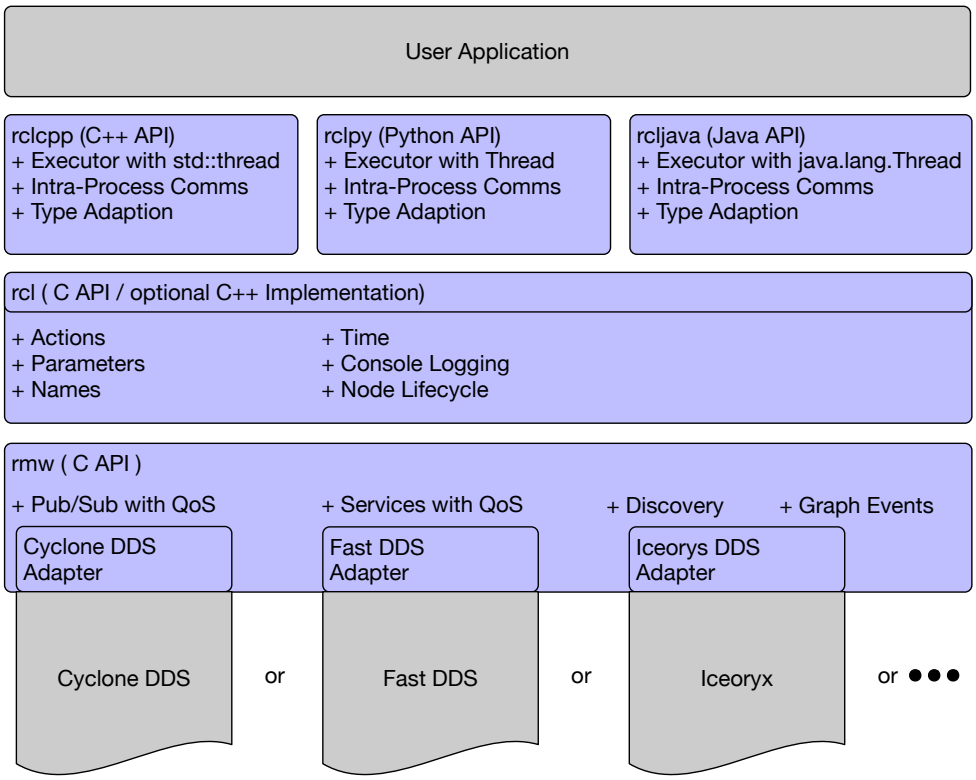

Figure 2.5: ROS 2 client library stack. Adapted from [61].

Starting from the bottom of the stack, ROS 2 supports several different DDS implementations. The DDS is an industry-standard for decentralized communication and is available from different vendors. These implementations differ, for example, in terms of range (intra-network vs. intra-platform communication) or real-time capability. A prominent example of an intra-network DDS is FastDDS [137] by eprosima. Iceoryx [138] is an example of an intra-platform DDS that uses shared memory for data exchange. Further, there are also mixed forms that optimize communication depending on the destination. For instance, CycloneDDS [26] combines intra-network and intra-platform communication and employs Iceoryx internally.

The ROS middleware (RMW) includes an individual adapter for each of these DDS implementations abstracting implementation-specific properties. Based on that, the layer provides essential communication, including publish-subscribe and service communication with quality-of-service parameters. Additionally, it cares about the discovery of other nodes and can detect and handle changes (events) in the computation graph.

The ROS client library (rcl) is located on top of the RMW and builds more advanced functionality, e.g., ROS 2 actions, the handling of parameters for ROS nodes, or console logging.

For the design of the user application, ROS 2 offers the usage of C++, Java, or Python. Each high-level language provides a user-level scheduler (alias Executor), intra-process communication via shared pointers, and type adaption, comprising extensions for easier conversation of ROS interfaces to common data types. Intra-process communication is leveraged by the ROS 2 concept of node composition [60]. Node composition allows the gathering of nodes manually or dynamically into a process to enable a sparing usage of resources and zero-copy communication between nodes in the shared process.

In recent work, Ichnowski et al. [38] accelerate ROS-based applications by shifting compute-intensive workloads into a public cloud environment. In their approach, FogROS 2, the authors establish a virtual private network connection between the robot and a public cloud to enable DDS-based communication between local nodes running on the robot's compute platform and nodes running in the public cloud. Although there are significant communication overheads, the authors achieve speedups up to 45× due to cloud offloading compared to local execution on the robot's compute platform.

The communication of ROS nodes and different DDSs have frequently been the subject of research. Much work deals with the analysis of the communication performance, e.g., in terms of latency [43], real-time performance [90, 91], or for different quality-of-service (QoS) parameter setups different QoS parameters [29, 62, 115]. Other work proposes tools for the analysis of communication in ROS 2 applications [8] or the dynamic binding of DDS implementations [68] according to implementation-specific properties.

2.2.3 *User-Level Scheduling*

Generally, the applications developer has two options for implementing the ROS 2 nodes. First, the developer can realize a particular node by implementing a $while(1)$-loop and use, e.g., blocking or non-blocking subscription for data receiving or publishing to send data to other nodes. In such a case, the node must regularly poll the communication layer for available messages. This ROS node can then execute as a Linux thread or even as a separate Linux process, and the underlying Linux scheduler distributes the nodes to the cores of the CPU.

However, the second and more common model under ROS 2 is the event-driven model, where nodes register callbacks that are executed when specific events occur. The behavior description of ROS nodes is therefore divided into one or more callbacks. There are four categories of callbacks: Callbacks executed by any node when a (periodic) timer event occurs, callbacks executed by a subscriber on a received message, callbacks executed by a ROS 2 service server on a received service request, and callbacks executed by a ROS 2 service client on a received service response. ROS actions do not need a separate

callback class because they are technically based on publish-subscribe and service communication.

ROS provides a so-called executor function that interacts with the underlying communication layer and timer infrastructure to catch events and execute callbacks in a run-to-completion mode utilizing one or more worker threads. That is, callbacks are not preempted. By default, ROS 2 offers standard single-threaded and multithreaded executors for C++, Java, and Python applications that implement the scheduling algorithm sketched in Figure 2.6. The algorithm comprises two nested loops. In the outer loop, the executor interacts with the underlying communication layer to collect all ready subscriber, server, and client callbacks into a *readySet*. The *readySet* is a copy of all callbacks registered at the executor that are ready to execute callbacks at this point. In the inner loop, the executor checks for timer-triggered callbacks and executes them if such are available. Then, subscriber, server, and client callbacks are considered in that order, and if such a callback is ready, it is executed and removed from the *readySet*. If no more callbacks are ready, the outer loop's next iteration is started after a configurable waiting time.

The scheduling algorithm has no explicit assignment of priorities for callbacks. The ROS 2 executor implicitly implements priorities in the sense that timer-triggered callbacks get high priority, and the other callbacks lower priorities since they are first collected in the outer loop and then executed in the inner loop in the order shown in Figure 2.6. Within one callback category, requests are ordered by the sequence of their initial registration at the executor.

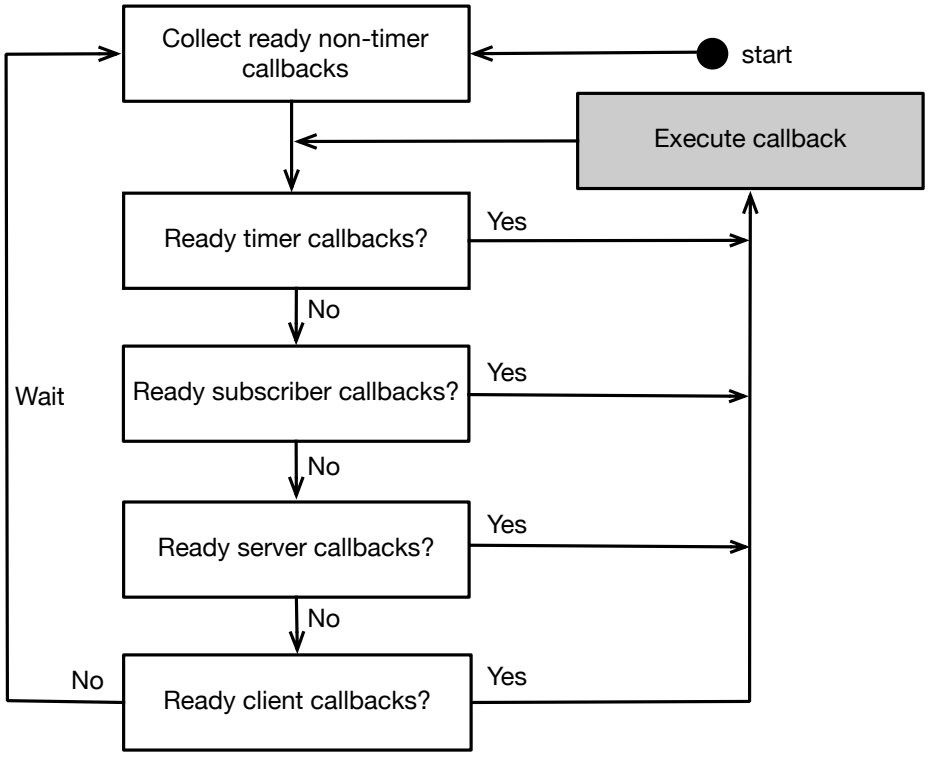

Figure 2.6: ROS 2 standard executor scheduling algorithm. Adapted from [17].

In recent years, much research has been done in analyzing the ROS executor in terms of schedulability and real-time behavior. The real-time behavior of the ROS 2 executor was studied in [17]. The authors analyzed the response time of ROS 2 applications. They provided a scheduling model, a worst-case response time analysis, and general insights into the real-time behavior of ROS 2. Based on that work, other publications deal with more precise response time analyses for more realistic results [12, 113] or improved scheduling strategies for better real-time performance [20, 104, 111]. Additionally, there has been work done to either automatically analyze the scheduling and latency performance in ROS 2 [7, 10, 44, 83] or automatically reduce the latency and jitter of ROS 2 applications by using an automatic latency manager [11].

Recently, alternative concepts to the standard ROS 2 executor were also researched. For example, in [106], an executor for micro ROS platforms [106] equipped with embedded microcontrollers was presented. This executor is fully coded in C, supports domain-specific requirements, and improves the analysis of real-time aspects. In another work [132], the microROS executor was used to replace the standard executor of ROS 2, aiming to improve the real-time performance of the overall application. Ghiglino and Sarabia [32] presented an improved implementation of the ROS 2 executor based on a lock-free ring buffer. The implementation outperforms standard ROS 2 executor implementations, especially for large callback sets.

2.3 RELATED APPROACHES FOR ROS-FPGA INTEGRATION

Integrating reconfigurable logic into the ROS ecosystem is still an open topic of research [123]. Several approaches have been presented in recent years that integrate reconfigurable hardware accelerators into a ROS-based software architecture. Similar to the survey paper in [88], this section distinguishes the literature into two categories: application-specific integrations of reconfigurable hardware into ROS-based applications work and general approaches that aim to ease the integration for any application.

2.3.1 *Application-Specific Approaches*

Several examples exist in the literature from the research area of application-specific integrations. In order to use hardware acceleration for faster computation, all of the presented works either use reconfigurable hardware for image capturing and preprocessing and the generation of control signals for a motor, or they offload compute-intensive parts of ROS nodes to hardware leveraging a remote-procedure-call sequence. Examples are presented in [36, 67, 73, 74].

Hasegawa et al. [36] describe their autonomous driving architecture based on ROS 1 and running on a Zynq 7020 SoC. Most components, e.g., Lane Dection, Obstacle Detection, Traffic Signal Detection, and Path Planning, run on the progressing system, whereas image capturing, including preprocessing

and a pulse-width modulation (PWM) signal generation, is realized in the programmable logic. A similar application architecture is described in [73], which implements image capturing and preprocessing in hardware.

The ZytleBot [67, 74] implementation goes a step further and does not only implement preprocessing and actor control signals in hardware but also compute intensive parts of ROS nodes. In order to achieve faster execution and better real-time performance of the overall application, the traffic signal detection based on a support-vector machine was implemented in hardware. The authors report a speedup of over 270× for the hardware implementation compared to the software.

The usage of reconfigurable hardware in the context of drones is presented in [76]. Their proposed framework, MPSoC4Drones, helps application developers generate Ubuntu-based operating system images supporting memory-mapped FPGA components and the drone flight controller PX4.

2.3.2 *General Approaches*

Despite the previously presented application-specific acceleration of subfunctions in ROS-based applications, several more general approaches are reported in literature.

A general platform for experimental robots is reported in [108]. In order to create a robotics prototyping platform, they divide the dual-core Zynq-7000 platform into three sections: first, there is the reconfigurable logic responsible for sensor preprocessing and the generation of robot output signals. Second, one core of the dual-core ARM processor processes control algorithms with hard real-time constraints on bare metal. Third, the other core processes Linux and ROS and therefore provides an interface for other ROS nodes. A distributed network of FPGAs can extend the signal conditioning part using TosNet, which provides memory access across multiple nodes by memory mirroring.

Yamashina et al. [131] proposed so-called ROS-compliant FPGA components to meet three requirements. First, ROS-compliant FPGA components must be able to communicate with any other ROS node in the network via publish-subscribe communication. Second, the functionality implemented in hardware must be equivalent to the software implementation, and third, the message and interface type of the component must be equivalent to the software implementation. The proposed architecture to meet these requirements comprises a ROS node that is implemented in software and accesses the hardware component, i.e., the accelerator, via a software wrapper. Communication within the ROS network is wholly handled in software, and whenever acceleration is needed, only the payload of the ROS message is transmitted to the hardware component. Semantically, the communication between the ROS software wrapper and the hardware accelerator is a RPC, realized in Xillinux [129]. In a case study, the authors apply their concept of an image labeling problem and achieve a speedup of 1.7× compared to pure software implementation running on the CPU.

Based on works about ROS-compliant FPGA components, the authors outline high development costs due to the custom interfacing of hardware acceleration kernels for each individual application. Therefore, in [80, 130], the automated design tool cReComp (creator for the reconfigurable component) is presented to help generate ROS-compliant FPGA components and thus reduce development costs. For the implementation of a ROS-compliant FPGA component with cReComp, the developer has to modify a configuration file and create user logic for the hardware accelerator. The configuration file contains information about the interface between the processing system and the programmable logic. cReComp generates the software and hardware parts for this interface. An evaluation by a group of test developers confirmed higher design productivity compared to manually designed interfaces.

ROS-compliant FPGA components and cReComp were taken up in another work [79] for an architecture exploration of ROS networks. For the architecture exploration, the authors propose hardware/software partitioning on the ROS node level, which is convenient due to the loose connection of ROS nodes via topics. In order to partition ROS nodes, a single ROS node can be replaced by a ROS subnetwork, proving equivalent functionality. In the next step, ROS-compliant FPGA nodes and cReComp support the application developer to implement compute-intensive ROS nodes in hardware. Finally, the authors propose collecting reusable components into a library to improve design efficiency for future projects.

ROS-compliant FPGA components potentially suffer from high communication latencies between ROS nodes, especially implementations with lower computation times [78]. Therefore, in follow-up work, Sugata et al. [109] aims to reduce these times by implementing the ROS publish/subscribe messaging in hardware. The authors motivate their work by demonstrating the communication overheads of a hardware-accelerated image processing example, in that the communication occupies 85 percent of the total application latency comprising message receiving, processing, and sending. In ROS 1, the communication is divided into two phases: the connection establish phase based on the XMLRPC-protocol, which involves the ROS master as well as the publisher and subscriber nodes, and the data transmission phase based on the TCPROS protocol, in that data is directly send from the publishing node to the subscribing nodes. In the presented approach, the authors aim to accelerate only the data transmission phase since it copes with more significant amounts of data and is repeated for each data transmission, in contrast to the connection-established phase. In order to process the data transmission phase in hardware, the implementation relies on a hardware TCP/IP stack (SiTCP [118]). The evaluation on an example application shows reduced communication time between nodes by 50 percent.

Ohkawa et al. [77] extend this work by using HLS for accelerator implementation and ROS protocol interpretation to increase productivity. Their approach takes the ROS message definition, the ROS node configuration, and behavioral code written in C/C++ for the accelerator and generates the FPGA

design. The infrastructure of the generated design includes several components: the hardwired TCP/IP stacks for the data communication phase, a data conversion between ROS messages and the application, an interface between the data conversion and the application, and, finally, the application itself. In order to evaluate the improvements in design efficiency, the authors compare the lines of code for their HLS-based approach (127 lines) and an implementation in a hardware-description language (860 lines) and observe a reduction of 85 percent.

Eisoldt et al. [27, 28] presented ReconfROS, an integration approach following a remote-procedure-call pattern via shared memory. ReconfROS targets system-on-chip architectures, providing a processing system executing ROS and its nodes and a programmable logic accommodating hardware acceleration kernels. The communication between the ROS nodes in the processing system and the hardware acceleration kernels is done via shared-memory regions. For larger amounts of data, ReconfROS equips acceleration kernels with dedicated memory interfaces for data access without CPU utilization. In order to execute a particular algorithm in hardware, the corresponding ROS node has to provide data in a shared-memory region, start the kernel via memory-mapped registers, and then wait for its execution. The authors evaluate their approach for a path detection algorithm and achieve lower runtimes and higher energy efficiency due to hardware acceleration.

Forest, a framework for generating hardware accelerated ROS 2 nodes from high-level synthesis designs, was presented in [45]. The architecture of a ROS 2 node accelerated by Forest is similar to the ROS-compliant FPGA nodes introduced in [131] except for the usage of software drivers and hardware infrastructure from the Xilinx PYNQ open-source project [82]. The toolflow of Forest is based on a configuration file that includes information about the input and output variables of the accelerator and its locations. This information, in combination with the sources of the acceleration kernels, is used to generate a ROS 2 interface package and a ROS 2 package, including the accelerated ROS 2 node. The tool flow can also generate corresponding listener and talker ROS 2 nodes for communication and processing testing.

While [77, 109] migrate almost a complete ROS node to hardware, Podlubne and Göhringer [87] go one step further and propose a methodology for full-hardware implementation of several ROS nodes. The architecture template for the proposed methodology is presented in Figure 2.7 and comprises four parts: ROS application nodes that use publish/subscribe communication, one converter block per node, the communication interface comprising arbitration, ROS protocol generation and TCP/IP interface, and a high-level management unit (so-called Manager). Each node is companioned by a node-specific converter core, which provides a standardized interface to the communication components of the architecture as an AXI interface. The interface between node and converter is a one-to-one equivalence of the corresponding ROS message in a parallized form, which means one signal per ROS message item. For communication with other ROS nodes outside of the FPGA, the protocol generator forms ROS compatible messages

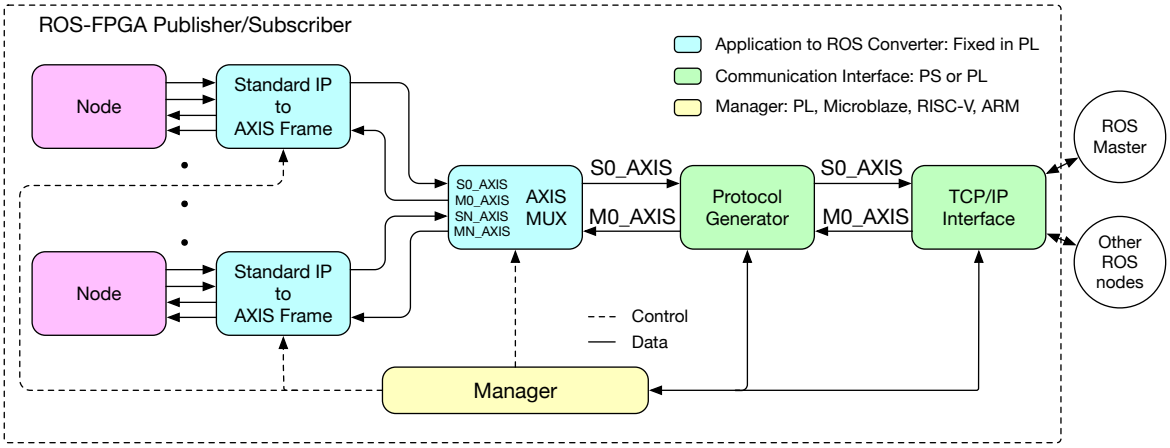

Figure 2.7: The ROS-FPGA architecture enables the implementation of ROS nodes without a processing system. Adapted from [87].

from the AXIS data. It sends it through the TCP/IP interface. However, communication is also possible in the other direction. In this case, data from other ROS nodes arrive at the TCP/IP interface block, which is then forwarded to the corresponding subscriber nodes via the protocol generator. Conceptually, the application-to-ROS converter must reside in hardware, but the communication interface and the manager could also be mapped to the processing system of the FPGA platform. However, the main feature of this methodology is the option to fully implement one or more ROS nodes in hardware and map them to reconfigurable logic without needing a processor. Following this methodology, any application implemented in reconfigurable hardware can be made ROS-compatible.

In a follow-up work [89], the authors examine in particular the automated generation of interfaces between ROS node and converter based on ROS messages. In particular, this work aims to automatically generate the infrastructure around the actual ROS nodes. The authors follow a model-based approach, where the message definition is first converted into an intermediate model, which is later transformed into a hardware description language using a template system. Thus, the approach supports not only ROS 1, but also ROS 2 and other middlewares.

The industry has also included the integration of reconfigurable hardware in ROS-based robotics architectures in its product portfolio. Mayoral-Vilches [64] points out costs due to the general-purpose nature of common compute platforms such as CPUs and GPUs. Examples are the need for additional hardware due to fixed architectures, time inefficiencies that make it hard to meet real-time deadlines, and higher power consumption. Therefore, Xilinx introduced the KRIA robotics stack (KRS) [63, 64], a set of libraries and utilities for FPGA-based hardware acceleration. In order to follow a roboticist-centric approach for hardware acceleration, the KRIA toolflow builds on modified standard ROS 2 build tools. Therefore, Xilinx Vitis, responsible for the build of acceleration kernels, and XRT [128] (Xilinx Runtime library), responsible for the runtime communication between ROS 2 node

and acceleration kernel, are hidden from the application developer. From the architecture point of view, KRIA follows the concept of acceleration kernels invoked from standard ROS 2 software nodes.

Such as Kria, RobotCore [65] aims to use standard ROS 2 tools for building hardware-accelerated applications. Therefore, they propose a technology-agnostic approach, which abstracts the ROS build system from vendor-specific tools. These extensions are contributed to ROS 2 ecosystem and maintained by the ROS 2 Hardware Acceleration Working Group [97]. Additionally, the authors propose a benchmarking infrastructure based on the Linux Tracing Toolkit next generation (LTTng) aiming to provide a lean standardized approach for ROS 2 applications. The authors target the communication overhead between hardware accelerated ROS nodes as a third contribution. Therefore, they implemented two approaches: (i) a kernel fusion approach that merges two subsequent acceleration kernels and (ii) direct communication via intra-FPGA streaming between two subsequent nodes. The evaluation shows a speedup of (i) 27 percent or (ii) 24 percent, respectively. However, due to the resulting transformations on the applications computation graph, the methodology breaks with the design philosophy of ROS.

3

DESIGN AND IMPLEMENTATION OF RECONROS

This chapter presents the design and implementation of ReconROS, an open-source framework for standardized robotic computing on FPGAs. ReconROS combines ROS 2 and ReconOS (in 32-bit and 64-bit) and is available for SoC architectures from Xilinx with both 32-bit and 64-bit architectures. ReconROS allows robotics developers to utilize hardware acceleration for ROS applications either as hardware-accelerated ROS nodes or as ROS nodes mapped completely to hardware. The latter option provides a consistent programming model for ROS applications, independently of the mapping of ROS nodes to software or hardware. In order to enable resource time-sharing during runtime, the ReconROS executor leverages dynamic partial reconfiguration with a reconfigurable slot model following standard ROS 2 programming paradigms. As a result, (i) robotics application developers can exploit exchangeable hardware acceleration from their known programming environment and event-driven programming model, and (ii) the limited hardware resources are operated efficiently.

The chapter starts with design considerations in Section 3.1. This chapter depicts different approaches for integrating reconfigurable hardware into ROS-based applications and compares the ReconROS framework with approaches from related work. The architecture of ReconROS, including extensions for the ReconROS executor, is the subject of Section 3.2 followed by explanations about the proposed design tool flow in Section 3.3. The programming model of ReconROS and example applications realized with ReconROS for a more extensive demonstration are the contents of Section 3.4. Lastly, Section 3.5 reports about results to quantify overheads of hardware threads in general and the dynamic reconfiguration process during runtime.

The architecture of ReconROS was previously presented at the International Conference on Field Programmable Technology (FPT) [53] and in a journal publication in the ACM Transactions of Reconfigurable Technology and Systems (TRETS) [50]. Architecture extensions for leveraging dynamic partial reconfiguration were presented in a separate conference paper on the Euromicro Conference Series on Digital System Design (DSD) [51]. This chapter mainly follows the descriptions from the TRETS journal [50] and DSD conference [51] publication. The automatic generation of C macros for ROS message transfer between hardware and software was built on the master's project of Sorel Horst Middeke [66].

3.1 DESIGN CONSIDERATIONS

The goal of the ReconROS framework is to provide developers of ROS 2-based robotics applications with a flexible means to utilize programmable logic for hardware acceleration. On the level of ROS 2 applications, several schemes for such integration are sketched in Figure 3.1. Figure 3.1(a) shows a scheme where some parts of a ROS 2 node, typically runtime-consuming functions, are mapped to one or several accelerators in programmable logic. The semantics of the communication between the ROS 2 node and the accelerators is that of a RPC. The node communication still relies on standard ROS 2 communication paradigms. In Figure 3.1(b), a hardware accelerator is shared between several ROS 2 nodes. Communication semantics is still RPC, but the implementation is more involved since proper arbitration between the accesses of the ROS 2 nodes is required. The third scheme shown in Figure 3.1(c) is the most advanced and allows the mapping of complete ROS 2 nodes to hardware. Essentially, the hardware accelerator is turned into a ROS 2 node. In this scheme, all ROS 2 nodes can communicate via the ROS 2 communication mechanisms, independently of their mapping to software or hardware. Semantically, this is the most intriguing scheme since it provides a consistent programming model across hardware and software where all ROS 2 nodes use exactly the same ROS 2 functions.

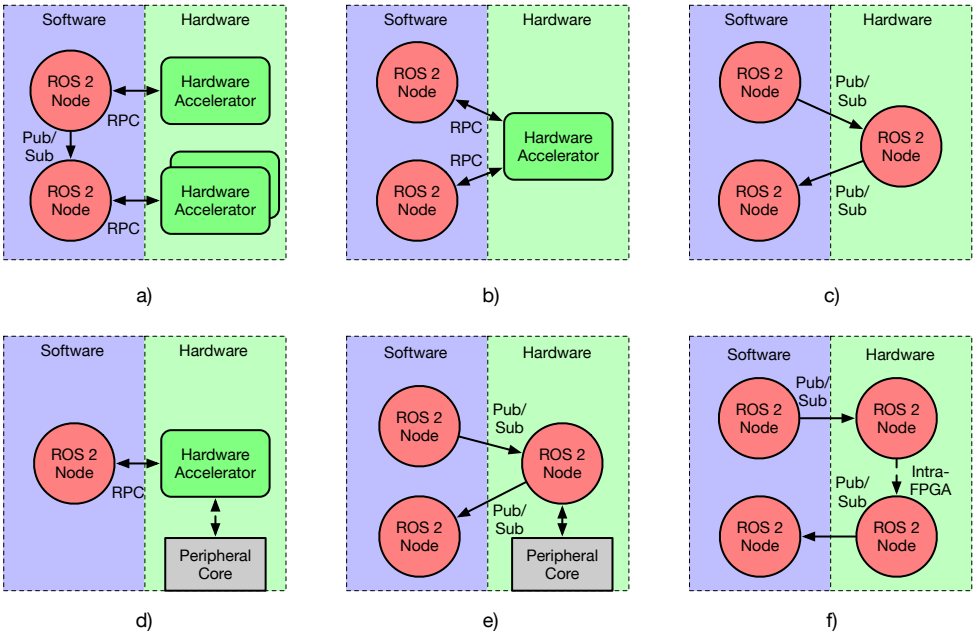

Figure 3.1: Different schemes for integrating ROS 2 node with hardware accelerators. Adapted from [50].

Often, developers decide to attach interfaces to sensors and actuators directly to the reconfigurable hardware and provide peripheral cores in hardware to access them rather than putting them under operating system control on the host CPU. Figure 3.1(d) and Figure 3.1(e) sketch such schemes with dashed lines. While these schemes are popular for maximizing

performance in concrete robotics applications, there are also two possible pitfalls:

First, flexibility is reduced since other ROS 2 nodes cannot access directly connected peripherals, and much less so when the ROS 2 nodes are mapped to different compute nodes in a distributed system. Second, many sensors and actuators come with standardized interfaces and corresponding drivers, e.g., USB, for which the use of an existing, software-accessible peripheral of the computing platform is much more productive than implementing suitable interfaces and protocol stacks in hardware.

Along the same line, the scheme shown in Figure 3.1(f) directly connects several ROS 2 nodes mapped to hardware via intra-FPGA communication without relying on ROS 2 communication mechanisms. This increases performance, especially for large amounts of data, but again lacks flexibility since the mapping of the ROS 2 nodes is severely constrained.

ReconROS integrates the ROS 2 middleware with the ReconOS/Linux architecture and programming model for hardware/software multithreading on platform FPGAs and can realize all schemes shown in Figure 3.1(a)-(f) and their combinations. On the one hand, ReconOS enables us to develop applications as a set of software and hardware threads under the shared memory model. On the other hand, ROS 2 allows for declaring several ROS 2 nodes within one Linux process. Therefore, in the schemes shown in Figure 3.1(a)(b)(d), each hardware accelerator is encapsulated by a ReconOS hardware thread. In contrast to most of the related work, ReconROS hardware accelerators can communicate with the ROS 2 software nodes not only by passing data in an RPC manner but it can also use shared memory communication in the Linux virtual address space, which is more efficient when larger data structures have to be passed and more intuitive from the applications designers point of view. In such a case, pointers to arbitrarily large ROS 2 messages are passed, and the accelerators themselves retrieve the relevant message payload from shared memory. Furthermore, since ReconOS hardware threads can execute standard operating system synchronization primitives, the required arbitration for the scheme in Figure 3.1(b) is straightforward to realize. In the more advanced schemes shown in Figure 3.1(c)(e), ReconOS hardware threads implement complete ROS 2 nodes and allow them to access operating system functions and also ROS 2 communication primitives, using the whole set of standard and even custom-defined ROS messages. Due to the usage of fpgaDDS, an extension of ReconROS for intra-FPGA communication aiming to preserve the programming model of ROS, ReconROS enables direct communication between hardware-mapped ROS 2 nodes following the scheme of Figure 3.1(f). fpgaDDS is subject in Chapter 5.

Table 3.1 compares ReconROS with related approaches. In contrast to all other approaches except for Forest [45] and KRIA [63–65], ReconROS leverages the more future-oriented ROS 2 version which promises improved scalability and real-time properties. Hardware acceleration of a ROS node primarily implies partitioning the node and implementing it as hardware/-

software co-design. This is followed by all approaches except FPGA-ROS [87], which maps nodes exclusively to hardware. Mapping several ROS nodes to hardware is possible in ReconfROS [27], KRIA [63–65], and ReconROS. Full memory access for hardware accelerators and arbitrarily long ROS messages are featured by ReconfROS [27] and ReconROS. A consistent hardware/-software programming model and the native support of dynamic partial reconfiguration are unique features of ReconROS. However, although dynamic partial reconfiguration could generally be used to exchange hardware accelerators in the various approaches, the approaches lack infrastructure for dynamic reconfiguration with less CPU utilization and (software) implementations for scheduling and placement of the accelerators that hide the complexity from the user.

Regarding the support of all available ROS 2 communication paradigms, all approaches relying on an RPC paradigm could easily support ROS services and actions besides publish-subscribe communication, as they have to change the software ROS node only. However, since all other approaches do not explicitly mention the support of other communication paradigms except for publish-subscribe communication, they are listed in brackets. Intra-FPGA communication between multiple nodes mapping ROS 2 topics with multiple-publisher-subscriber to hardware is a unique feature of ReconROS. However, Robotcore [65] supports intra-FPGA communication between hardware acceleration kernels, whereby the approach of ReconROS is more comprehensive due to its ability to connect more than two nodes, providing a consistent programming model for this communication type compared to standard ROS communication and automatic infrastructure generation.

3.2 RECONROS ARCHITECTURE

ReconROS inherits most of its hardware architecture from the underlying ReconOS [1, 58]. Figure 3.2 shows an example architecture with two hardware threads implementing ROS 2 hardware-mapped nodes (HMNs) and several software threads implementing ROS 2 hardware-mapped nodes (SMNs). The hardware threads are mapped to reconfigurable slots. They are connected to the Linux operating system kernel running on the CPU via the OSIF and to shared memory via the MEMIF. A so-called OSFSM is attached to each hardware thread to serialize the thread's operating system interactions. On the CPU, the communication with the OSIF is handled by a ReconROS driver and by lightweight delegate threads that serve the operating system calls for the hardware threads. The memory subsystem enables the hardware threads to access the whole address space of the ReconROS application, including shared memory and memory-mapped peripherals. ReconOS supports virtual memory and therefore includes a MMU in its memory subsystem.

To realize ReconROS, we needed to develop two components, the ReconROS stack and the ReconROS API for software and hardware threads.

Characteristic	ROS-compliant FPGA components [77, 109, 130, 131]	FPGA-ROS [87]	ROS-Enabled HW Framework for Robotics [108]	ReconfROS [27]	Forest [45]	Kria / RobotCore [63–65]	ReconROS (this thesis)
ROS version	1	1	1	1	2	2	2
Support of hardware/software co-designed ROS nodes	✓	✗	✓	✓	✓	✓	✓
Multiple ROS nodes per FPGA	✗	✓	✗	✗	✗	✓	✓
Consistent hardware/software programming model	✗	✗	✗	✗	✗	✗	✓
Memory access for hardware accelerators	✗	✗	✗	✓	✗	✗	✓
Support of arbitrarily long ROS messages	✗	✗	✗	✓	✗	✓	✓
Support of ROS services and actions	(✓)	✗	(✓)	(✓)	(✓)	(✓)	✓
Dynamic partial reconfiguration	✗	✗	✗	✗	✗	✗	✓
Intra-FPGA communication	✗	✗	✗	✗	✗	(✓)	✓

Table 3.1: Comparison of approaches for integrating hardware accelerators with ROS. Extended from [50].

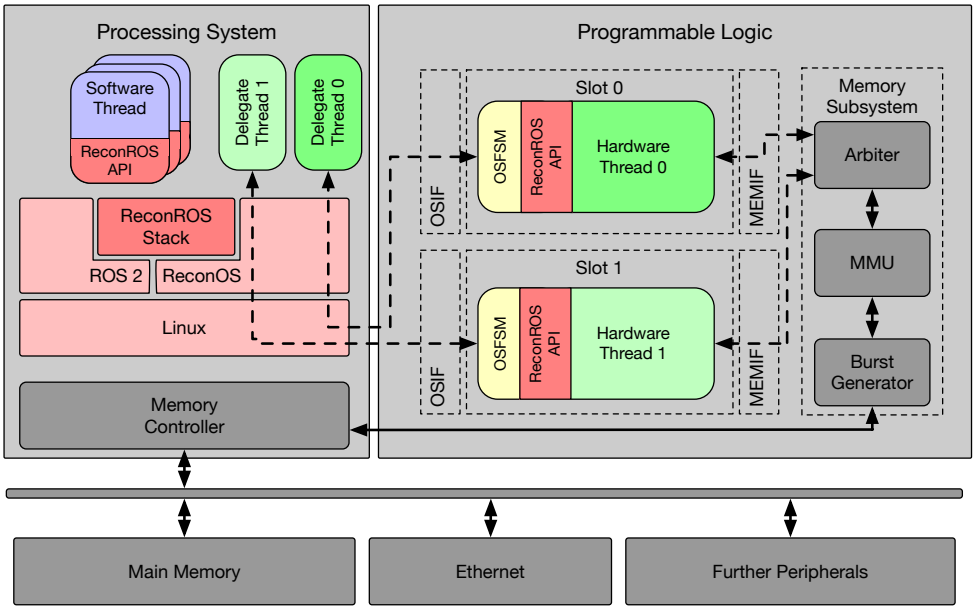

Figure 3.2: ReconROS architecture with two hardware-mapped ROS 2 nodes (threads) and several software-mapped ROS 2 nodes (threads). Adapted from [53].

The ReconROS stack extends the existing set of ReconOS objects, such as semaphores or mailboxes, with new objects related to ROS, e.g., *rosnode*, *rossub*, *rospub*, and *rosmsg*. These objects relate to ROS 2 nodes, ROS 2 publishers, ROS 2 subscribers, and ROS 2 messages, and can be created when configuring a ROS 2 application. Additionally, the ReconROS stack was extended by further ROS-related objects for ROS service and action-based communication.

The ReconROS API abstracts the standard ROS 2 API and allows ReconOS threads to access the objects of the ReconROS stack. As Figure 3.2 indicates, the ReconROS API is available for both software and hardware threads. For example, it includes the three functions `ROS_SUBSCRIBER_TAKE` for blocked message subscribing, `ROS_SUBSCRIBER_TRY_TAKE` for unblocked message subscribing, and `ROS_PUBLISHER_PUBLISH` for message publishing. Other functions as part of the ReconROS API allow, e.g., the usage of ROS 2 services and actions. Software threads can access the ReconROS API and the standard ROS 2 API to utilize a richer set of functions. For hardware threads, the exemplary set of three functions implemented in the ReconROS API is sufficient to implement ROS 2 HMNs that receive data, process it, and send it back using publish-subscribe communication. However, due to the flexibility of the underlying ReconOS system, any ROS 2 function can be made available for hardware threads.

In contrast to related work, our ROS 2 HMNs can access shared memory, thus implementing a more efficient ROS message handling. When hardware threads access functions of the ReconROS API for subscribing or publishing to topics, the OSIF and the delegate thread mechanism are used to pass pointers between the ReconROS stack in software and the hardware threads

to allow them to access the ROS message data structures in memory through their MEMIFs. Compared to message communication via the OSIF, which corresponds roughly to the mechanism used in related work, this design decision brings about two advantages: First, the MEMIF provides higher data rates due to the AXI high-performance interface of the processing system. Second, the data can be transmitted without using the processing system, which leads to more potential for parallel execution of software and hardware threads.

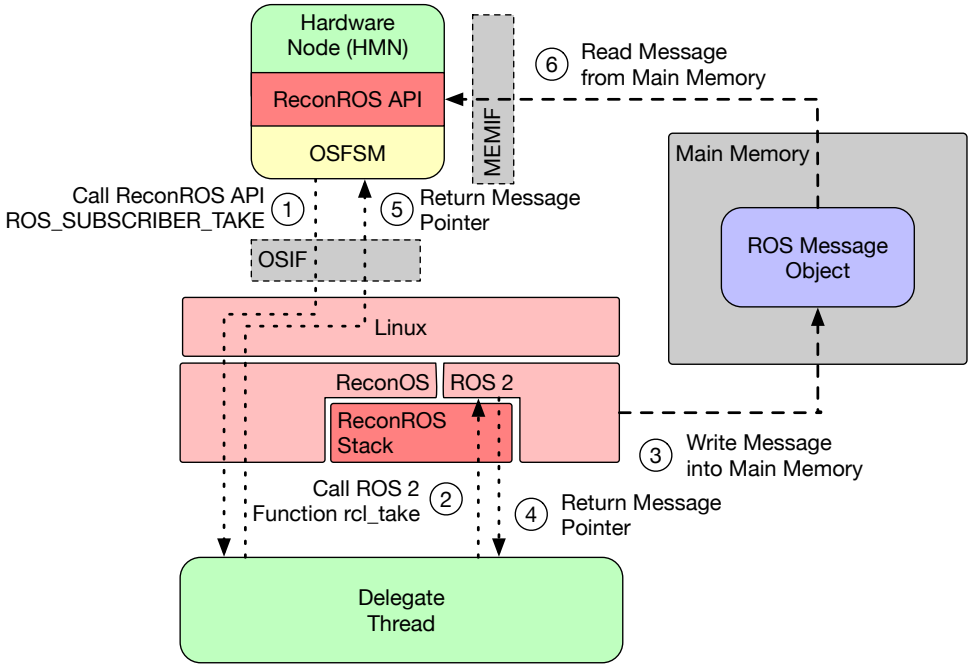

Figure 3.3: Sequence of events when a ROS 2 hardware-mapped node (HMN) calls the `ROS_SUBSCRIBER_TAKE` function from the ReconROS API. Adapted from [53].

Figure 3.3 exemplifies the sequence of events when a hardware ROS 2 node initiates a `ROS_SUBSCRIBER_TAKE` operation from the ReconROS API ①. The function call of the hardware thread includes the command for this API function and a reference to the subscriber. The command is transmitted by the OSFSM and unblocks the corresponding delegate thread on the CPU. The delegate then executes the ROS 2 subscriber take function `rcl_take` on behalf of the hardware thread ②. When a message for the subscribed topic becomes available, the ReconROS stack stores it in main memory ③ and unblocked the delegate thread ④, which in turn sends the message pointer via the OSIF back to the hardware thread ①. Subsequently, the hardware thread can read the message via its MEMIF ⑥. Publishing a message from a hardware thread works analogously: First, the hardware threads store the message in the main memory. Then, it sends a `ROS_publish` command and the message pointer via the OSIF to its delegate thread, which executes the command.

Regarding realizing ROS 2 nodes using reconfigurable hardware, we have considered static designs only so far. As a result, ROS 2 HMNs have to be statically placed in reconfigurable logic, where they remain until the application terminates. From the application designer's perspective, the HMNs usually run in $while(1)$-loops that start with blocking reads for new input data, process the data, and write the output.

In order to support a more advanced operating mode, the ReconROS executor extends ReconROS by supporting dynamic partial reconfiguration during runtime. Figure 3.4 highlights the resulting hardware architectural changes.

According to the underlying ReconOS architecture, the programmable logic part of a platform FPGA contains a set of n reconfigurable slots that can accommodate hardware threads during runtime. The number and sizes of these reconfigurable slots are application-specific and, therefore, configured during the design process. Each such reconfigurable slot is connected to an OSIF for communication with the host operating system Linux running on the processor cores and to a MEMIF for accessing shared external memory.

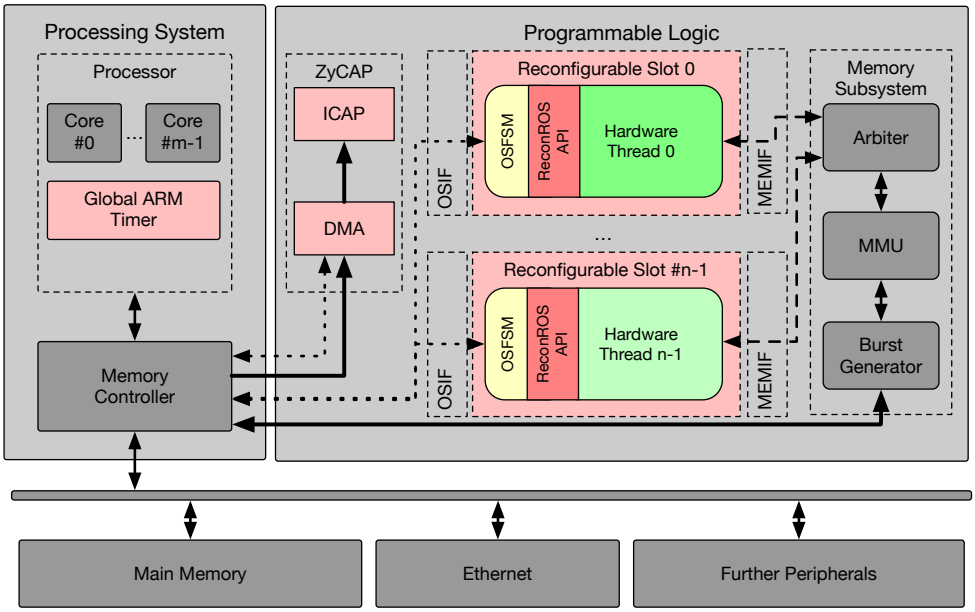

Figure 3.4: Hardware architecture for the ReconROS executor. Architectural changes or components that are now used are marked red. Adapted from [51] and [53].

Hardware threads are loaded into reconfigurable slots on demand during runtime with a partial reconfiguration process utilizing the ZyCAP implementation [121]. ZyCAP comprises an internal configuration access port (ICAP) interface and a direct memory access (DMA) block on the hardware side and a Linux kernel driver and user libraries on the software side. Library functions are available to load reconfigurable slots by setting up DMA transfers from external memory to the ICAP interface.

We have chosen ZyCAP over using the processor configuration access port (PCAP) or the ICAP directly for two reasons: First, ZyCAP nominally

features 3× higher performance for writing bitstreams, i.e., 382 MByte/s for the ZyCAP compared to 128 MBytes/s for PCAP [121]. Second, ZyCAP includes a DMA controller that lowers CPU load for partial reconfiguration and, in turn, frees the CPU for executing other components of the robotics application. In the hardware architecture, the ZyCAP block is connected to the processing system (PS) via a high-performance port (HPx) to transfer the bitstream and to an AXI-Lite interface for the configuration of DMA transactions. From the existing ZyCAP project, we have simplified the user library functions to the three primary functions `ZyCAP_Init()` for the initialization, `ZyCAP_Write_Bitstream()` for blocking write of the bitstream into the ICAP, and `ZyCAP_DeInit()` for de-initialization, and integrated them into the ReconROS library. Additionally, we have adapted the ZyCAP Linux kernel driver to more recent Linux kernels.

3.3 ReconROS Design Flow

The design flow for a ReconROS application adapts the original ReconOS design flow [1] and is sketched in Figure 3.5. The flow starts with the specification of a ReconROS project comprising a project configuration file, the sources for software and hardware threads that represent the ROS 2 nodes, and the definition of message types used for the application.

The configuration file specifies the used ROS 2 objects with their dependencies, the ReconOS architecture including, in particular, the number of reconfigurable slots and the mapping of hardware threads to reconfigurable slots, and the settings for the build tool flow.

The basic element of each ReconROS application is the *rosnode* object, which represents a ROS 2 node in the network. A *rosnode* object can be extended by one or more communication objects, which can be a subscriber (*rossub*) or publisher (*rospub*) objects for specific topics in case of publish/-subscribe communication, service (*rossrvs* / *rossrvc*) objects for client-server communications, and action (*rosacts* / *rosactc*) objects for ROS 2 actions.

In addition, each of these extensions, i.e., publisher, subscriber, service, and action, requires a reference to an instance of a ROS message *rosmsg* of a specific type. Declarations of *rosmsg* objects include the communication type, a group, and the message type. For example, a specific message declaration could specify 'Image' as the message type, 'sensor_msgs' as a group, and publish/subscribe as the communication type.

Threads for ROS 2 software-mapped nodes (SMNs) can be developed in C/C++, and threads for ROS 2 HMNs in C/C++ for use with high-level synthesis or, alternatively, in VHDL or Verilog. Importantly, we provide the same ReconROS API for software and hardware threads, which greatly simplifies the creation of hardware-accelerated versions of software threads.

Based on the configuration file and the sources, the RDK creates the ReconROS binaries for the specific project.

The RDK command `export_msg` extracts information from the message package definition and creates a Colcon project [22], which is then compiled

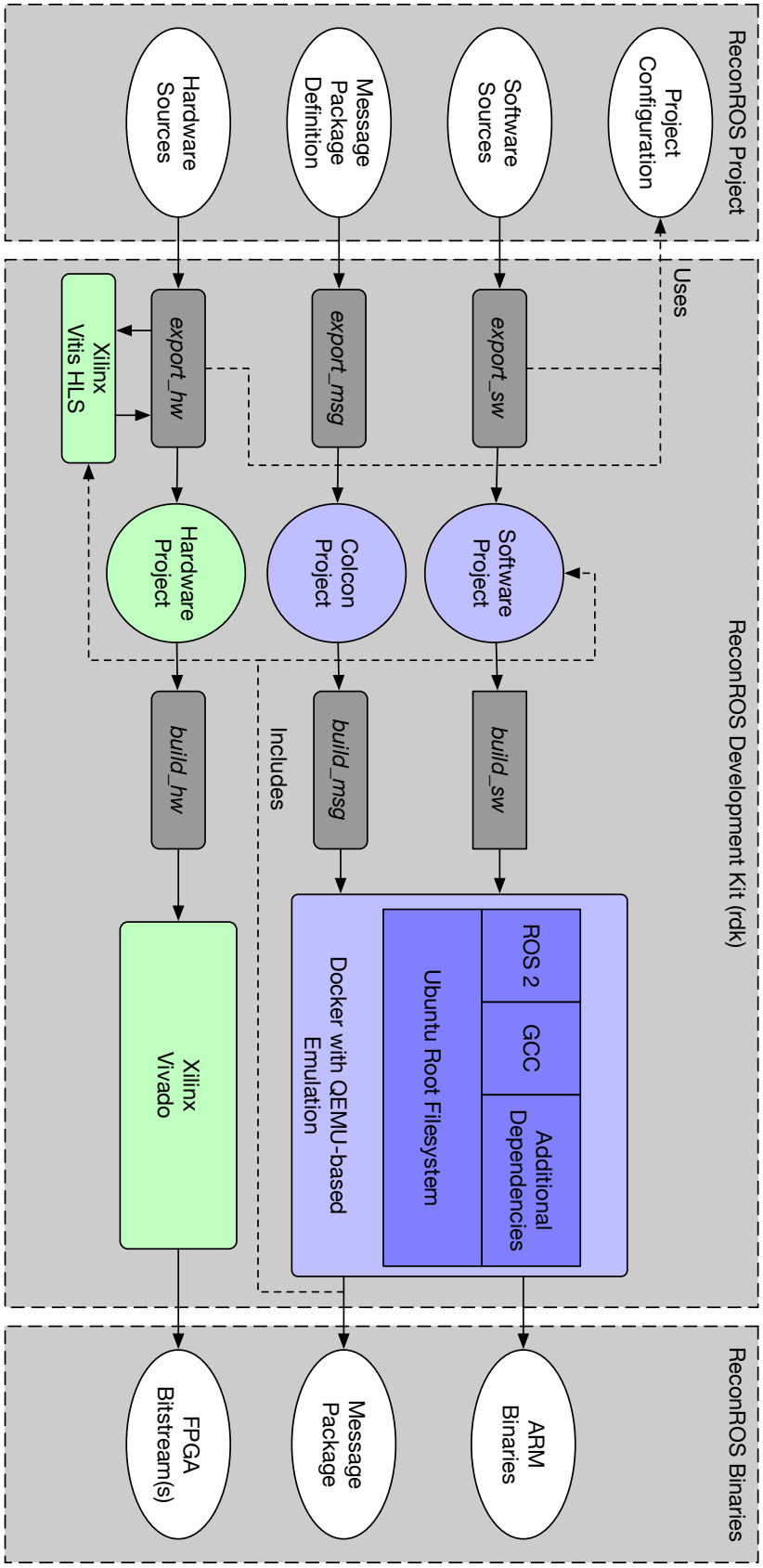

Figure 3.5: ReconROS design flow. Adapted from [50].

to the message package by the command `build_msg`. Colcon is a ROS 2 build tool, and the message package comprises message-related data and scripts that are used by the ROS 2 runtime. The RDK command `export_sw` creates the software project based on the sources for software threads and configuration data. The software project also includes the ReconOS delegate threads, all necessary initialization functions for the ReconOS primitives, and the ROS 2 middleware dependencies. Moreover, the software project includes header definitions for the messages, which are part of the compiled message package. Since we target Xilinx platform FPGAs of the Zynq-7000 and Zynq UltraScale+ series, which contain ARM Cortex cores, the RDK command `build_sw` creates binaries for the ARM architecture.

Both commands, `build_sw` and `build_msg` employ an ARM docker container (32-bit or 64-bit, depending on the target platform) emulated with QEMU [92] to build the binaries. Compared to a standard cross-compilation toolchain for the embedded ARM cores, our setup greatly simplifies the ROS 2 build step with all its dependencies since the package manager within the container can be used. Finally, the RDK command `export_hw` creates the hardware project based on the sources for hardware threads and configuration data. The hardware project contains the complete ReconROS architecture with its OSIFs, MEMIFs, and supporting modules. The command calls Xilinx Vivado HLS for high-level synthesis and thus also requires the message header definitions. The FPGA bitstream is then created by the RDK command `build_hw`.

3.4 PROGRAMMING MODEL

One of the main features of ReconROS is the consistent programming model across the hardware/software boundary. Since robotics designers are usually more experienced in software than hardware designing, the consistent programming model has become an essential aspect of the design of ReconROS.

The translation of the programming model in the hardware context is mainly done by extending ReconOS by ROS-related primitives in the ReconROS stack and supplying the ReconROS API for hardware and software source code. The other key component of this concept is the usage of high-level synthesis for hardware generation. Besides limitations such as dynamic memory allocation or pointer conversation for complex data structures, functional descriptions can be moved between HMNs and SMNs. However, it is worth mentioning that this does not result in performance-optimized designs. Hardware descriptions in high-level languages benefit from hints provided by additional source code annotations (pragmas), indicating room for performance optimization, e.g., by leveraging parallelism.

For the handling of messages in C/C++ code, ROS uses complex structs, which are shipped during the ROS 2 installation process by default. The Colcon tool flow (cf. Section 3.3) generates customized headers for custom message definitions. In general, high-level synthesis tools can include these

message headers in the tool flow and can handle read and write operations from and to single members of these structures. This allows for providing the semantic structure of messages both in software and hardware sources. However, as already indicated, the high-level synthesis tools have limitations regarding pointer operations, whereby the ease of transferring messages from the main memory to the FPGA and back suffers.

In order to still allow easy transfer of complete messages from the main memory to the FPGA, ReconROS automatically generates C macros for this purpose. To generate these macros, we extract the structure of the messages from their origin message definition files. The resulting structure comprises the member's name, e.g., width for an image datatype, and its type, e.g., int32. For a nested message with multiple levels, we substitute non-primitives with their set of members until non-primitive datatypes no longer occur. The resulting list is used to generate macros that read the members of the message member by member from the main memory or write them member-by-member into the main memory. The target message types are synthesized from the ReconROS project configuration file.

In the static form, ReconROS-based ROS 2 nodes are started in the applications startup phase and keep active until the application terminates. Regarding the programming model, ROS threads ran in $while(1)$-loops. Within the loop body, the thread blocks until new input data arrives, processes the input data, and outputs the results afterward. However, this processing schema can handle standard ROS-related tasks. This programming model is designed to keep the corresponding thread active runtime, making it unsuitable for event-based programming.

However, the programming model supported by the ReconROS executor is modified compared to the static programming model for ReconROS. The programming model for the executor generalizes the concept of callbacks for event-driven programming and introduces hardware callbacks. Following this concept, hardware callbacks are ReconROS threads assigned to specific events during initialization and invoked after the event happens. The result of the event, e.g., a received message in case of a subscription of a topic, is then accessible directly by the callback.

Listing 3.1 outlines an example callback after the message receiving running with the ReconROS executor. The message's origin can be, e.g., a subscription or a service request. The message is already available in the main memory when the callback is invoked. The ReconROS macro `THREAD_GETINITDATA()` returns the pointer to the message. This information can be used to process the message and eventually output results using different communication paradigms. The last statement `THREAD_EXIT()` signals the executor the end of the callback and allows for other threads to be executed.

Listing 3.1: Example HLS implementation of a simple hardware callback.

```
CALLBACK_ENTRY() {
    // Callbacks initial data contains a pointer to the message
    pMsg = THREAD_GETINITDATA();
    // Compute on the received message
    ..
    // Callback returns with THREAD_EXIT macro
    THREAD_EXIT();
}
```

3.4.1 *Static Execution Example*

As an example, we elaborate on a ROS 2 application comprising four nodes, which is shown in Figure 3.6. Node 1 captures images from a camera and publishes them to the topic */image_raw*. Node 2, the digital image processing node (DIP), subscribes to this topic, offloads the image processing to Node 3, the Sobel filter node (Sobel), and publishes the filtered images to the topic */image_filtered*. Node 4 reads and displays the filtered images. The data exchange between the Sobel and DIP nodes is done with a ROS 2 service called *sobel_service*. The ReconROS application comprises Nodes 2 and 3, where both are to be mapped to reconfigurable hardware and run on either a single or two FPGA platforms. Nodes 1 and 4 are assumed to exist or are being compiled with appropriate ROS 2 design flows to other target architectures, e.g., desktop PCs.

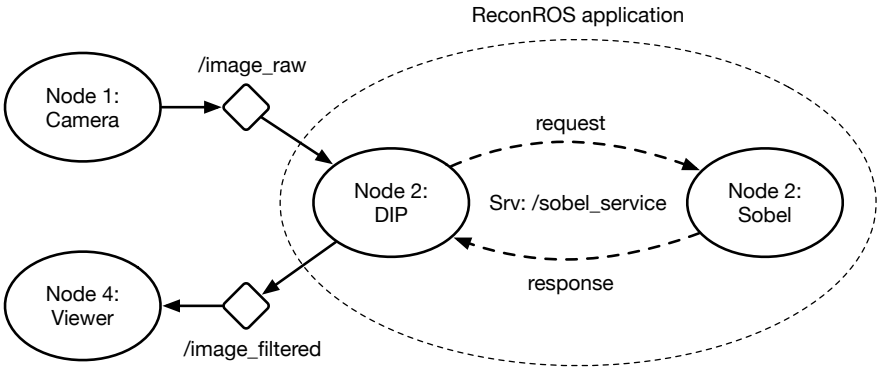

Figure 3.6: Example ROS 2 application including two ReconROS hardware nodes. Adapted from [50].

Listing 3.2: Configuration file (ROS 2-related part) for the ReconROS application shown in Figure 3.6. Taken from [50].

```
   [ResourceGroup(at)ResourceGroupSobel]
2  node_3 = rosnode, "Sobel"
   filter_service_msg = rossrvmsg, application_msgs, srv, SobelSrv
   filter_server = rossrvs, node_3, filter_service_msg, "sobelservice",
        10000

   [ResourceGroup(at)ResourceGroupDIP]
7  node_2 = rosnode, "DIP"
   filter_service_msg = rossrvmsg, application_msgs, srv, SobelSrv
   filter_client = rossrvc, node_2, filter_service_msg, "sobelservice",
        10000
   image_msg = rosmsg, sensor_msgs, msg, Image
   sub = rossub, node_2, image_msg, "/image_raw", 10000
12 pub = rospub, node_2, image_msg, "/image_filtered"
```

Listing 3.5 shows the ROS 2-related part of the configuration file for the Nodes 2 and 3. The information for the ROS 2 nodes is organized into so-called resource groups. Lines 1–4 specify node 3, beginning with defining a rosnode object named "Sobel" in line 2. In line 3, a message object of type ROS 2 service message is defined with further references to a ROS 2 message package and the communication as well as service types. Line 4 declares a ROS 2 server object for a ROS 2 service, connects it to the ROS 2 node *node_3* and the message object *filter_service_msg*, assigns the name "sobelservice" to it, and sets the polling time for checking for new service requests to 10000 μs.

Lines 6–12 specify node 2, including the rosnode object named "DIP", the same message object as used by node 2, and a client object for a ROS 2 service. Additionally, node 2 is extended with the message object *image_msg* of a ROS 2 built-in message type and corresponding subscriber and publisher objects for the topics */image_raw* and */image_filtered*.

Listing 3.3 presents C/C++ code for the HLS-implementation of the "Sobel" ROS 2 node. Using the ReconROS API, the processing loop starts in line 3 with a blocking read for a new service request. When a request becomes available, the function `ROS_SERVICESERVER_TAKE` returns a pointer to the service request data structure. With the help of the `OFFSETOF` macro, line 4 determines another pointer to the address of the request's payload. The macro `MEM_READ` is employed first to read the address of the image in line 7 and then to read the image into a ram structure within the FPGA in line 8. After a Sobel filter function is executed on the image in line 10, the result is written back to the main memory via the `MEM_WRITE` macro. Finally, the node sends the filtered data back to the node requesting the filter service (`ROS_SERVICESERVER_SEND_RESPONSE`). This code example shows the steps required to create a ReconROS application and focuses on simplicity rather than optimized performance. For example, overlapping processing

with memory transfers using a line buffer approach would be a natural optimization.

Listing 3.3: C/C++ code (partial) for the HLS implementation of the "Sobel" ROS 2 node. Taken from [50].

```
while(1) {
  // Wait for service request and get pointer to payload
  pMsg = ROS_SERVICESERVER_TAKE(resourcedip_srv,
      resourcedip_filter_srv_req);
  pMsg += OFFSETOF(application_msgs__srv__SobelSrv_Request, img.data.
      data);

  // Get a pointer to the image in main memory and copy it to FPGA-
      internal memory
  MEM_READ(pMsg, pPayloadService, sizeof(int));
  MEM_READ(pPayloadService[0], ram, IMAGE_SIZE * 4);

  //Apply the filter to the image
  SobelFilter(ram);

  // Write filtered image back to main memory and send service
      response
  MEM_WRITE(ram, pPayloadService[0], IMAGE_SIZE * 4);
  ROS_SERVICESERVER_SEND_RESPONSE(resourcedip_srv,
      resourcedip_filter_srv_res);
}
```

Listing 3.4 displays a similar procedure for the "DIP" node, which is expanded with three communication objects, a subscriber object for the topic */image_raw*, a client object for the service*/sobel_service*, and a publisher object for the topic */image_filtered*.

Listing 3.4: C/C++ code (partial) for the HLS implementation of the "DIP" ROS 2 node. Taken from [50].

```
while(1) {
 // Wait for the published image and get the pointer to the payload
     via OFFSETOF
 pMsg = ROS_SUBSCRIBER_TAKE(resourcesobel_subdata,
     resourcesobel_image_msg);
 pMsg += OFFSETOF(sensor_msgs__msg__Image, data.data);

 // Get a pointer to the image in main memory and copy it to FPGA-
     internal memory
 MEM_READ(pMsg, pPayloadPubSub, sizeof(int));
 MEM_READ(pPayloadPubSub[0], ram, IMAGE_SIZE * 4);

 // Request filter service, pServiceRequest is set up during
     initialization
 MEM_WRITE(ram, pServiceRequest[0], IMAGE_SIZE * 4);
 ROS_SERVICECLIENT_SEND_REQUEST(resourcesobel_srv,
     resourcesobel_filter_srv_req);

 // Wait for service response and get the pointer to the payload
 pMsg = ROS_SERVICECLIENT_TAKE(resourcesobel_srv,
     resourcesobel_filter_srv_res);
 pMsg += OFFSETOF(application_msgs__srv__SobelSrv_Response, img.data.
     data);

 // Get a pointer to the payload and copy it to FPGA-internal memory
 MEM_READ(pMsg, pPayloadService, sizeof(int));
 MEM_READ(pPayloadService[0], ram, IMAGE_SIZE * 4);

 // Write filtered image back to memory and publish it
 MEM_WRITE(ram, pPayloadPubSub[0], IMAGE_SIZE * 4);
 ROS_PUBLISHER_PUBLISH(resourcesobel_pubdata, resourcesobel_image_msg
     );
}
```

3.4.2 *Dynamic Execution Example*

Figure 3.7 outlines an example of a computation graph using publish-subscribe communication. The architecture is inspired by the auto race challenge for the Turtlebot 3 robot [116], in which a robot has to deal with different challenges along a predefined racetrack. The proposed ROS architecture can follow a street line, park the robot on a marked parking lot, or navigate the robot in a (illuminated) tunnel. The ROS node */camera* captures images from the robot camera and publishes them to the topic */image*. The ROS 2 image processing package */image_proc* subscribes to this topic and publishes the rectified images to the topic */image_rectified_color*.

The */state_ctrl* node implements a robotic state machine comprising three states for a lane following mode, a tunnel navigation mode, and a parking mode. Depending on its actual state, the */state_ctrl* node forwards the input image data to one of the three nodes */lane_following*, */tunnel_navigation*, */parking*. Each node subscribes to a separate image topic and derives proper driving commands for the */ctrl* topic. After task completion, the */state_ctrl* node receives feedback data subscribing to the corresponding */*_done* topic.

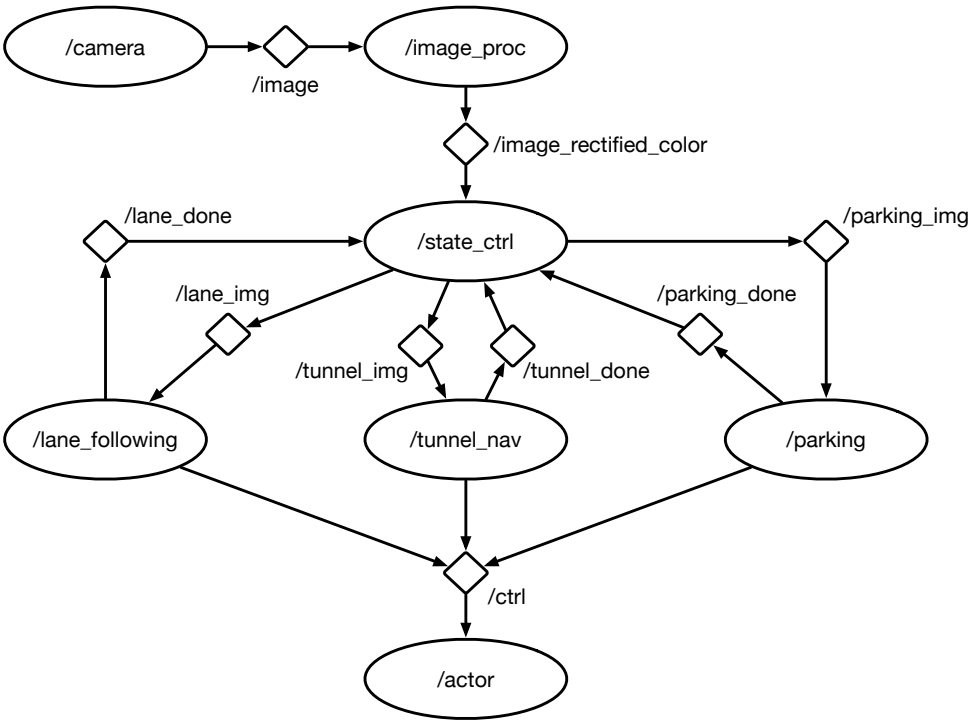

Figure 3.7: ROS computational graph example for demonstrating programming for dynamic execution. Adapted from [51].

As an example application for the ReconROS executor, we elaborate on the ROS 2 application from Figure 3.7. The ReconROS application from this design example handles the three nodes */lane_detection*, */tunnel_nav* and */inv_parking*. The remaining nodes are assumed to be developed and compiled with ROS 2 design flows and to be eventually mapped to other computing platforms.

All three considered ROS 2 nodes comprise subscriber functionality for getting input data. According to the event-based programming approach, callbacks are invoked after the arrival of messages. In this example, the three callbacks are implemented in hardware and designed for execution in a reconfigurable slot. In case the reconfigurable fabric can not accommodate all three implementations simultaneously due to resource limitations, dynamic partial reconfiguration must be used to configure and execute some implementations sequentially. Since all three nodes are running exclusively until their task is completed, the overhead for the hardware reconfiguration is paid for only once per state change in */state_ctrl*. The process of recon-

figuration itself is invoked by the */state_ctrl* node by publishing data to the corresponding input topic of one of the three considered nodes.

Listing 3.5: Configuration file (Partial reconfiguration / ROS 2 related part) for the ReconROS application shown in Figure 3.7. Taken from [51].

```
[HwSlot(at)ReconfSlot(0:0)]
Id = 0
Reconfigurable = true
Region0 = SLICE_X0Y150SLICE_X103Y199, DSP48_X0Y60DSP48_X7Y79,
    RAMB18_X0Y60RAMB18_X5Y79, RAMB36_X0Y30RAMB36_X5Y39

[ResourceGroup(at)RGLaneFollowing]
node = rosnode, "/lane_following"
imag_msg = rosmsg, sensor_msgs, msg, Image
done_msg = rosmsg, std_msgs, msg, Uint8
ctrl_msg = rosmsg, TurtleBot, msg, Control
sub_imag = rossub, node, img_msg, "/lane_img"
pub_done = rospub, node, done_msg, "/lane_done"
pub_ctrl = rospub, node, ctrl_msg, "/ctrl"

[ResourceGroup(at)RGTunnelNav]
node = rosnode, "/tunnel_nav"
imag_msg = rosmsg, sensor_msgs, msg, Image
done_msg = rosmsg, std_msgs, msg, Uint8
ctrl_msg = rosmsg, TurtleBot, msg, Control
sub_imag = rossub, node, img_msg, "/tunnel_img"
pub_done = rospub, node, done_msg, "/tunnel_done"
pub_ctrl = rospub, node, ctrl_msg, "/ctrl"

[ResourceGroup(at)RGParking]
node = rosnode, "/parking"
imag_msg = rosmsg, sensor_msgs, msg, Image
done_msg = rosmsg, std_msgs, msg, Uint8
ctrl_msg = rosmsg, TurtleBot, msg, Control
sub_imag = rossub, node, img_msg, "/parking_img"
pub_done = rospub, node, done_msg,"/parking_done"
pub_ctrl = rospub, node, ctrl_msg, "/ctrl"
```

Listing 3.5 shows the ROS 2 related part of the ReconROS configuration file for the overall ReconROS design project. The file starts with a block for the specification of the reconfigurable slot, which will accommodate the hardware callbacks. Lists of contiguous resources do the specification for each type (configurable logic block (CLB), or look-up table (LUT) slices respectively, DSP, BRAM18, BRAM36). These lists can be derived by drawing pblocks using the FPGA design tool flow Xilinx Vivado and by reading the resulting constraints. Templates with predefined regions are available to help users with limited knowledge in this field. The information for the ROS 2 nodes is organized into so-called resource groups. Lines 6–13 specify the */lane_following* node, beginning with defining a rosnode object named "/lane_following" in line 7. In lines 8–10, the message objects used by the

node are defined with further references to a ROS 2 message package (e.g., *sensor_msgs*) and the communication (*msg*) as well as message types (e.g., *image*). Lines 11 – 13 declare primitives for the subscription of input data from topic */lane_img* and the publication of control and feedback data to topic */lane_done* and */ctrl*.

Lines 15–22 specify the */tunnel_nav* node and lines 24 – 31 the */parking* node, including the rosnode object, message declarations, and the attachment of publisher and subscriber to the rosnode.

Listing 3.6: C/C++ code (partial) for the HLS implementation of the subscriber callback for the */filter* ROS 2 node. Taken from [51].

```
1  // Initial data contains a pointer to the message
   pMsg = THREAD_GETINITDATA();

4  // Calculate the offset in the message structure via OFFSETOF
   pMsg += OFFSETOF(sensor_msgs__msg__Image,
       data.data);

   // Get a pointer to the image in memory and
9  // copy it to FPGA-internal BRAM
   MEM_READ(pMsg, pPayloadImage, sizeof(int));
   MEM_READ(pPayloadImage[0], ram,
       IMAGE_SIZE * 4);

14 // Process on new data
   // done and ctrl are pre-allocated arrays
   done[0] = lane_following(ram, ctrl);

   // Write results to main memory
19 MEM_WRITE(ctrl, pCtrl, sizeof(int));
   MEM_WRITE(done, pDone, sizeof(int));

   // Publish results from main memory
   ROS_PUBLISHER_PUBLISH(rglanefollowing_pub_ctrl,
24 rglanefollowing_ctrl_msg);

   ROS_PUBLISHER_PUBLISH(rglanefollowing_pub_done,
       rglanefollowing_done_msg);

29 // Callback termination
   THREAD_EXIT();
```

In Listing 3.6, C/C++ code for the HLS implementation of the subscriber callback of the */lane_following* node is presented. The presented code is not performance-optimized. The callback starts with accessing the initial data of the callback, which provides a pointer to the message object in the main memory. At this point, the message is already in the memory and ready for access. The position of the image data is calculated using the `OFFSETOF` macro in line 3. Using the resulting pointer, the first use of `MEM_READ` macro reads the address of the image, and then, with the second use of the macro,

the callback reads the image into the *ram* memory within in FPGA. After the execution of the lane following function on its actual input data, the callback writes resulting control data and the feedback data to main memory via the MEM_WRITE macro. After that, the callback publishes the filtered image using the node-related publisher.

The other two nodes considered in this design example are implemented equally, except for the processing function and the names of publisher and message instances. Again, the data received from the subscriber is loaded into the internal memory of the FPGA and processed. The results are written back to the main memory before publishing to the output topics.

Listing 3.7 displays a similar procedure for the */parking* node, which basically relies on the same procedure as described in Listing 3.6. Similar to the previous procedure, data is read into the FPGA, processed, and written back. However, instead of accessing ROS message items individually, the automatically generated message read macro `MEM_READ_ROS_MESSAGE_*` has been used. This message transfer macro was generated for the *image* message type from the *sensor_msgs* ROS 2 package (cf. Section 3.4).

Listing 3.7: C/C++ code (partial) for the HLS implementation of the subscriber callback for the */parking* ROS 2 node.

```
    // Initial data contains a pointer to the message
    pMsg = THREAD_GETINITDATA();

    // Copy the complete message into BRAM
5   MEM_READ_ROS_MESSAGE_sensor_msgs_msg_Image(pMsg, ImageMsg);

    // Process on new data
    // done and ctrl are pre-allocated arrays
    done[0] = parking(ImageMsg->data.data, ctrl);
10
    // Write results to main memory
    MEM_WRITE(ctrl, pCtrl, sizeof(int));
    MEM_WRITE(done, pDone, sizeof(int));

15  // Publish results from main memory
    ROS_PUBLISHER_PUBLISH(rgparking_pub_ctrl,
    rgparking_ctrl_msg);

    ROS_PUBLISHER_PUBLISH(rgparking_pub_done,
20      rgparking_done_msg);

    // Callback termination
    THREAD_EXIT();
```

The last needed user-created file for this example application is the main.c file, in which the ReconROS executor is instantiated and configured. Listing 3.8 shows the needed steps. In line 2, the ReconROS executor is initialized for execution without software workers but with one hardware worker using the `ReconROS_Executor_Init` function. The fourth argument

for calling that function is the path to the partial bitstreams in the local file system. In lines 3–5, the hardware callbacks are registered at the executor. The list of arguments comprises the executor instance, the ROS 2 node name, the *ResourceMask*, the ReconROS primitive type, the callback-creating ReconROS primitive instance, and the ReconROS target message primitive. The last line of the code spins the executor and blocks until the application is terminated. The programming interface of the ReconROS executor is oriented on existing solutions such as the standard ROS client library for C++ (rclcpp) or the rcl executor.

Listing 3.8: C/C++ code (partial) main thread of the ReconROS application. Taken from [51].

```
    // Initialize the ReconROS executor without SW workers and one HW
        worker
2   ReconROS_Executor_Init(&reconros_executor, 0, 1,
        "/mnt/bitstreams/");

    ReconROS_Executor_Add_HW_Callback(&reconros_executor,
        "/lane_following", 1, ReconROS_SUB,
7       rglanefollowing_sub_imag, rglanefollowing_img_msg);

    ReconROS_Executor_Add_HW_Callback(&reconros_executor,
        "/tunnel_nav",     1, ReconROS_SUB,
        rgtunnelnav_sub_imag,     rgtunnelnav_img_msg);
12
    ReconROS_Executor_Add_HW_Callback(&reconros_executor,
        "/parking",        1, ReconROS_SUB,
        rgparking_sub_imag,       rgparking_img_msg);

17  // Run the executor
    ReconROS_Executor_Spin(&reconros_executor);
```

3.5 EXPERIMENTAL EVALUATION

This section first evaluates the ReconROS architecture. The chapter starts evaluating the overhead for nodes being suspended due to the migration to hardware and continues to analyze the achieved performance for reconfiguring slots during runtime.

3.5.1 *Hardware-Mapped Node Overheads*

To characterize runtime overheads when mapping ROS 2 nodes to hardware instead of software and contrasting them to communication times within a ROS 2 network, we have implemented a ping-pong ReconROS application with two ROS 2 nodes distributed onto a desktop PC and a Mini-ITX 7Z100 board containing a Xilinx Zynq-7100 platform FPGA, connected via Gigabit Ethernet (GbE) as shown in Figure 3.8. The platform FPGA runs Ubuntu

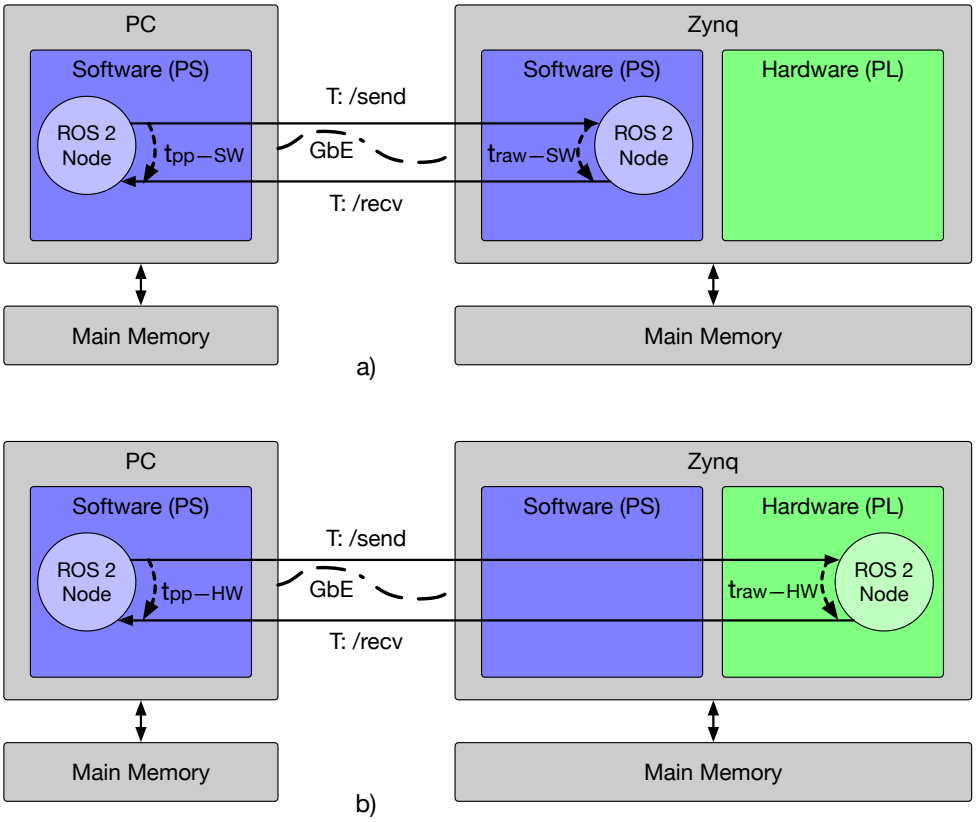

Figure 3.8: ReconROS ping-pong application for overhead estimation. Taken from [53].

18.04 and ReconROS based on ROS 2 dashing. All ROS 2 nodes use the same C/C++ source for software and hardware implementations. Software implementations have been compiled with optimizations level O3, and hardware implementations have been created with HLS without any optimizations. All reported runtimes are the result of averaging over 1000 executions.

The first experiment determines the basic overhead for mapping a ROS 2 node to hardware. It consists of an echo application, where the ROS 2 node on the PC publishes messages to the topic */send* and the ROS 2 node on the Zynq subscribes to this topic, receives messages in local memory and publishes them to the topic */recv*. Table 3.2 presents the runtimes for the echo tasks in software and hardware, $t_{pp-echo-SW}$ and $t_{pp-echo-HW}$, measured as $t_{pp} = t_{end} - t_{start}$ on the PC as indicated in Figure 3.8, and the resulting speedup $S_{pp-echo}$.

The echo SMN performs no operations except calling subscribe/publish functions. In order to implement the same behavior, the echo HMN needs ReconOS signaling to communicate between the underlying hardware thread and the software-bound delegate thread. Since only pointers to messages and identifiers for the topics and the message are passed, the echo nodes exhibit a runtime independent of the message size. As Table 3.2 shows, for the minimal message size of 4 Byte, there is a measurable slowdown due to the ReconOS signaling, but for larger message sizes, this overhead is

Message size	$t_{pp-echo-SW}$ [ms]	$t_{pp-echo-HW}$ [ms]	$S_{pp-echo}$
4 Byte	0.81	1.2	0.68×
8 KiB	10.65	10.48	1.02×
1 MiB	52.21	52.16	1.01×
6 MiB	363.91	363.30	1.00×
10 MiB	630.37	624.02	1.01×

Table 3.2: Runtimes and speedups for the echo ping-pong application. Taken from [53].

completely hidden behind the communication times. It has to be noted that mapping a ROS 2 node to hardware reduces the load on the CPU, and this can become a source for additional speedups for the overall ROS 2 applications. Such an effect, albeit very small, can be observed in the echo experiment, where some speedups are slightly more significant than others.

The second experiment is a copy application that evaluates the memory read/write performance of ROS 2 HMNs. The difference to the echo application is that the Zynq-bound ROS 2 nodes create a copy of the message in local memory before publishing to topic T:/recv. Table 3.3 presents the runtimes for the raw copy tasks in software and hardware, $t_{raw-copy-SW}$ and $t_{raw-copy-HW}$, and the resulting raw speedup $S_{raw-copy}$, as well as the runtimes for the overall copy ping-pong application, $t_{pp-copy-SW}$, $t_{pp-copy-HW}$, and the resulting speedup $S_{pp-copy}$ for different message sizes.

Message size	$t_{raw-copy-SW}$ [ms]	$t_{raw-copy-HW}$ [ms]	$S_{raw-copy}$
4 Byte	0.01	0.01	1.00×
8 KiB	0.03	0.13	0.23×
1 MiB	3.59	12.81	0.28×
6 MiB	18.91	76.35	0.25×
10 MiB	31.54	127.19	0.25×

Table 3.3: Runtimes for the raw copy ROS 2 nodes in software and hardware. Taken from [53].

Since the underlying ReconOS implementation has a lower memory bandwidth than Zynq's ARM processor subsystem, we observe a slowdown for the raw ROS 2 hardware copy node, distinct for larger message sizes and saturates at about 0.25. Thus, copying a message of 10 MiB is about 4× slower in hardware than in software. While improving ReconOS' memory subsystem would obviously improve the situation, Table 3.4 also shows that for the overall copy ping-pong application where we have to take commu-

Message size	$t_{pp-copy-SW}$ [ms]	$t_{pp-copy-HW}$ [ms]	$S_{pp-copy}$
4 Byte	1.69	1.71	0.99×
8 KiB	11.39	10.78	1.06×
1 MiB	58.71	66.25	0.89×
6 MiB	381.44	438.03	0.87×
10 MiB	643.47	735.30	0.86×

Table 3.4: Runtimes for the overall copy ping-pong application and corresponding speedups. Taken from [53].

nication into account, the slowdown is less pronounced and saturates at around 0.86. Again, due to the effects of the underlying software stacks of Linux, ROS 2, and ReconOS, and the possible parallel execution of hardware and software threads, the speedups are not consistently decreasing, and for 8 KiB the speedup is even more considerable than one.

Related work [109] has also reported on measured communication times between a ROS node on a PC and a ROS node on an ARM/Zynq connected with Gigabit Ethernet. For one-way communication, the authors determined approximately 60*ms* for a 1 MiB message and approximately 275*ms* for a 6 MiB message. Comparing with the corresponding data points of Table 3.2, which are for two-way communication, we see that ReconROS achieves higher performance, albeit on a different ROS version.

3.5.2 *Reconfiguration Overheads*

For quantizing the reconfiguration time, we have created a ReconROS setup with four reconfigurable slots, RS #0, . . . , RS #3. Table 3.5 shows the number of available resources per reconfigurable slot, the resulting bitstream size S, and the measured reconfiguration times t_{rc} for the four reconfigurable slots.

Reconfigurable slot	Slice LUTs	DSPs	BRAMs (36 / 18)	S [Byte]	t_{rc} [ms]
RS #0	20800	160	60 / 120	2838976	24.0
RS #1	20800	160	60 / 120	2838976	24.0
RS #2	41600	320	240 / 120	5285728	38.4
RS #3	40800	280	200 / 100	4883328	36.9

Table 3.5: Reconfiguration slots with resources (Z7100 slices LUTs, DSPs, and BRAMs), bitstream size and reconfiguration time. Taken from [51].

Using linear regression on the measured reconfiguration times and a reconfiguration time model that includes a constant offset part t_{offset} and

a bitstream size dependent part S/B, where B denotes the transfer bandwidth, i.e., $t_{rc} = S/B + t_{offset}$, our measurements result in $t_{offset} = 6.8ms$ and $B \approx 160MByte/s$. The achieved bandwidth is much lower than the results reported in [121]. The authors of [121] apparently used a bare-metal implementation of the ZyCAP driver without an operating system. Our current implementation suffers from copying the bitstream between the user and kernel space. An improved implementation of the Linux driver with a zero-copy approach, e.g., based on `get_user_pages`, would increase the performance.

The reconfiguration times reported in Table 3.5 are directly dependent on the size of the reconfigurable slot and the corresponding bitstream size. However, they are basically independent of the hardware callback's functionality. The reconfiguration time adds to the execution time of a hardware callback only if the targeted reconfigurable slot is not yet configured with the required bitstream.

The results of the measurements indicate that hardware callbacks are best suited for longer-running periodic tasks with frequencies up to a few tens of hertz or sporadic tasks. However, performance improvements in the Linux driver would increase the applicability of hardware callbacks.

3.6 CHAPTER CONCLUSION

In this chapter, we have shown the architecture and implementation of the ReconROS framework. The design considerations compare our approach with related approaches from the literature and demonstrate the advantages of ReconROS compared to other approaches. The architecture of ReconROS enables the implementation of complete ROS 2 nodes in hardware. Regarding the programming model, ReconROS supports static and dynamic execution using event-driving programming. The consistent programming model allows non-hardware experts to develop HMNs on the one hand and enables easy migration of nodes from hardware to software and vice versa on the other. Further advantages for developing HMNs result from the features that ReconROS has inherited from ReconOS, such as virtual memory access.

The evaluation in this chapter quantifies HMN overheads for implementing ROS 2 nodes in hardware and overheads due to dynamic partial reconfiguration. Both overhead types have to be considered during the application's design time. However, the next Chapter 4 shows that despite the overheads, significant speedups can be achieved through hardware acceleration.

4

TASK MAPPING AND PARALLELISM IN RECONROS

In the last chapter, we have presented the architecture, design flow, and programming model of the ReconROS framework, including extensions for the ReconROS executor on the hardware level.

In this chapter, we are going a step further and present approaches to map ROS applications on the node level, either to reconfigurable hardware or software. Node-level in this context means that the mapping of nodes is considered individually per node and not in combination with other nodes or the communication between nodes. The ReconROS framework supports two different modes of task mapping: static and dynamic task mapping.

As described in the following, static task mapping excels in runtime performance since it does not require any hardware reconfiguration during runtime. Typical use cases for static mapping are thus tasks such as image pre-filtering of camera data or sensor reading, tasks that receive huge amounts of raw data and come with high demands on processing throughput and latency since they are placed in lower levels of a robot's processing pipeline. These requirements may prevent utilizing dynamic partial reconfiguration because of the reconfiguration time overheads involved. While static mapping is natively supported by ReconROS and thus represents the default operating mode, dynamic placement requires the addition of the ReconROS executor, which handles scheduling and mapping during runtime. Dynamic task mapping involves reconfiguration time overhead but provides more flexibility in managing the reconfigurable resource, as hardware tasks are only configured when needed.

This chapter is divided into four parts: The first, Section 4.1 describes the static mapping of tasks using ReconROS. The second, Section 4.2 presents the dynamic mapping of tasks during runtime and includes the scheduling and placement strategies of the ReconROS executor. In the following, Section 4.3 outlines the exploitation of parallelism using ReconROS. Finally, the last Section 4.4 reports about experiments for both mapping strategies.

This chapter mainly follows the IEEE International Conference on Robotic Computing (IRC) conference publication. Additionally, parts of the static mapping have already been presented in the FPT conference publication [53], and some parts of the dynamic mapping follow the DSD conference publication [51].

4.1 STATIC TASK MAPPING

The primary task mapping approach is the static task mapping scheme in ReconROS resulting in software-mapped nodes (SMNs) and hardware-mapped nodes (HMNs). Figure 4.1 shows an overview of the mapping. Each of the n nodes in the ROS 2 programming model is implemented by one thread in ReconROS, either a software thread (SWT) or a hardware thread (HWT). Since ReconROS software threads are standard POSIX threads, they are managed by the operating system's scheduler and assigned to one of the processor's c CPU cores. However, the software threads can also be pinned to specific cores using operating system services for thread distribution. For example, Linux provides an interface that allows setting the affinity for threads to specific cores and running threads exclusively on selected CPU cores.

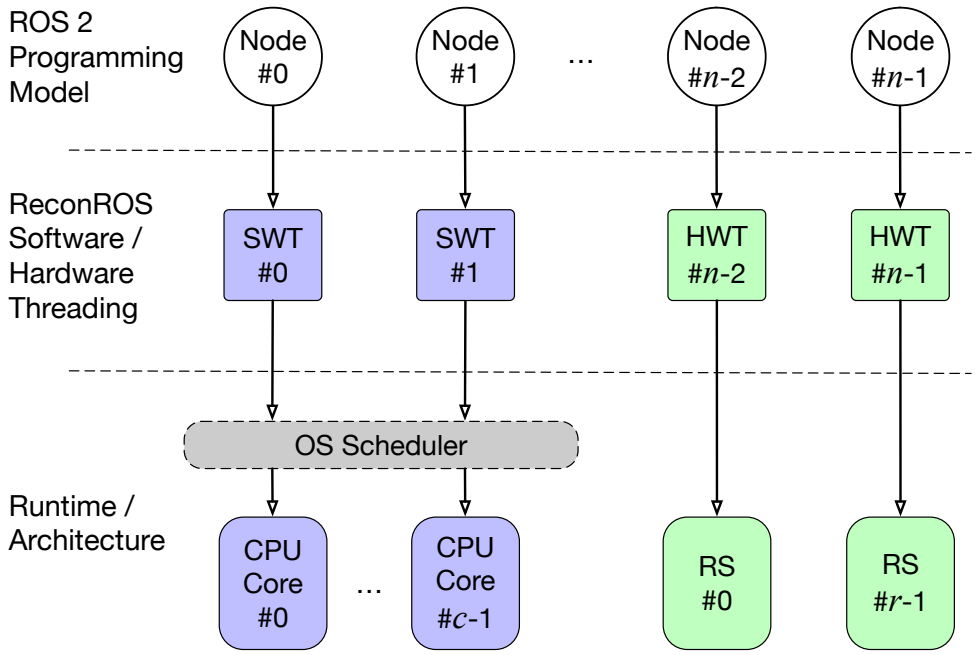

Figure 4.1: Static task mapping of ROS 2 nodes to CPU cores (in purple) and reconfigurable hardware slots (in green) using ReconROS. Taken from [52].

ReconROS hardware threads are assigned to one of the r RSs by the vendor's toolchain during design time. In the static mapping approach, there is actually no need to define reconfigurable slots as rectangular areas with physically constrained borders in the logic fabric. One can leave the logic placement on the overall FPGA to the vendor tools, which results in two advantages: First, no user-constrained placement increases resource utilization since there is no internal fragmentation, i.e., no unused areas in the reconfigurable slots in case the hardware thread does not need all resources available in the reconfigurable slot. Second, free placement leads to faster designs since, due to the larger optimization space, vendor tools typically find placements with improved timing.

4.2 DYNAMIC TASK MAPPING

The dynamic task mapping scheme in ReconROS is enabled by the ReconROS executor. In a previous Section 3.2 of this thesis, the hardware extensions on the ReconROS architecture for enabling dynamic partial reconfiguration during runtime have already been introduced. However, for the complete description of the executor, the questions of the scheduling (which function should be executed next?) and replacement (where should it be executed?) need to be answered.

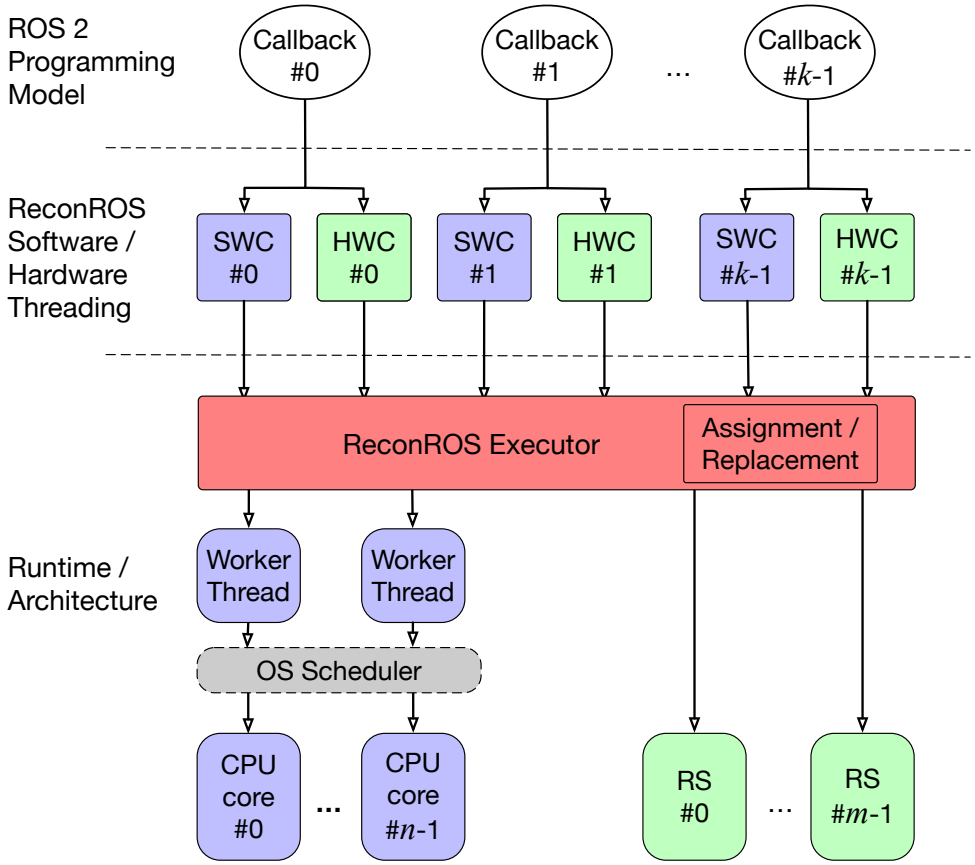

Figure 4.2: Dynamic task mapping of ROS 2 callbacks to CPU cores (in purple) and reconfigurable hardware slots (in green) using ReconROS with an executor and reconfigurable slot assignment/replacement strategy (in red). Taken from [52].

Figure 4.2 shows an overview of the mapping. Using programming language interfaces in C++ or Python, a ROS 2 node is decomposed into a set of callbacks. Following the ROS 2 event-based programming approach, specific events, e.g., receiving a new message, lead to the execution of callbacks.

Implementing these callbacks results in ReconROS software callbacks (SWCs) and hardware callbacks (HWCs), which are essentially ReconROS threads. In Figure 4.2, each ROS 2 callback is implemented as a software and a hardware callback. Implementing both in hardware and software allows for advanced task mapping where the scheduler can potentially decide

whether to execute an invocation of a callback in software or hardware at runtime.

SWCs and HWCs are registered at the ReconROS executor during application startup. The scheduling algorithm of the executor selects the callback to be executed next and assigns it to an available resource, i.e., a CPU core or a reconfigurable slot. Technically, the executor uses software worker threads that – when assigned – execute the callback function. Using worker threads instead of separate threads leads to lower runtime overheads for callback execution than separate thread creation for each invocation callback. The scheduler of the underlying operating system handles the worker threads and assigns them to the available CPU cores.

In contrast to software callbacks, which are quickly assigned to CPU cores, the start of hardware callbacks is more involved. After a reconfigurable slot has been selected for a hardware callback, the ReconROS executor initiates writing the partial configuration bitstream to the FPGA using the DMA controller and the interface (cmp. Section 3.2). This process takes considerably more time, in the order of *ms*, than loading a software callback onto a CPU core. Moreover, the exact time needed for reconfiguration depends on the size of the reconfigurable slot.

4.2.1 *Scheduling*

The main steps in designing the ReconROS executor are providing timers and creating a scheduling or dispatching algorithm, respectively, that utilizes all available processor cores and reconfigurable slots for callbacks.

In the ROS 2 stack, timers are part of the high-level libraries rclcpp (C++) or ROS client library for Python (rclpy) (Python) and use the operating system to measure wall clock time. Since ReconROS builds on rcl, the underlying standard framework for ROS primitives, we have added a corresponding timer primitive. Our implementation uses the ARM Cortex-A9 global timer (cf. Figure 3.4) as its primary time reference, and a low-overhead function `ros_timer_is_ready()` to check whether a time interval has expired.

Developing a generalized executor concept algorithm is more challenging than executor design for standard software-based applications. In contrast to the standard ROS 2 executor (cf. Section 2.2.3) that dispatches ready-to-execute callbacks to several identical software worker threads, typically one per available processor core, the ReconROS executor can either execute callbacks in software or hardware and, if executed in hardware, in specified reconfigurable slots. Therefore, our executor implementation is structured into an executor main thread, one software worker thread per processor core, and one hardware worker thread per reconfigurable slot. The main thread maintains four callback lists that include all callbacks registered at the executor, i.e., one for timers, one for subscribers, one for service servers, and the last one for service clients. Each callback list entry comprises a unique identifier, a pointer to the received message in case of non-timer callbacks, and a *ResourceMask* containing a field for software and each reconfigurable

slot. If the execution mode is software, the corresponding field includes a function pointer to the callback code. If the execution mode is hardware, the corresponding fields contain pointers to callback bitstreams for the reconfigurable slots.

The overall m software and n hardware worker threads are started during the initialization of the executor. Each of these threads implements the inner loop of Figure 2.6. Figure 4.3 displays the functionality of the hardware worker thread for reconfigurable slot x. The thread accesses the callback lists in the order of timers, subscribers, service servers, and service clients and searches for ready callbacks with a matching entry x in the *ResourceMask*. If such an entry is found (is not zero), the thread checks whether the corresponding bitstream is already loaded in the reconfigurable slot x. If so, the callback is started; otherwise, partial reconfiguration is performed to load the callback bitstream. The worker thread waits until the callback is finished and runs into the next loop iteration.

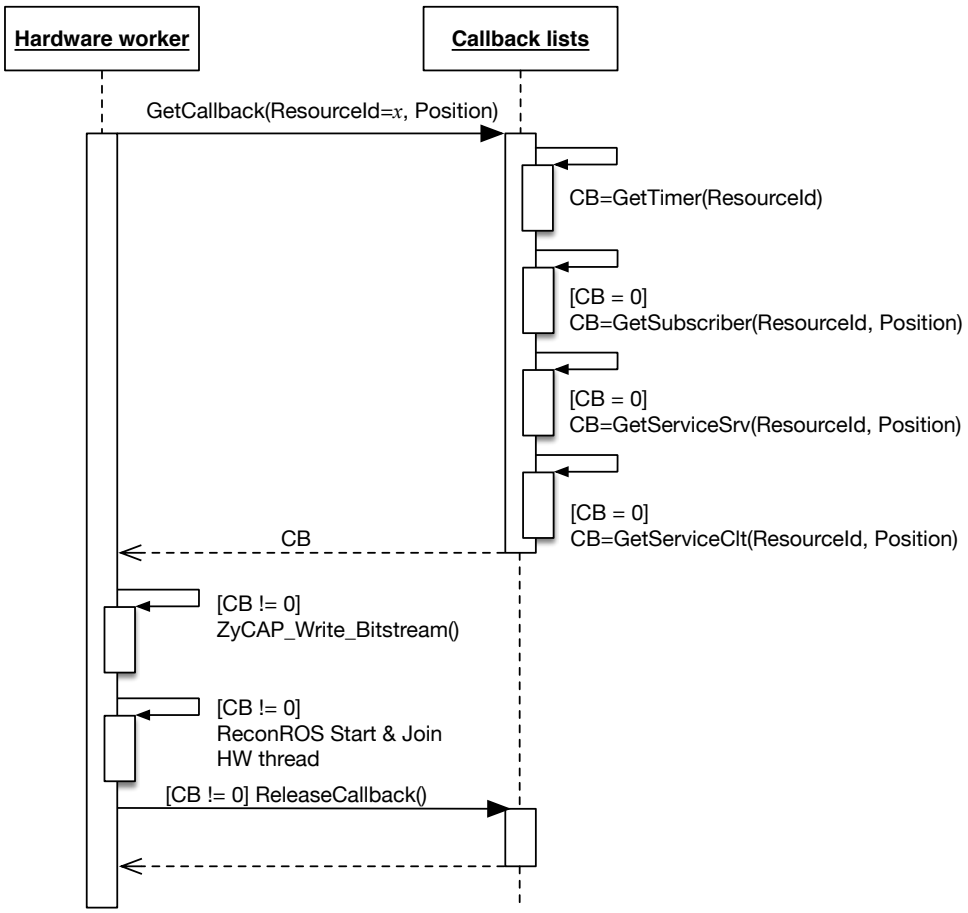

Figure 4.3: Sequence diagram for a hardware worker thread. Taken from [51].

The standard ROS 2 executor shown in Figure 2.6 collects ready non-timer callbacks before entering the inner loop. The gathering ensures that all callbacks collected up to a specific time will be executed before new non-timer callbacks are considered. Very frequently appearing subscriber callbacks,

for example, can thus not lead to starving callbacks for service servers and clients. Since our ReconROS executor uses more independent worker threads, we resort to a different mechanism to avoid starvation. Each worker thread maintains an *OffsetVector* that holds for each non-timer callback list an entry *Position* that identifies the callback last served. Whenever the worker thread checks the list for the next ready callback, it starts the search from *Position*+1. After serving the callback, *Position* is incremented. *Position* is initialized with the length of the list and wrapped around to zero when the end of the callback list is reached. Software worker threads are identical, except that they start callback functions in software.

The ReconROS executor tries to mimic the behavior of the ROS 2 standard executor and requires the designer to specify the size of the reconfigurable slots and, for each hardware callback, the possible reconfigurable slots to which the callback can be mapped. Obviously, different and improved ReconROS executor designs are conceivable. For example, for callbacks that can be run in software and hardware, the executor could decide which mode to choose at runtime. Moreover, involved resource management problems arise if the reconfigurable slots are of different sizes and hardware callbacks are available for different reconfigurable slots. Such improved executor scenarios are left to future work.

4.2.2 *Replacement*

In the standard implementation of the ReconROS executor, the assignment of hardware callbacks to reconfigurable slots is relatively simple. Each reconfigurable slot has a worker thread attached, trying to catch the following incoming callback invocation when the reconfigurable slot is idling. Due to this assignment strategy, a reconfigurable slot may be reconfigured with a hardware callback, although this callback would still have been available from a previous execution in another reconfigurable slot.

In order to improve this situation, we propose and have implemented an optimized assignment/replacement (called placer) module in the ReconROS executor.

In order to avoid unnecessary reconfiguration processes, the executor considers all reconfigurable slots currently idle upon receiving a new callback invocation. During runtime, the placer takes care of a map of status information about all reconfigurable slots. This information includes the callback ID for each slot placed last and the current state (running or waiting).

The placer provides options for interaction with this map: first, a hardware worker thread can ask for a new callback to execute. In this case, the placer checks the map only for the requested reconfigurable slot. If the request for a new callback fails, the hardware worker thread can ask the placer to update the map. In this case, the placer invokes the callback list and asks for a new ready-to-execute callback following the standard scheduling procedure (cf. Section 4.2.1). If successful, the placer inserts the received callback into the map according to the following procedure:

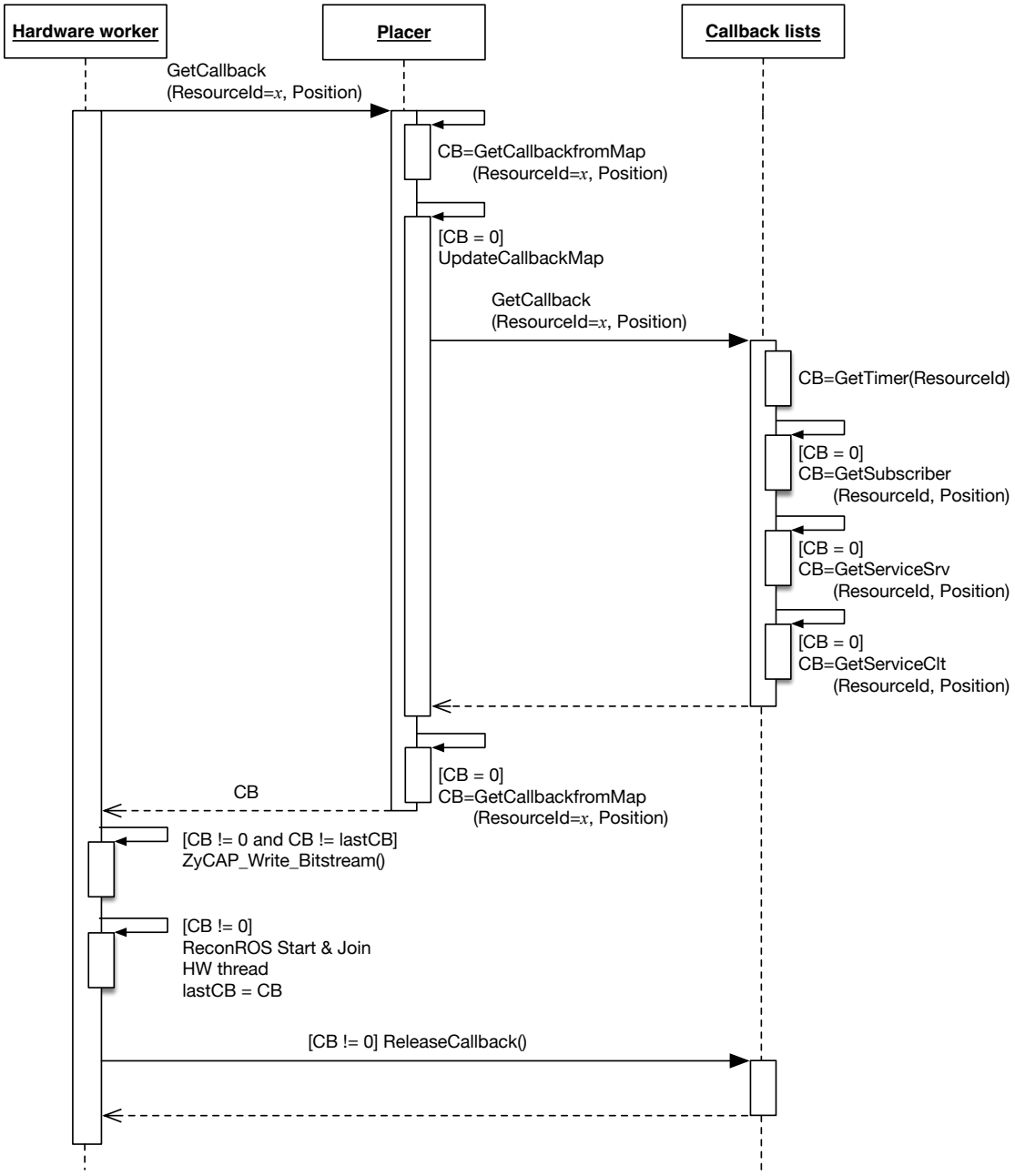

Figure 4.4: Sequence diagram for a hardware worker thread including an optimized replacement strategy.

1. The module first checks whether the configuration for the new callback already exists in one of the available reconfigurable slots. If so, the hardware callback is assigned to this slot.

2. If the callback is not already configured, the module checks whether one of the reconfigurable slots is empty. If such a slot exists, the hardware callback is assigned.

3. If all reconfigurable slots are configured with hardware callbacks, one has to be selected for replacement. For this, we have implemented two

techniques: least recently used (LRU), where we replace the hardware callback that has not been used for the longest time, and least frequently used (LFU), where we replace the hardware callback that has been used least often. These options can be enabled in the executor.

After finishing the map update, the current hardware worker thread tries to receive a new callback for its reconfigurable slot by asking the placer again. Once the executor assigns a callback to the current reconfigurable slot, the worker thread checks whether a reconfiguration is mandatory. Then, the reconfiguration process is started, and the callback is started upon completion. Figure 4.4 shows the resulting sequence diagram.

4.3 EXPLOITATION OF PARALLELISM

ReconROS helps improve the performance of ROS 2 applications by exploiting parallelism on two levels. First, ROS 2 nodes mapped to hardware threads or hardware callbacks, respectively, achieve for many functions higher performance than their software counterparts due to massive low-level parallelism at the bit-level offered by FPGA technology. Additionally, FPGA implementations can benefit from customized operators, number formats, data paths, and memory architectures.

Second, running several hardware threads/callbacks in parallel exploits task-level parallelism and parallel to the software threads/callbacks on the CPU cores.

In addition, we have extended ReconROS to allow us to use thread-level parallelism for exploiting data parallelism for ROS 2 nodes using publish/subscribe communication. To this end, we start several identical ReconROS software and/or hardware threads (or callbacks) for a ROS 2 node. Since all these threads belong to the same ROS 2 node, incoming messages will be sequentially assigned to the threads. Our extensions in the ReconROS primitives for publish/subscribe communication ensure that also the publish phases occur in the correct order. During an interaction with the communication primitive, the calling thread provides its thread identifier between $0...nthreads - 1$. This thread identifier is then compared with a counter, and publishing threads are blocked until the counter value matches the thread identifier. Then, the counter is incremented.

Figure 4.5 displays an exemplary schedule, where a data-parallel ROS 2 node is implemented by three software threads and one hardware thread, all functionally identical. Messages are received in the *take* phase. In the example, hardware thread 3 has a shorter *computation* phase than the software threads and has to be blocked before going into its *publish* phase since software thread two must be allowed to publish its results first.

Through our extension to ReconROS, mapping a ROS 2 node to several data-parallel tasks does not require any modifications to the thread's code. This form of parallelism is of interest, especially for stateless ROS 2 nodes, i.e., without data dependencies between iterations. Examples of such functions are image filtering or feature-detection tasks. On the other hand, ROS 2

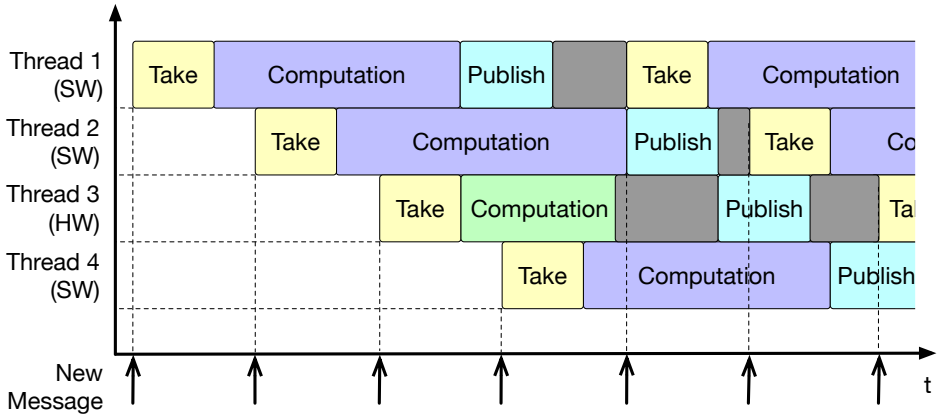

Figure 4.5: Exploiting data parallelism for a ROS 2 node in ReconROS. Adapted from [52].

nodes that need to maintain state between iterations, such as navigation functions that must keep track of state across the iterative computations, are not well-suited for this computing pattern.

4.4 EXPERIMENTAL EVALUATION

This section reports experiments demonstrating and evaluating task mapping using ReconROS and the ReconROS executor.

First, we start with an experiment showing several exemplary ROS 2 nodes mapped statically to hardware. After that, the evaluation of the dynamic task mapping approach elaborates on a comparison of the proposed ReconROS executor with the standard ROS 2 multi-threaded executor in an example comprising five callbacks. Finally, we detail an experiment evaluating the different replacement strategies of the ReconROS executor for dynamically mapped hardware tasks.

4.4.1 *Static Task Mapping*

In this section, the described example application with different characteristics in terms of memory accesses and computation aims to demonstrate the potential of hardware acceleration using ReconROS.

All hardware threads and callbacks have been implemented in C++ and synthesized with the high-level synthesis tools of Xilinx Vitis 2021.2. The computing platform is a ZCU104 evaluation board comprising an UltraScale+ MPSoC running Ubuntu Linux 20.04, ReconROS, and ROS 2 galactic. The ReconROS infrastructure and the HMNs are clocked at 150 MHz. Moreover, a desktop PC with an Intel Core i5-8000 CPU running Ubuntu Linux 20.04 and ROS 2 galactic is connected to the FPGA evaluation board via Gigabit Ethernet.

For the demonstration of ReconROS, we have implemented the following application comprising four nodes running on the platform and one node

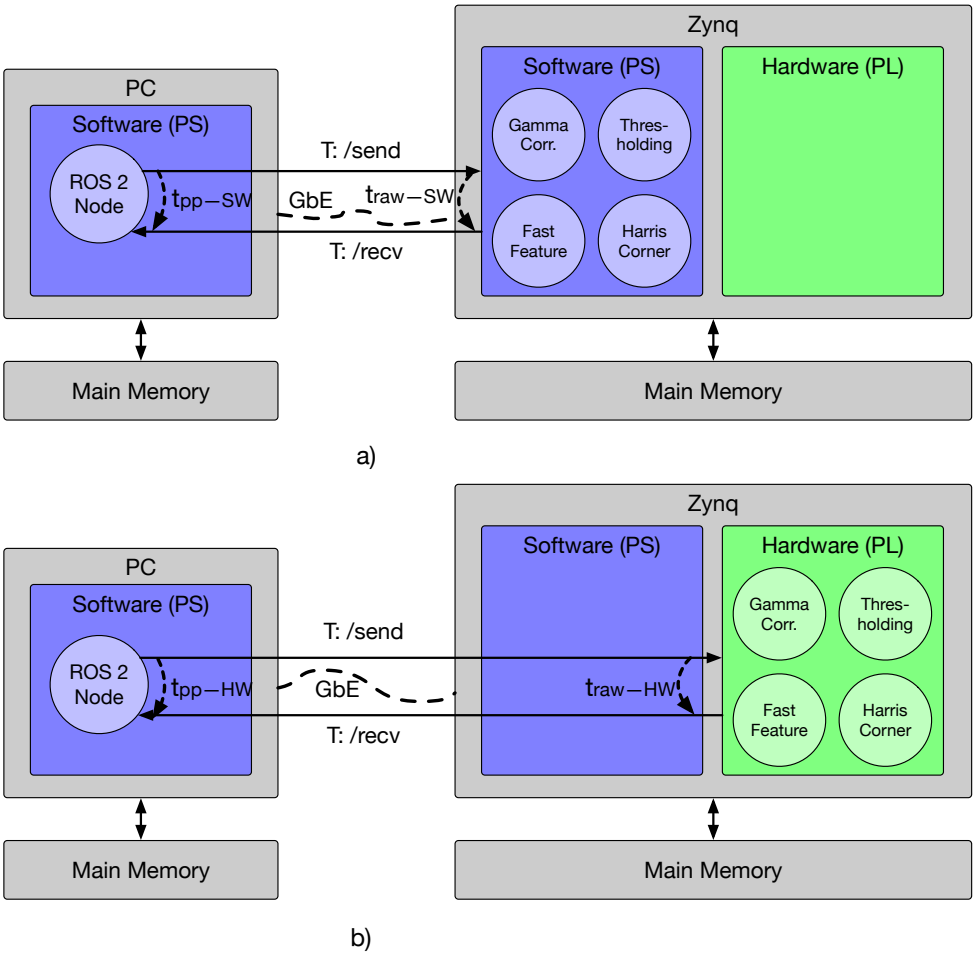

Figure 4.6: ReconROS ping-pong application including five example ROS 2 nodes either in hardware or software. Adapted from [53].

running on the desktop PC shown in Figure 4.6. All hardware implementations rely on the open-source Xilinx Vitis Image processing library [122], whereas all four software implementations rely on the OpenCV framework [81].

GAMMA CORRECTION: This node applies a gamma correction on an RGB color image with 640 × 480 pixels and outputs an RGB color image with the same dimensions. The gamma correction is a non-linear operation aimed at adjusting an image's brightness values to the eye's non-linear perception [33]. The implementation, both for hardware and software, follows an approach based on look-up tables that are pre-computed for faster computation during runtime. The look-up table comprises 256 values for each channel, which results in 768 values in sum.

COLOR THRESHOLDING: This node applies a color space transformation on an input image followed by a thresholding operation. The color transformation maps the incoming RGB image with 640 × 480 pixels into

the HSV color space. The HSV color space comprises one number each for hue, saturation, and value [33]. After conversation, a thresholding operation selects all pixels in a range between two constants by setting pixels from inside the range to 255 on all three channels. In contrast, pixels outside the range are set to 0 on all three channels.

FAST FEATURE DETECTION: The FAST feature detection implemented in this node example is a more advanced image processing operation. The FAST algorithm was presented in 2006 [101] aiming to increase the computational efficiency of feature detection. Due to its computational efficiency, it is widely used, e.g., in the famous ORB-SLAM2 [69] implementation. FAST detects features of an image by inspecting pixels on a cycle with a fixed radius around a current pixel of interest. This node uses FAST to compute features on a 640×480 grayscale image. The resulting detected feature pixels are inserted into an empty (black) 640×480 image as white pixels.

HARRIS CORNER DETECTION: In terms of computation, the most expensive demonstrator node for this evaluation is the Harris corner detection algorithm. The Harris corner detection algorithm is another feature detection algorithm first presented in 1988 [35]. The first step of the algorithm is the computation of the gradients of the grayscale input image in x and y-direction using a Sobel filter [33]. The resulting gradients G_x and G_y are cross-multiplied and squared to G_x^2, G_y^2, and G_{xy} for each input pixel. The second step of the algorithm applies an average filter on G_x^2, G_y^2, and G_{xy} to $\langle G_x^2 \rangle$, $\langle G_y^2 \rangle$, and $\langle G_{xy} \rangle$ [34]. In the final step, the resulting gradients are used to compute a score for each pixel by $R = (\langle G_x^2 \rangle \cdot \langle G_y^2 \rangle - \langle G_{xy} \rangle^2) - k \cdot (\langle G_x^2 \rangle + \langle G_y^2 \rangle)^2$. A pixel is now detected as a corner if $R > k_{threshold}$. For the implementation in this experiment, k is fixed to 0.04 and $k_{threshold}$ to 566. Similar to the FAST feature detection, the Harris corner detection implementation outputs an empty 640×480 grayscale image with detected features inserted.

ReconROS node	CLB	DSP	BRAM
Gamma correction	2116 (0.92%)	0 (0.00%)	6.0 (1.92%)
Color Thresholding	2637 (1.14%)	3 (0.17%)	8 (2.56%)
FAST feature detection	5217 (2.26%)	0 (0.00%)	6.5 (2.08%)
Harris corner detection	7235 (3.14)	11 (0.17%)	24.0 (7.69%)

Table 4.1: Resource usage and utilization (in % of the Xilinx XCZU7EV-2FFVC1156) for the implemented ReconROS nodes. Resource figures are reported for CLBs, DSPs, and BRAMs.

Table 4.1 displays resource usage and FPGA utilization for the four nodes, and Table 4.2 the raw runtimes for the Zynq-bound ROS 2 nodes, which are either mapped to the ARM core (t_{raw-SW}) or to reconfigurable logic (t_{raw-HW}) and the resulting raw speedup S_{raw}. Table 4.3 shows the runtimes

between nodes running on PC and Zynq measured in the ping-pong fashion shown in Figure 4.6.

Besides the gamma correction node, all three other nodes achieve a speedup larger than one. In general, the speedup achieved increases with the complexity of the node. On the one hand, in the case of gamma correction, where the channel value of a specific pixel only has to be replaced by another value from the LUT, hardware acceleration leads to a speedup smaller than one since the memory transfer between reconfigurable logic and processing system predominates. On the other hand, the implementation of Harris corner detection shows a speedup of more than 17×, which can be achieved by processing the data in a pipeline.

Another observation from the experiments is the fact that the raw computation time of the four examples depends highly on the input and output data size since we have measured more or less only two different execution times (7.5 *ms* for 900 kB in/out and 5.4 *ms* for 300 kB in/out). Again, this is due to the pipeline structure of the hardware accelerators.

The communication overheads of ROS 2, especially the network transfer between the FPGA evaluation board and the desktop PC, blur the speedups for all four nodes.

ReconROS node	Message Size In/Out	t_{raw-SW} [ms]	t_{raw-HW} [ms]	S_{raw}
Gamma correction	900/900 KiB	3.91	7.54	0.52×
Color Thresholding	900/900 KiB	7.95	7.52	1.06×
FAST feature detection	300/300 KiB	13.87	5.40	2.56×
Harris corner detection	300/300 KiB	95.35	5.40	17.64×

Table 4.2: Raw runtimes of software and hardware ROS 2 nodes and corresponding speedups.

ReconROS node	Message Size In/Out	t_{pp-SW} [ms]	t_{pp-HW} [ms]	S_{pp}
Gamma correction	900/900 KiB	4.00	3.64	0.90×
Color Thresholding	900/900 KiB	40.36	31.74	1.27×
FAST feature detection	300/300 KiB	22.37	14.06	1.59×
Harris corner detection	300/300 KiB	112.89	13.94	8.09×

Table 4.3: Roundtrip runtimes of software and hardware ROS 2 nodes and corresponding speedups.

To summarize the set of experiments detailed in this section: We have shown that while there is an overhead for mapping a ROS 2 node to hardware, the impact on an overall ROS 2 application depends on many factors such

as i) the raw speedup of the ROS 2 HMN, ii) the message size, iii) the overall application's topology and involved communication patterns and times, and iv) the ratio between node computation times and communication times. The memory access performance for ROS 2 HMNs is lower than for their software counterparts, which provides optimization potential for future work. Additional speedups can be realized through the parallel execution of hardware and software threads.

Finally, all hardware and software versions of the ReconROS nodes are semantically identical. Creating the different versions requires a change in the ReconROS configuration file before running the functions of the ReconROS development kit only. This flexibility in generating variants of ROS 2 hardware-accelerated nodes is one of the main features of ReconROS.

4.4.2 *Dynamic Task Mapping Example*

So far, we have statically mapped ROS 2 nodes to the reconfigurable hardware in the evaluation. Although this can avoid overheads due to dynamic reconfiguration, it may also result in poorer utilization of resources since these are reserved by the respective ROS node during the entire runtime.

However, ReconROS offers the possibility to map ROS 2 nodes dynamically to the hardware in the form of callbacks through its executor. These callbacks can be mapped to hardware or software. Therefore, as a first part of evaluating the ReconROS executor, we have measured the runtimes for five ROS 2 nodes, more precisely, their callbacks. All callback functions have been coded in C/C++ and synthesized with Xilinx Vivado HLS to a Zynq Z7100 on a MiniITX FPGA board. The hardware callbacks run at 120 MHz, the ReconROS infrastructure at 100 MHz, and the ARM Cortex-A9 at 666 MHz.

SOBEL FILTER: This callback implements a Sobel image filter [33] operating on three channels (RGB) of dimension 640×480. The filter applies two filter kernels on each image channel and calculates the dot product's absolute value as an approximation for the geometric mean. The ROS 2 input and output messages are of the type Image from the ROS 2 sensor message package.

NUMBER SORTING: This callback provides a ROS 2 service, which sorts an array of 32-bit unsigned integers based on the odd-even transposition sort algorithm [42]. The algorithm is based on a comparator network that employs n stages with n comparisons each to sort n numbers. The ROS 2 node on the PC generates random numbers and publishes messages comprising 2048 numbers as an array.

MNIST CLASSIFIER: This callback classifies handwritten digits from the MNIST dataset using a neural network. The classifier is implemented using ROS 2 publish/subscribe communication. It subscribes for input images of

size 28×28 and publishes the estimated digit as an unsigned integer. The classifier consists of three convolution, pooling, and fully connected layers. The achieved accuracy is about 97%.

INVERSE KINEMATICS: This callback computes control signals for driving a servo motor that sets a joint angle based on the desired position and orientation of a robotic manipulation platform. The application is part of a more extensive mechatronic system for controlling the movements of a Stewart platform [107] with six degrees of freedom. The computation involves coordinate transformations and an iterative implementation of the arctan() function. The ROS 2 input message is an unsigned 32-bit integer packed with two fixed-point numbers in Q8.6 format that represent the desired rotation angles of the platform around the x-axis and the y-axis. The ROS 2 output messages is also a 32-bit unsigned integer containing a 10-bit unsigned integer, the pulse width coded control signal for the motor.

HASH CALCULATION: The hash calculation callback is implemented to demonstrate a callback triggered by a periodic timer. The algorithm reads a 1920×1080 image with 24-bit color depth from main memory and calculates its SHA256 hash value at each run. Afterward, the hash value is published as an unsigned integer array with eight elements to a ROS 2 topic.

ROS 2 callback	$t_{exec-HW}$ [ms]	$t_{exec-SW}$ [ms]	Speedup
Sobel filter	16.50	42.00	2.5×
Number sorting	0.85	41.00	48.2×
MNIST classifier	11.90	16.50	1.4×
Inverse kinematics	0.35	1.50	4.3×
Hash calculation	81.00	94.00	1.2×

Table 4.4: Execution times for five ROS 2 callbacks in hardware ($t_{exec-HW}$) and software ($t_{exec-SW}$), and the resulting speedup. Taken from [51].

Table 4.4 shows the execution times for the callbacks, comprising the execution time for the callback function in software and the time between starting and completing the callback in hardware, respectively. The reconfiguration times are not included in this measurement. Speedups are achieved for all five callbacks, with the hash calculation resulting in the lowest and the number sorting resulting in the highest speedup. The wide range of speedups achieved is due to different task characteristics, e.g., data dependencies for computations and the amount of data for input and output, and varying degrees of optimization for the five designs, e.g., for memory transmission and handling and parallelism in the execution pipeline. While all implementations leave room for further improvements, the results do motivate the use of hardware callbacks. How the speedup achieved for a

single callback propagates to the overall robotics system, e.g., a complex robotics application for an autonomous robot, is highly application-specific. Therefore, we consider isolated measurements of single callbacks in this thesis.

In the following experiment, we compare the performance of a ROS 2 application based on the three callbacks in software using the standard ROS 2 executor with two different hardware/software mappings using our ReconROS executor.

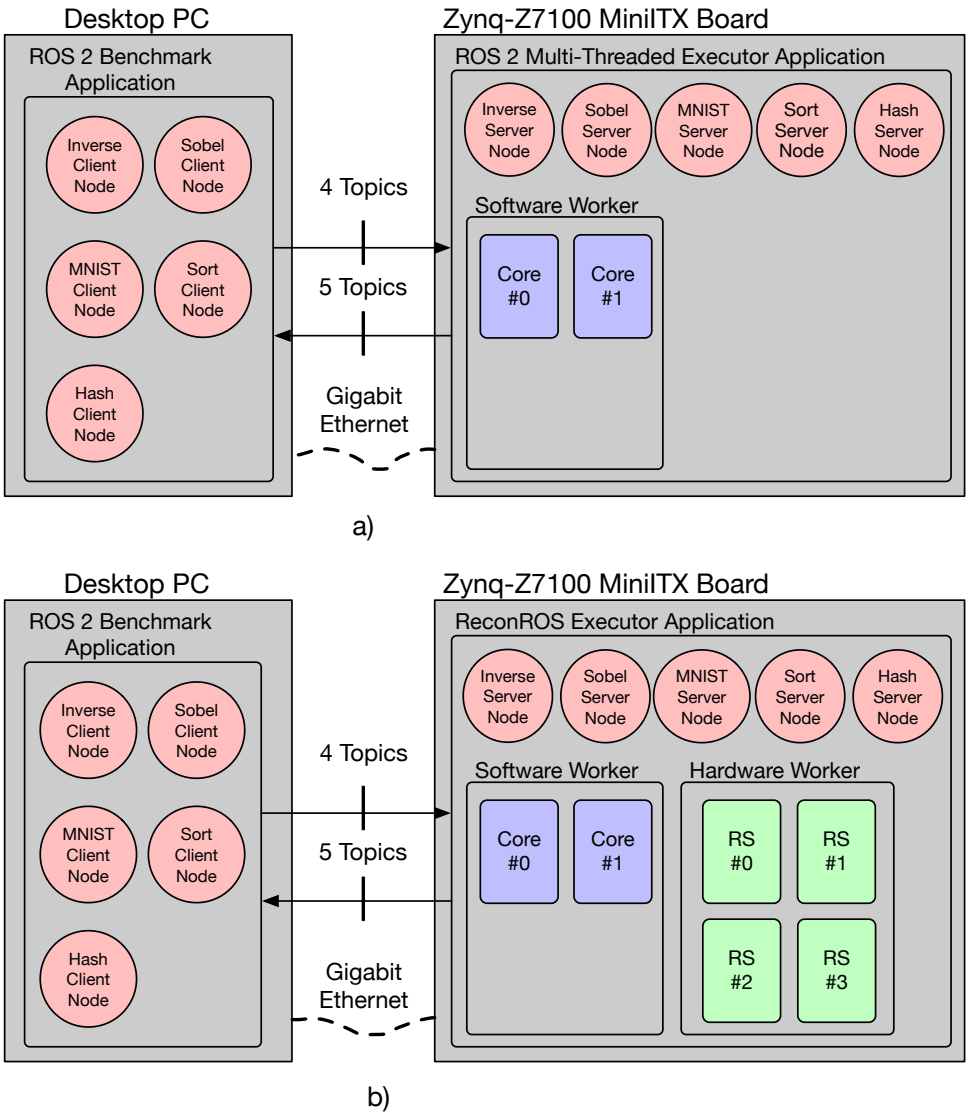

Figure 4.7: Experimental setup for a ROS 2 application with a standard ROS 2 executor (a), and our ReconROS executor (b). Taken from [51]

The ROS 2 setup is illustrated in Figures 4.7(a) and (b). On the desktop PC, there are five ROS 2 client nodes programmed in C++ and compiled against the ROS 2 rclcpp library. These client nodes, i.e., the *Sort client*, *Inverse client*, *Sobel client*, and *MNIST client* nodes, comprise a publisher and a subscriber. Starting with an initial published message, the client's subscriber waits for a

response from the corresponding server node on the FPGA. After receiving a new message for the topic, the clients immediately publish a new message for their server counterpart. The resulting roundtrip times are logged during the experiment. The *Hash client* node forms a particular case. Since the hash server node only publishes messages, the client on the desktop PC receives the messages and reports the times between consecutive messages. During this experiment, the ReconROS executor runs without a dedicated placer in its most simple form (cf. Section 4.2.2).

Figure 4.7(a) sketches an all-software mapping, where a multi-threaded standard ROS 2 executor with two software worker threads on the FPGA dispatches the callbacks from the server nodes to two processor cores. Figure 4.7(b) displays the setup with the ReconROS executor and an additional four hardware worker threads that dispatch callbacks to four reconfigurable slots. We have evaluated two mappings under the ReconROS executor, a mixed software/hardware mapping where the four callbacks with the highest speedups according to Section 4.4, i.e., *Number sorting*, *Inverse kinematics*, *Sobel filter*, and *MNIST classifier*, are executed in hardware and the *Hash calculation* callback is executed in software. The all-hardware mapping finally runs all callbacks in hardware.

A detailed listing of the average roundtrip time and the resulting average speedups for both ReconROS setups in Table 4.5 shows significant speedups for the *Inverse kinematics* and *Number sorting* nodes and roughly equal roundtrip times for the *Sobel filter* and *Mnist classifier* nodes.

	ROS 2 standard executor all-SW	ReconROS SW/HW	ReconROS all-HW
Inverse	27.4 ms (1.00×)	3.97 ms (6.90×)	4.41 ms (6.21×)
Mnist	36.67 ms (1.00×)	36.69 ms (1.00×)	37.77 ms (0.97×)
Sobel	184.56 ms (1.00×)	157.86 ms (1.17×)	179.70 ms (1.03×)
Sort	67.1 ms (1.00×)	3.20 ms (20.97×)	3.69 ms (18.18×)
Hash	249.64 ms	249.87 ms	249.87 ms

Table 4.5: Average execution times and speedups compared to the ROS 2 executor reference implementation (first column).

Figure 4.8 analyzes the resulting roundtrip times for the three mappings. The figure plots the relative frequency over the roundtrip time for each of the five ROS 2 nodes and the three mappings. The dashed lines denote the averages. Going from the all-software over the software/hardware to the all-hardware mapping, the speedups based on the averaged roundtrip times are 6.21 and 6.29 for *Inverse kinematics*, 0.97 and 1.00 for the *MNIST classifier*, 1.03 and 1.15 for the *Sobel filter*, 18.18 and 20.97 for *Number sorting*, and 1.00 for the *Hash calculation*. Overall, we make the following observations:

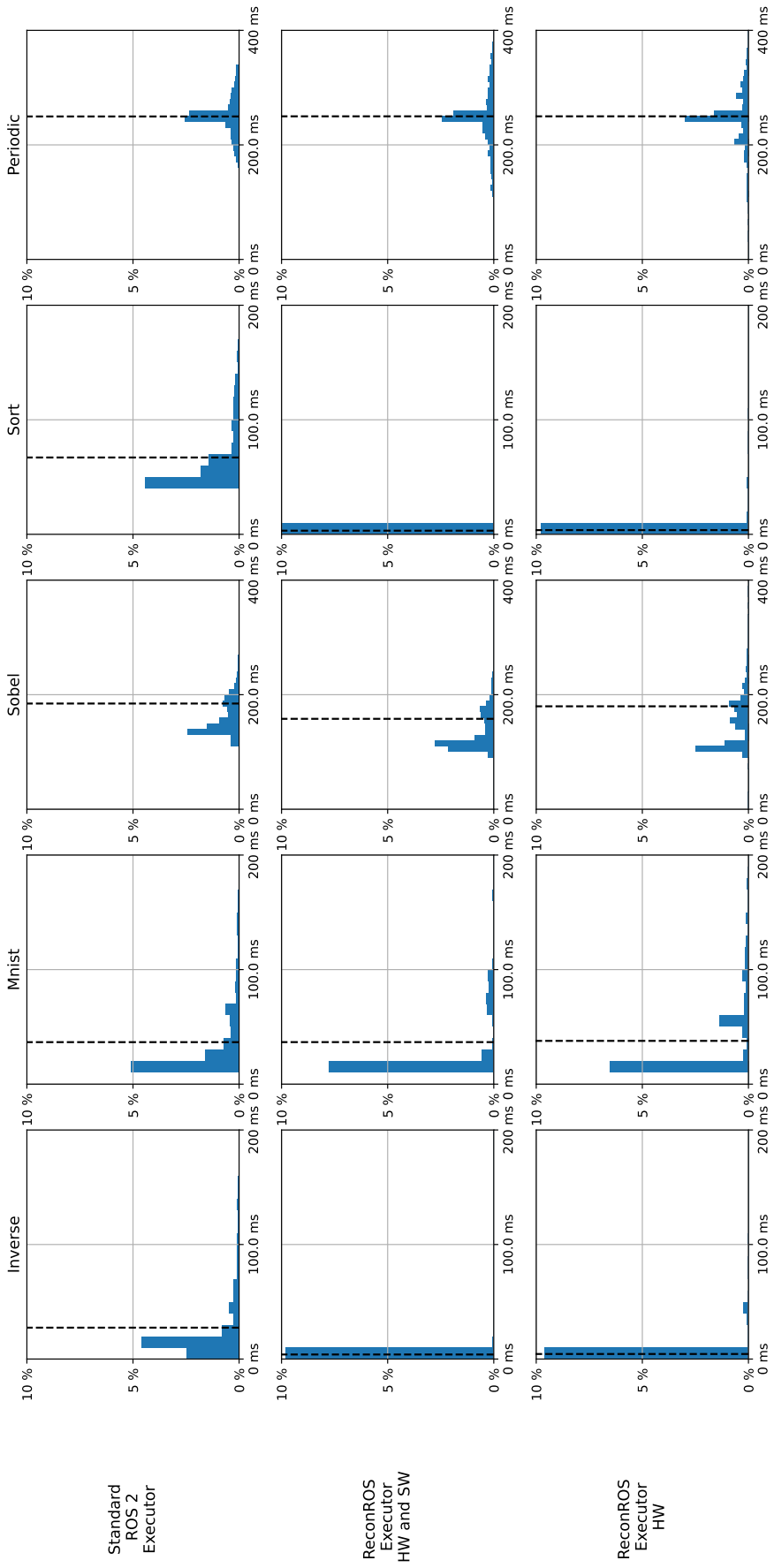

Figure 4.8: Relative frequencies of the roundtrip times for the ROS 2 standard executor and two ReconROS executor configurations; the dashed lines show the average roundtrip time for the specific ROS 2 node. Taken from [51].

- The speedups for the individual ROS 2 nodes within the overall application follow the trends for the callbacks measured in isolation, shown in Table 4.4, although generally lower due to the communication between desktop PC and FPGA board, the ROS 2 communication layers, and the executors. For *Number sorting*, *Inverse kinematics*, and to some extent the *Sobel filter*, distinct speedups are realized.
- The hash calculation is triggered with a 250 *ms* period. The distribution of roundtrip times shows entries with less and more than 250 *ms* since the ROS 2 client on the desktop PC measures times between arriving messages from this callback. Some messages are delayed, reducing the time for the following message.

4.4.3 *Hardware Callback Replacement*

Following Section 4.2.2, the ReconROS executor has been extended by a more advanced placement strategy aiming to reduce reconfiguration overheads.

The following evaluation investigates the replacement strategies for hardware callbacks in the ReconOS executor on a slightly different setup than previous experiments (cf. Section 4.4.2). The same set of callbacks comprising a Sobel image filter, an MNIST classifier, an inverse kinematics algorithm, and a number sorting node has been used for the experiment.

All hardware threads and callbacks have been implemented in C++ and synthesized with the high-level synthesis tools of Xilinx Vitis 2021.2. The computing platform is a ZCU104 evaluation board comprising an UltraScale+ MPSoC running Ubuntu Linux 18.04, ReconROS, and ROS 2 dashing. The ReconROS infrastructure is clocked at 100 MHz, and the hardware threads and callbacks are at 120 MHz. Moreover, a desktop PC with an Intel Core i5-8000 CPU running Ubuntu Linux 18.04 and ROS 2 dashing is connected to the FPGA evaluation board via Gigabit Ethernet.

The desktop PC executes a set of client nodes that periodically send messages and request services and thus create message events to be handled by the ReconROS executor on the platform FPGA. For example, the inverse kinematics client sends data every 30 *ms*, the MNIST client every 50 *ms*, and the Sobel client sends images every 100 *ms*. The sort client requests the number sorting every 250 *ms*.

The platform FPGA executes the ReconROS executor with its set of hardware callbacks on two reconfigurable slots using the replacement strategy without placer, LRU and LFU, and measuring the number of replacements. Table 4.6 lists the number of replacements per callback execution. The measurements show that our previous implementation led to replacement in some 60% of all callback executions. LRU and LFU perform much better, with a slight advantage for LFU with 42% replacements. This emphasized improvement even for only two reconfigurable slots, underlining the importance of the replacement strategy and motivating research for more advanced techniques such as speculative reconfiguration.

Replacement strategy	Replacements per callback execution
Without Placer	0.61
LRU	0.44
LFU	0.42

Table 4.6: Evaluation of different hardware callback replacement strategies. Taken from [52].

Overall, the required number of reconfigurations for the presented demonstrator application is decreased by 30 percent. This shows not only the benefits of using improved callback displacement algorithms and verifies the correct functionality of the implementation, but it also motivates the research for even more advanced techniques, e.g., by leveraging speculative reconfiguration.

4.5 CHAPTER CONCLUSION

This chapter deals with the task mapping and exploitation of parallelism in ReconROS. ReconROS provides flexibility by supporting static task mapping and dynamic task mapping. For the dynamic mapping of tasks, the ReconROS executor takes care of the scheduling and placement of tasks during runtime without the need for custom application-specific solutions for scheduling and placement. Regarding the exploitation of parallelism, ReconROS supports two levels of parallelism: low-level parallelism and task-level parallelism.

The evaluation for the mapping of tasks with ReconROS shows speedups in runtime for most of the statically mapped tasks. It can be observed that computationally expensive tasks benefit more from an acceleration in hardware, as the transfer to the reconfigurable hardware compensates for the speedups during the calculation. This fact motivates the following Chapter 5 aiming to reduce communication overheads for HMNs. In the evaluation and validation of the dynamic mapping, a speedup can be shown compared to the standard ROS 2 executor. Finally, the comparison of different placement strategies shows improvements in average reconfigurations needed for the LRU and LFU placement.

5

COMMUNICATION OPTIMIZATION IN RECONROS

In the previous two chapters, we could already present speedups due to hardware acceleration on the node level. In several cases, ROS 2 nodes mapped to hardware benefit from high parallelism in the reconfigurable hardware. However, the evaluation shows that speedups can suffer from data transfers between the hardware and software domains. Since, until now, hardware acceleration has been applied node-wise, input and output data have to be transferred for each HMN separately, even if the output data of a HMN is needed by a subsequent HMN.

In order to unlock more potential for hardware acceleration, this chapter deals with the mapping of ROS 2 computation graphs or subgraphs to the reconfigurable fabric aiming to reduce communication overheads between nodes. The mapping of complete subgraphs into the hardware domain allows to leverage intra-FPGA communication, promising higher bandwidths and lower latencies compared to communication between the hardware and software domains. This holds because all HMNs share one hardware/software communication interface. As already mentioned during the related work survey in the background chapter (cf. Chapter 2.3), ReconROS was not the first framework allowing intra FPGA communication. However, as described in the following chapter, our approach is more comprehensive than previous work. In contrast to related work, our approach allows, e.g., an arbitrary number of publishers and subscribers per topic, the automatic generation of the hardware infrastructure by our toolflow, and the usage of the standard semantic for communication of ROS 2.

Before introducing intra-FPGA communication, Section 5.1 introduces extensions on ReconROS to support a zero-copy communication scheme offered by standard ROS 2 communication. Then, we first present fpgaDDS in Section 5.2, a static intra-FPGA communication middleware aiming to reduce communication overheads between HMNs. In the standard version, fpgaDDS was limited to ROS 2 topics connected to nodes mapped to hardware only. Since, for this case, the insertion of single SMN could result in fallback to standard ROS 2 communication with a significant performance loss, we have introduced so-called gateways in Section 5.3. Figure 5.5 provides an example of such a fallback situation. Gateways synchronize data between a standard ROS 2 topic and a hardware-mapped topic and allow the use of hardware-mapped topics for a broader range of applications. In order to map computation graphs to hardware, Section 5.4 presents a methodology

leveraging fpgaDDS and gateways. Section 5.5 evaluates fpgaDDS in a multi-subscriber scenario. For a further evaluation based on real-world examples, the reader is referred to the ReconROS case studies in Section 6.2 and Section 6.3.

This chapter mainly follows the conference publication presented at the International Conference on Intelligent Robots and Systems (IROS) about fpgaDDS [48] and communication optimization in general presented at the International Conference on Robotics and Artificial Intelligence (ICRAI) [49]. The content of Section 5.1 follows the publication about AutonomROS [47] (cf. Section 6.3).

5.1 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). Due to a communication abstraction layer, the robotic developer can select between various available DDS implementations with different properties.

Several DDS implementations, e.g., fastRTPS [137], rely on standard socket communication for their default 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 transfers data to the same or other processes. However, this flexibility is paid for with lowered performance since the loopback mechanism results in overheads due to several data copy operations involved.

In order to mitigate such copy overheads, the Iceoryx [138] 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 [26] middleware that allows for simultaneously using socket-based and shared-memory-based communication. When selecting shared memory for communication, the topic 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.

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 sub-

scribing 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.

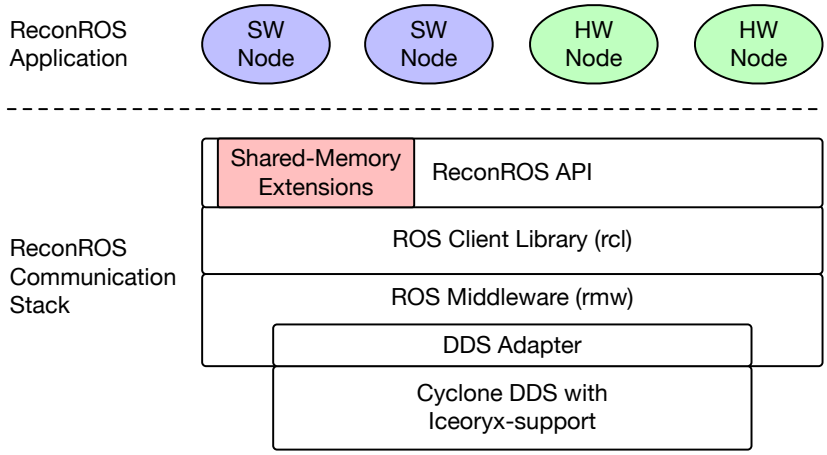

Figure 5.1: Extensions of the ReconROS API in the ReconROS communication stack comprising operations for zero-copy data transfers between nodes (red). Adapted from [47].

Figure 5.1 shows the resulting extensions in the ReconROS API. Overall, we have 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.
- `ROS_SUBSCRIBE_RETURN_LOANED` returns the read message to Iceoryx to re-use the memory chunk.

5.2 INTRA-FPGA COMMUNICATION ARCHITECTURE

The ReconROS framework uses ROS 2 standard communication middlewares for data transport between nodes, including HMNs. Although this excels in flexibility, as HMNs can communicate with arbitrary SMNs and HMNs, the overall application performance may be suboptimal due to the HMNs' mechanism of calling ROS functions via their delegate threads and due to competing memory accesses of all HMNs.

For this reason, we have introduced fpgaDDS, a novel and lean intra-FPGA DDS for ReconROS applications. fpgaDDS maps ROS communication between HMNs to the reconfigurable fabric and thus avoids many memory accesses while maintaining the ROS programming model for standard ROS 2 publish-subscribe communication, i.e., no changes to the ROS nodes are required. Figure 5.2 sketches how fpgaDDS and the corresponding ReconROS DDS adapter integrate into the ROS 2 communication stack.

The new intra-FPGA DDS has two benefits: First, mapping the data transport between communicating HMNs to the reconfigurable fabric saves many memory accesses and thus improves application performance. Second, moving nodes and communication from software to parallel executing hardware further reduces application execution time and jitter, which helps achieve predictable real-time behavior under ROS 2.

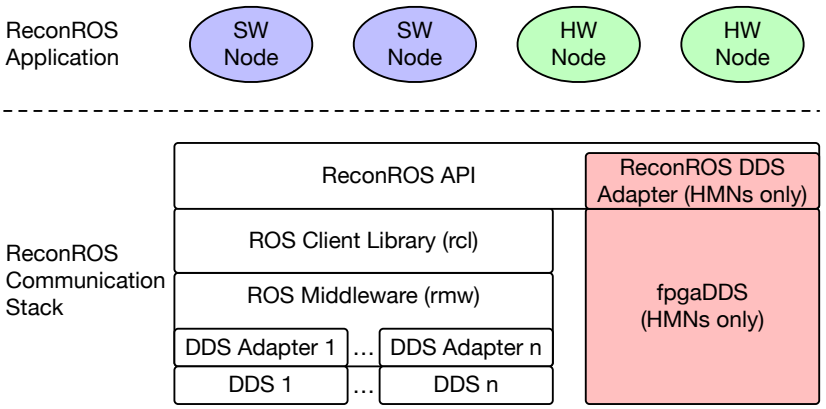

Figure 5.2: Extensions of the ReconROS communication stack include fpgaDDS and the corresponding ReconROS DDS adapter. The zero-copy extensions as part of the ReconROS API (cf. Section 5.1) are not emphasized in this figure. Adapted from [48].

The publish-subscribe communication principle of ROS 2 inspires the structural design of the fpgaDDS communication architecture. The fpgaDDS-extended ReconROS build flow generates an application-specific static AXI-streaming (AXIS) network for each ROS topic separately. Such AXIS-based hardware-mapped topics (HMTs) are lean, resulting in relatively simple communication protocols executed during runtime and high-performance implementations due to the total available bandwidth per HMT.

Figure 5.3 shows an fpgaDDS example comprising six ReconROS HMNs connected to two HMTs, A and B, by AXIS. In the figure, a directed connection from a HMN to a HMT represents a publisher and a connection from a HMT to a HMN is a subscription.

The internal design of a HMT depends on the required number of publishers and subscribers. In the simplest case, where the HMT receives data from one publisher and provides that data to one subscriber, the HMT includes a simple AXIS connection from input to output. If there are more publishers for an HMT, the inputs are gathered by an AXI interconnect IP block that arbitrates the input messages. The AXI interconnect is configured to implement an arbitration based on complete messages, ensuring consistency in the sense that complete messages are always forwarded. If there are more HMT subscribers, an AXI Broadcast IP block broadcasts the message to all subscribers. AXI interconnect and broadcast IP blocks can be concatenated to realize HMTs with arbitrary numbers of publishers and subscribers.

Figure 5.3 also indicates a FIFO buffer for the subscription of topic A by HMN 4. Such FIFO buffers are optional components for subscribers of HMTs.

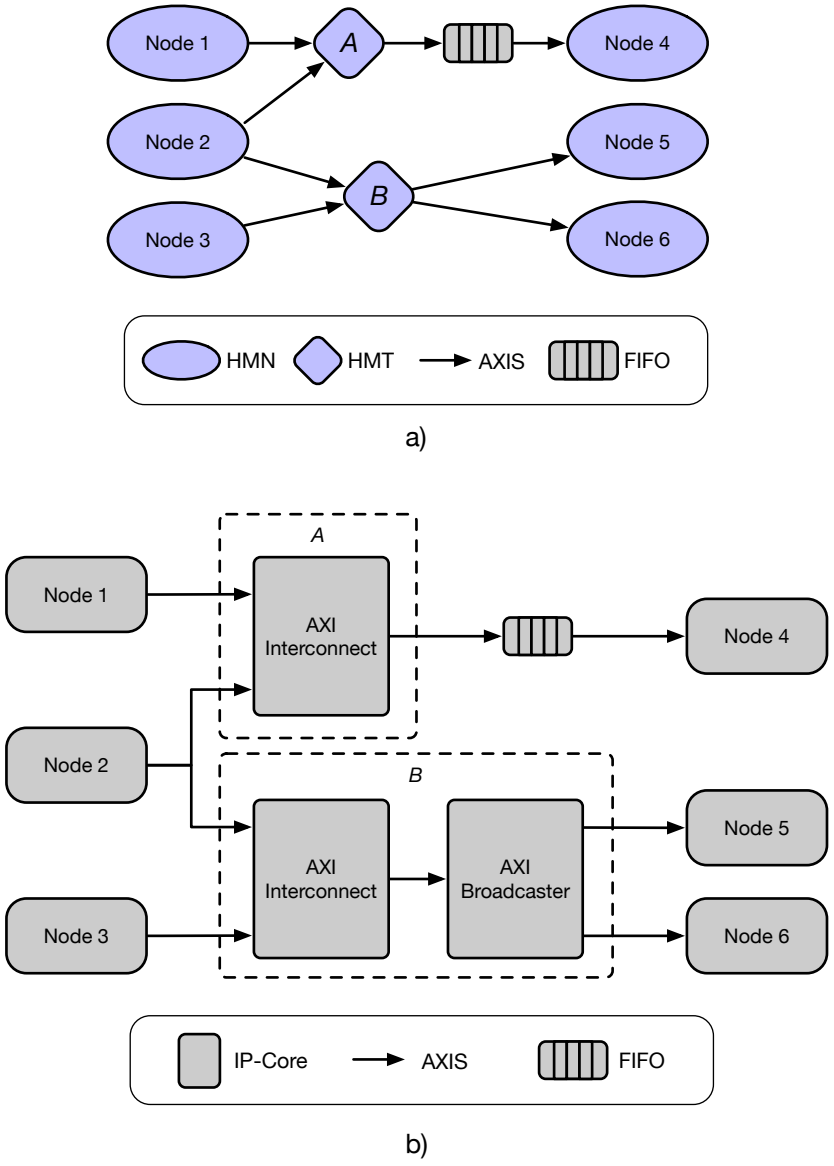

Figure 5.3: Schematic example for a computation graph with two hardware-mapped topics A and B (a) and the resulting instance of the communication architecture (b). Adapted from [48].

By adding FIFO buffers, the data processing of the subscriber nodes can be decoupled from the topic. This behavior can be helpful for asynchronous communication, for example, where the subscriber reads the data from the topic at a different time.

In ROS 2, DDS layers allow developers to specify various Quality-of-Service (QoS) parameters [96]. fpgaDDS, with its AXIS streaming architecture, realizes, by default, publish-subscribe communication with the QoS parameters *Keep All* and *Reliable*, where no messages are discarded. The number of stored messages depends on the size of the FIFO, and if the FIFO runs full, transmission blocks. In addition, due to its static DDS structure, fpgaDDS provides infinite *Lifespan Duration* and infinite *Lease Duration*.

The communication architecture for the fpgaDDS is synthesized based on the ReconROS project configuration file during design time. First, all subscribers and publishers with a hardware property attribute are identified and grouped for their topics. Then, for each HMT, the ReconROS build flow extends the HMNs by input (subscription) and output (publishing) ports, inserts the HMT AXIS infrastructure, and connects the ports and the infrastructure accordingly. The overall procedure corresponds to the discovery procedure in standard DDS implementations.

5.2.1 *ReconROS DDS Adapter*

The ReconROS DDS adapter aims to close the gap between the programming model for streaming networks at the fpgaDDS layer and the publish/subscribe communication mechanism in ReconROS. As a result, the programmer can use similar blocking and non-blocking functions for interaction with software-mapped and hardware-mapped topics.

In order to maintain compatibility between standard message structures provided by ROS 2 and the message structures used in our streaming fpgaDDS communication architecture, which is generated by a high-level synthesis tool flow, we need to serialize message objects on the transmitter side and de-serialize them on the receiver side.

Due to the multi-layer architecture of ROS 2 and the support of several different communication middlewares, the ROS 2 framework already provides functionality for ROS message serialization in software, even for complex nested message types. However, these functions can not be directly used for hardware generation since Vivado HLS provides limited support for pointer arithmetic, casting, and recursion.

Therefore, we extract the structure of messages, i.e., their components and data types, from the message definition file and replace all non-primitives with their sub-components. This also allows for resolving multi-level nested messages. The resulting list of primitives and arrays is then transformed into a C macro for message publishing and subscribing. During hardware synthesis, these macros are in-lined and generate hardware to write and read data sequentially to and from AXI streaming interfaces in blocking and non-blocking versions.

5.2.2 *Execution Modes*

Communicating HMNs can be operated in two execution modes with fpgaDDS. The first mode is the same as when using a software-based DDS and operates the receive, compute, and send phases sequentially. Figure 5.4(b) shows an example of a chain of three nodes with publish-subscribe communication.

With fpgaDDS, we can additionally leverage the dataflow option of the high-level synthesis tool to create an implementation where the phases are overlapped as much as possible, constrained only by the data flow.

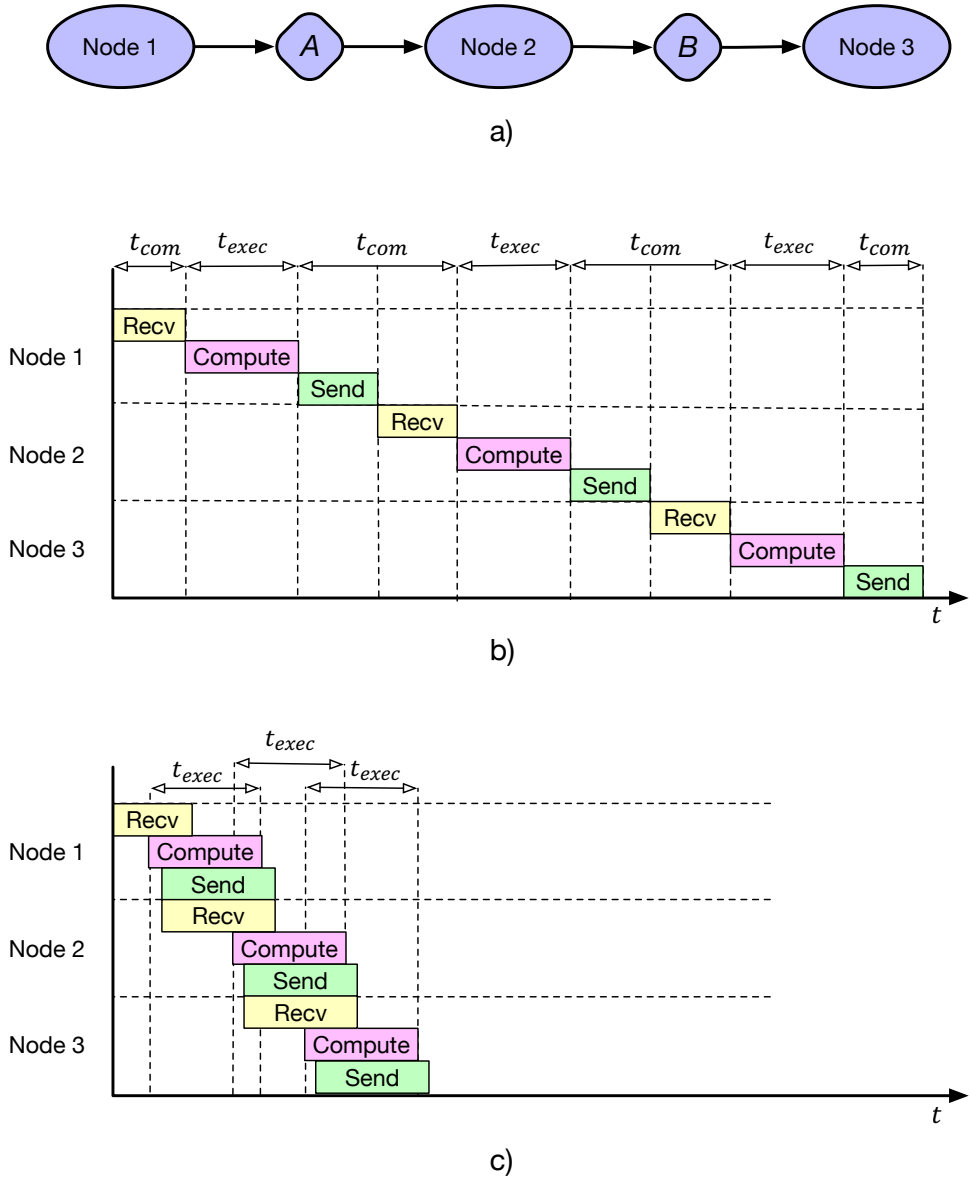

Figure 5.4: Execution modes for the HMN 2 from the computation graph (a) with communication (t_{com}) and execution (t_{exec}) phases: (b) sequential execution and (c) dataflow execution. Adapted from [48].

This operation mode is exemplified in Figure 5.4 (c) and can substantially improve overall execution time. This mode is useful for sets of nodes that operate on data in a streaming manner, which is typical, for example, for many low-level image processing tasks.

5.3 GATEWAYS FOR HARDWARE-MAPPED TOPICS

In the previous section, we have introduced fpgaDDS, which allows for completely mapping communication between HMNs to hardware. In this

section, we will extend the scope of fpgaDDS to ROS 2 topics, which are not exclusively connected to HMNs.

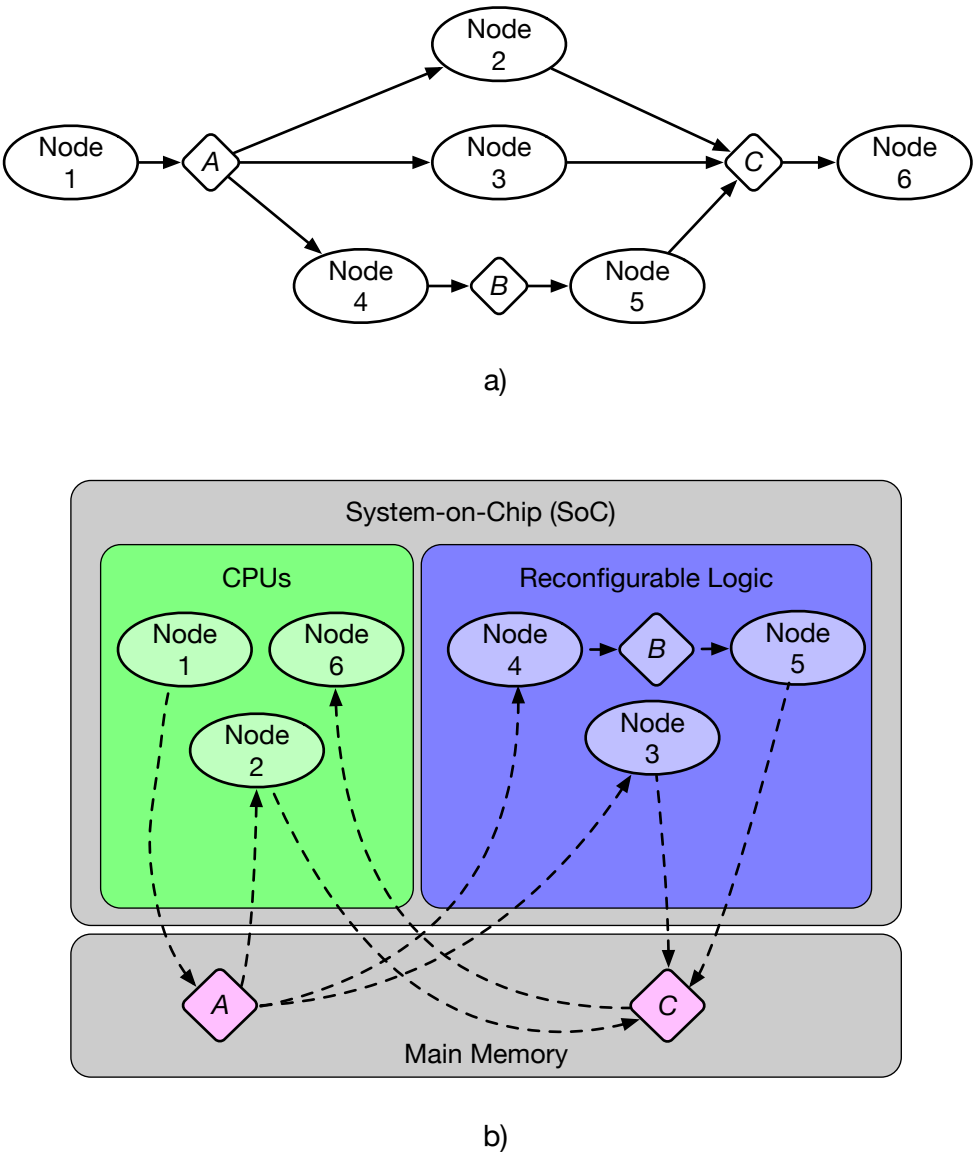

Figure 5.5: Example Application comprising nodes mapped to hardware and software and its communication. Taken from [49].

For a motivation, Figure 5.5 shows an example application as an ROS computation graph (a) and a resulting mapping (b) to a SoC architecture comprising a processing system and a reconfigurable fabric. ROS 2 nodes 1, 2, and 6 are mapped to software, and nodes 3, 4, and 5 are mapped to hardware. The standard mapping of topics is to software, which means the buffers holding messages are realized in the main memory external to the SoC. Efficient DDS implementations such as Iceoryx [138] are available to improve communication performance for shared memory architectures. However, communication to HMNs can challenge the memory interface of the configurable logic. For example, in the mapping of Figure 5.5(b), messages

from topic A, to which nodes 3 and 4 subscribe, are transferred two times to the configurable logic, reducing application performance. fpgaDDS maps topics completely to hardware if all nodes publishing to and subscribing from that topic are also mapped to hardware. Topic B in Figure 5.5(b) exemplifies this case. However, one has to fall back to the standard mapping of topics to software or main memory, respectively, when at least one node publishing to and subscribing from a topic is mapped to software.

As a result, we design a gateway to close the gap between software-mapped topics (SMTs) and HMTs. All hardware-mapped ROS 2 nodes that have to communicate with SMNs share one MEMIF to the main memory and, thus, the memory bandwidth. Gateways aim to reduce the number of data transfers per message in such cases to one.

5.3.1 *Gateway Architecture*

Figure 5.6 sketches the architecture of the gateway. A gateway comprises three main components: a software-mapped topic, a hardware-mapped topic, and the gateway core. Internally, the gateway core is implemented similarly to a node mapped to hardware and establishes publish-subscribe channels to both the software-mapped and hardware-mapped topics. Other hardware-mapped ROS 2 nodes publishing or subscribing to the topic connect to the gateway's HMT, and other software-mapped ROS 2 nodes to its SMT. Since the gateway core synchronizes the SMT and the HMT, only one data transfer to or from the main memory, i.e., between the software and hardware domains, is required per message, significantly reducing the required memory bandwidth.

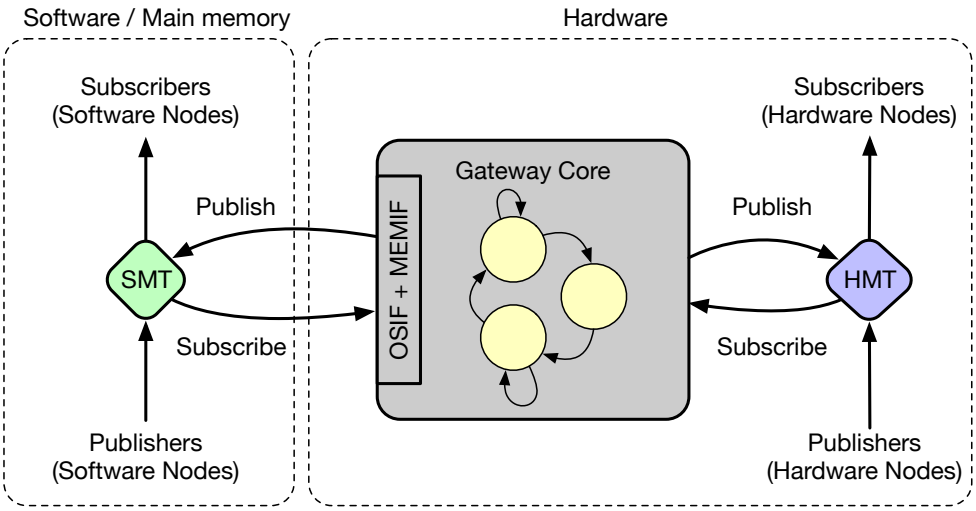

Figure 5.6: The architecture of the gateway. Taken from [49].

The finite state machine in Figure 5.7 presents the runtime behavior of the gateway core: At startup, the gateway core receives the location of the output message for its SMT from its delegate thread and sends a request to it for a new SMT message. Upon that, the delegate thread blocks and waits for

new messages. After receiving a new message, it responds to the gateway core through the OSIF. The gateway core polls both its OSIF for a response from its delegate thread and its HMT for a new message from a HMN. The message is transferred to the HMT in the first case. In the second case, the gateway core transfers the message to the main memory and then cancels the message request to the delegate thread before it publishes the data to the SMT. Since a new message on the SMT could potentially be received between request and cancel, the cancel process may respond with a pointer to a new message. This message is transferred to the HMT before the gateway starts a new request to its delegate. To avoid loops in the gateway core, for example, by messages that are received from an SMT, republished in the HMT, and then are received again by the gateway's subscriber on the HMT side, our gateway core implementation includes message filters in both its subscribers on the SMT and HMT sides. These filters check and discard messages for their publisher IDs if their source and destination identifiers match.

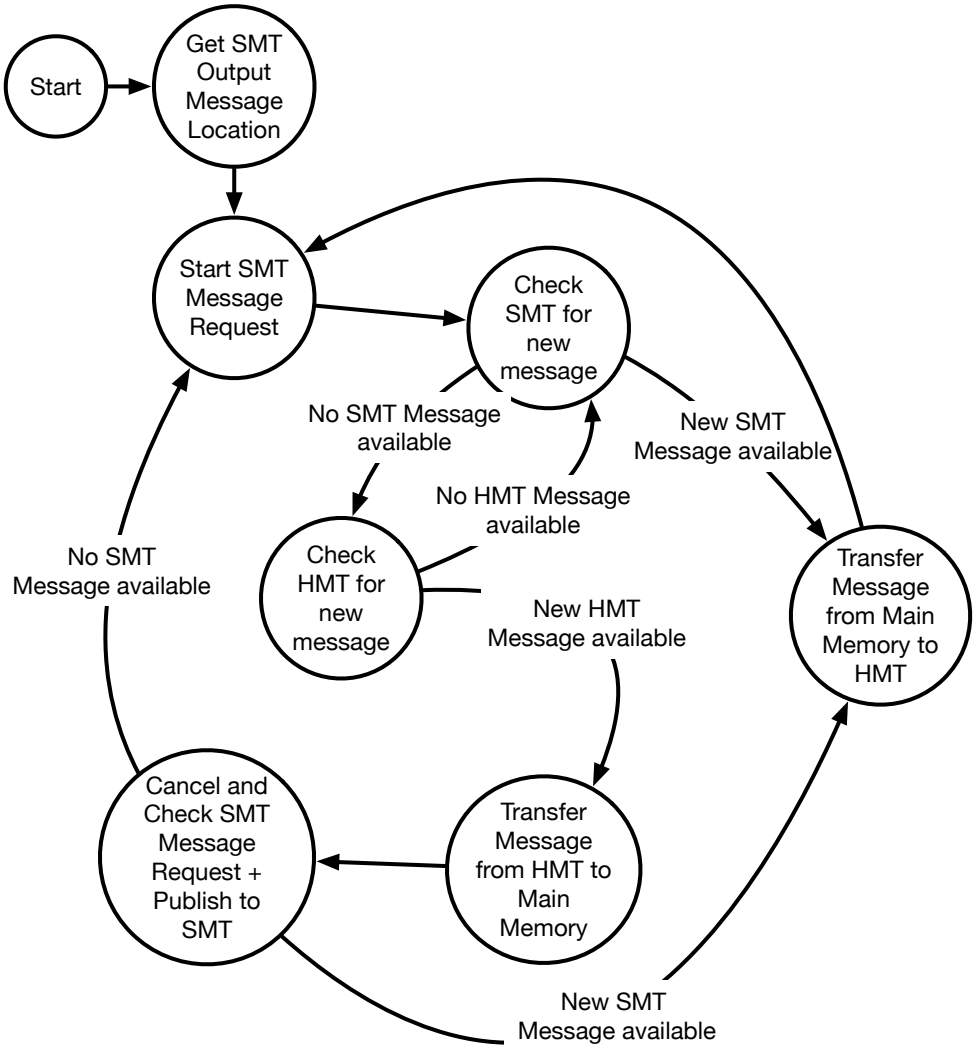

Figure 5.7: Runtime behavior of the gateway core. Adapted from [49].

5.3.2 *Gateway Design Flow*

Gateways are inserted into the ReconROS project using the configuration file. An example is shown in Listing 5.1. The first line of the example instantiates a gateway named Gateway1 of type ROSGateway. This gateway is placed into slot 0 of GatewaySlots and connects the "hwtopicname" and "swtopicname" topics. The message type is Image from the sensor message package of ROS 2.

Listing 5.1: build.cfg definition of a gateway connecting a SMT with a HMT of type *Image*.

```
1   [ROSGateway@Gateway1]
    Slot = GatewaySlots(0)
    HMT = "hwtopicname"
    SMT = "swtopicname"
    MsgType = sensor_msgs/Image
```

After instantiation in the configuration file, an extension of the ReconROS tool flow creates the implementation of the gateway. The procedure is sketched in Figure 5.8. First, the tool flow extension extracts information about the gateway from the ReconROS configuration file (cf. Listing 5.1). Next, it generates memory transfer functions from the hardware-mapped topic into the main memory and vice versa. Since this step requires information about the structure of the messages, the tool flow consults the ROS message definition files here. Second, the resulting functions are inserted into a gateway high-level synthesis template project. This HLS project is then handled by the ReconROS tool flow similar to standard ReconROS HMNs (cf. [50]). Besides the hardware generation procedure, this also includes the start of a separate delegate at runtime.

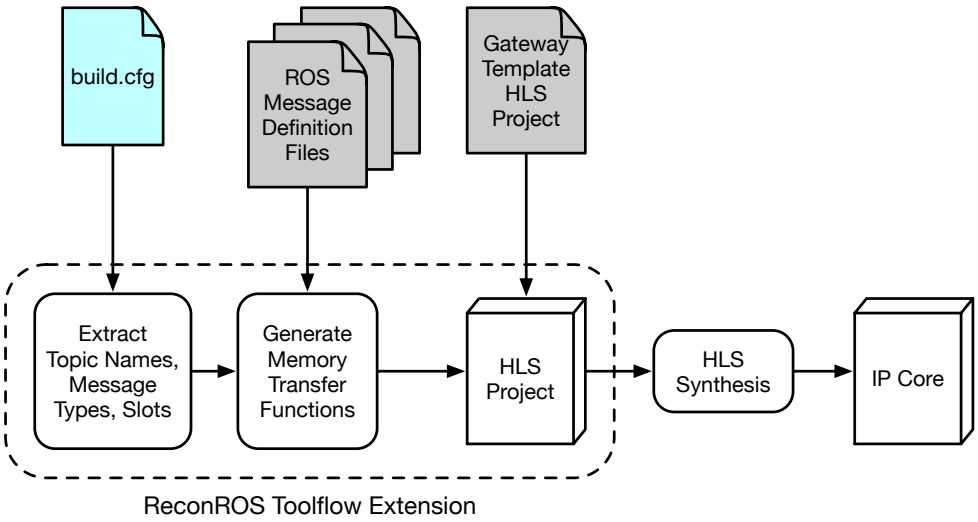

Figure 5.8: Toolflow for the automatic generation of gateways. Taken from [49].

5.3.3 *Performance Measurements*

After describing the architecture and functionality of the gateway, in this section, we evaluate the gateway performance through a synthetic setup. The measurements aim to characterize situations where a gateway should be preferred over an SMT and, thus, develop recommendations for the communication mapping step.

All ROS 2 HMNs and the gateways used for performance measurements have been implemented in C/C++ and synthesized to a hardware description language (HDL) format with the high-level synthesis tool Vivado Vitis 2021.2. The HDL codes and the ReconROS infrastructure were then synthesized to an FPGA bitstream. All SMNs, including the software part of ReconROS, have been compiled using gcc. We have leveraged the ZCU104 evaluation board comprising an UltraScale+ MPSoC FPGA running Ubuntu Linux 20.04, ReconROS, and ROS 2 galactic for the measurements.

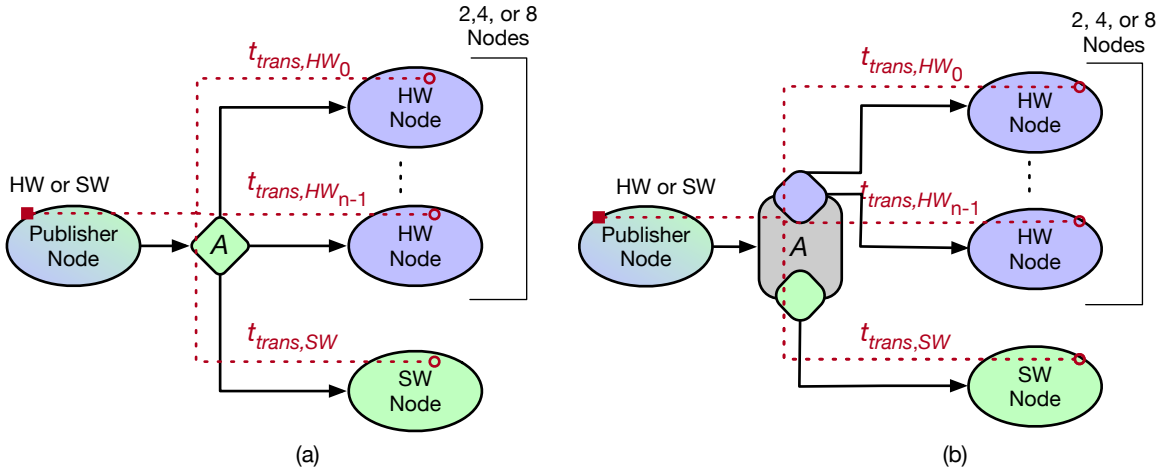

Figure 5.9: Test setup comprising one publishing node (HW or SW), 2, 4, or 8 subscribing HMNs, and one subscribing SMN. Subfigure (a) shows a test scenario with communication via a software-mapped topic, and (b) shows a test scenario leveraging our proposed gateway. Taken from [49].

Figure 5.9 sketches the experimental setup. In Figure 5.9 (a), a publishing node that is either a software-mapped or a HMN generates messages with random data and publishes it to a topic A. The message type is Image from the sensor message package of ROS 2 with image data of $10\,kB$, $100\,kB$, $1\,MB$, and $10\,MB$. A set of receiving nodes subscribe to topic A and copy the received message into their local memory. The subscribing node set comprises one SMN and 2, 4, or 8 HMNs. Figure 5.9(b) shows the same setup but with a gateway A instead of an SMT A. The 12 experiments were repeated 500 times, and the mean values of the measured transmission times are reported.

Figure 5.10 shows the results for the hardware publisher node. The left column of the figure reports the maximum transmission times to any of the subscribing HMNs: $t_{trans,HW} = \max_{0 \leq i < n}\{t_{trans,HW_i}\}$. The results show

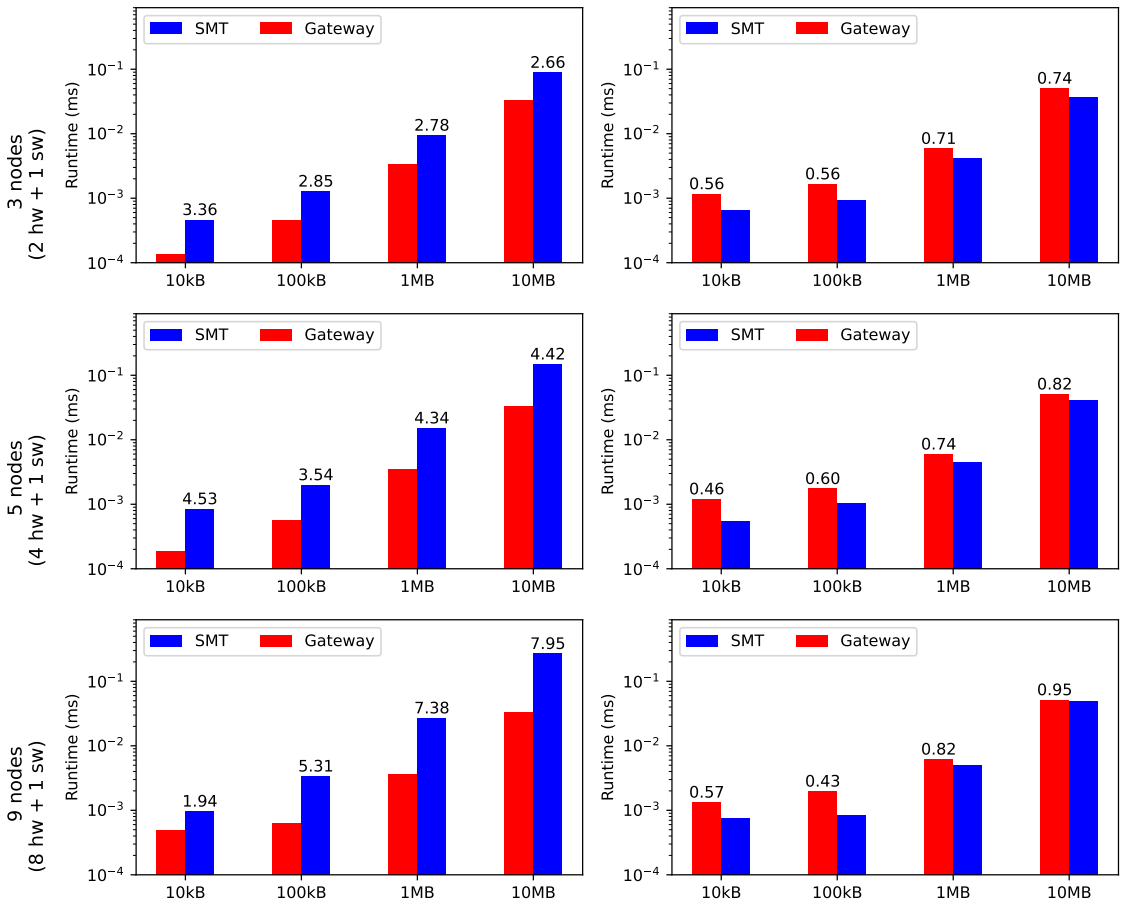

Figure 5.10: Measured maximum transfer times for hardware-to-hardware (left column) and hardware-to-software (right column) communication and the resulting speedups. Taken from [49].

significant speedups for the gateway compared to an SMT as soon as we have more than one hardware-mapped subscribing node. The range of speedups depends on the message size and, for the example of 8 HMNs, ranges from 1.94× for small messages to 7.95× for larger messages.

The right column of Figure 5.10 presents the transmission times for the publishing HMN to the subscribing SMN. Due to its internal design, the gateway introduces overhead for the hardware-to-software transmission. This overhead is significant for smaller message sizes and results in larger transmission times up to a factor of approximately 2. However, the overhead reduces for larger message sizes and increasing HMNs, and the speedup approaches 1×.

Figure 5.11 shows the results for the software publisher node. Again, the left column of the figure displays the maximum transmission times to any of the subscribing HMNs. The gateway overhead leads to speedups below 1× for smaller message sizes. Still, we achieve speedups up to 3.79× for larger message sizes, below those achieved for the hardware publisher node.

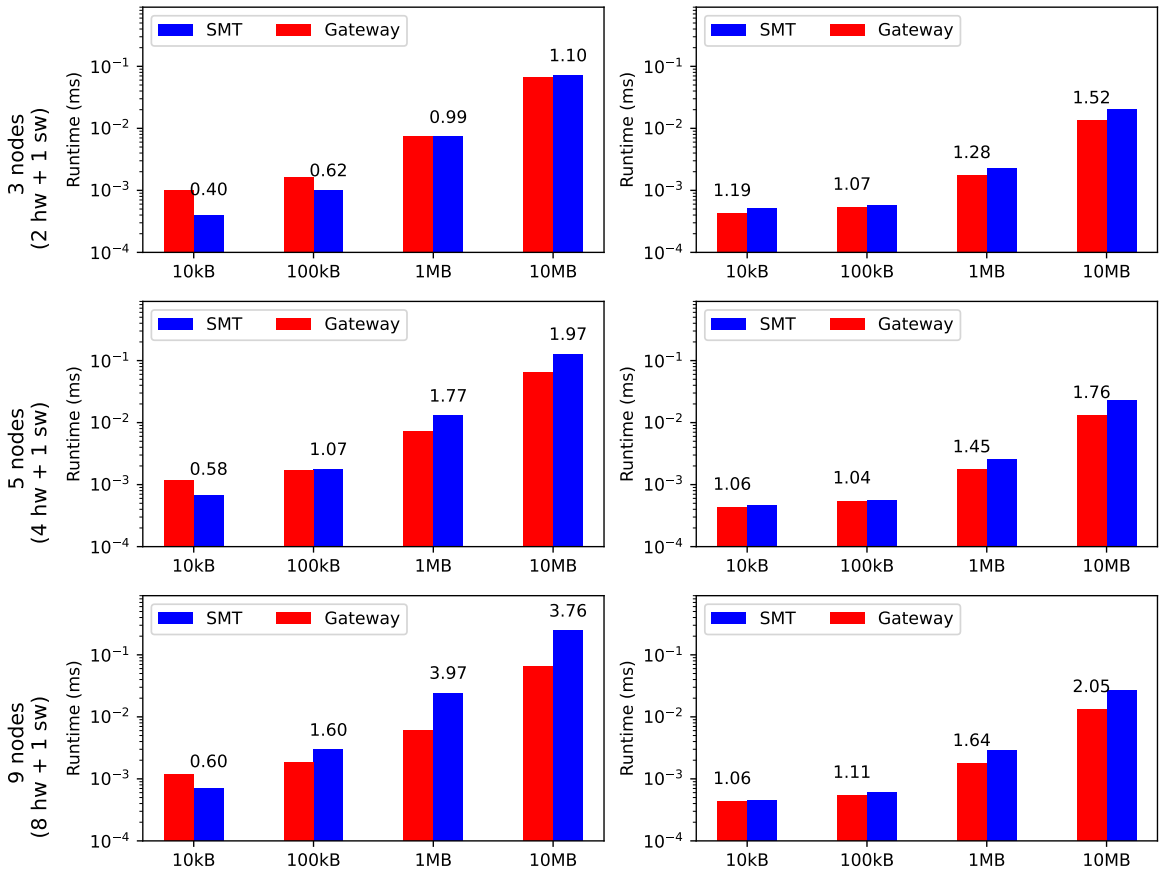

Figure 5.11: Measured maximum transfer times for software-to-hardware (left column) and software-to-software (right column) communication and the resulting speedups. Taken from [49].

The right column of Figure 5.11 presents the transmission times for the publishing SMN to the subscribing SMN. Here, we achieve speedups for the gateway for all measured message sizes and numbers of HMNs. The range of speedups for software-to-software communication is from 1.06× to 2.05×.

In conclusion, publish-subscribe communication with larger message sizes and multiple involved hardware-mapped subscriber nodes significantly benefits from gateways since the required memory bandwidth on the MEMIF is minimized in such situations. The benefits become less significant for smaller message sizes and fewer HMNs.

5.4 COMMUNICATION MAPPING METHODOLOGY

A ROS computation graph can be formally expressed as directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where the set of graph nodes $\mathcal{V}$ comprises both ROS nodes and ROS topics, i.e., $\mathcal{V} = (\mathcal{N}, \mathcal{T})$, and the set of graph edges $\mathcal{E}$ splits into edges indicating a publish function and edges denoting a subscribe function, i.e., $\mathcal{E} = (\mathcal{E}_{pub}, \mathcal{E}_{sub})$ with $\mathcal{E}_{pub} = \{(x, y) \,|\, x \in \mathcal{N}, y \in \mathcal{T}\}$ and $\mathcal{E}_{sub} = \{(x, y) \,|\, x \in$

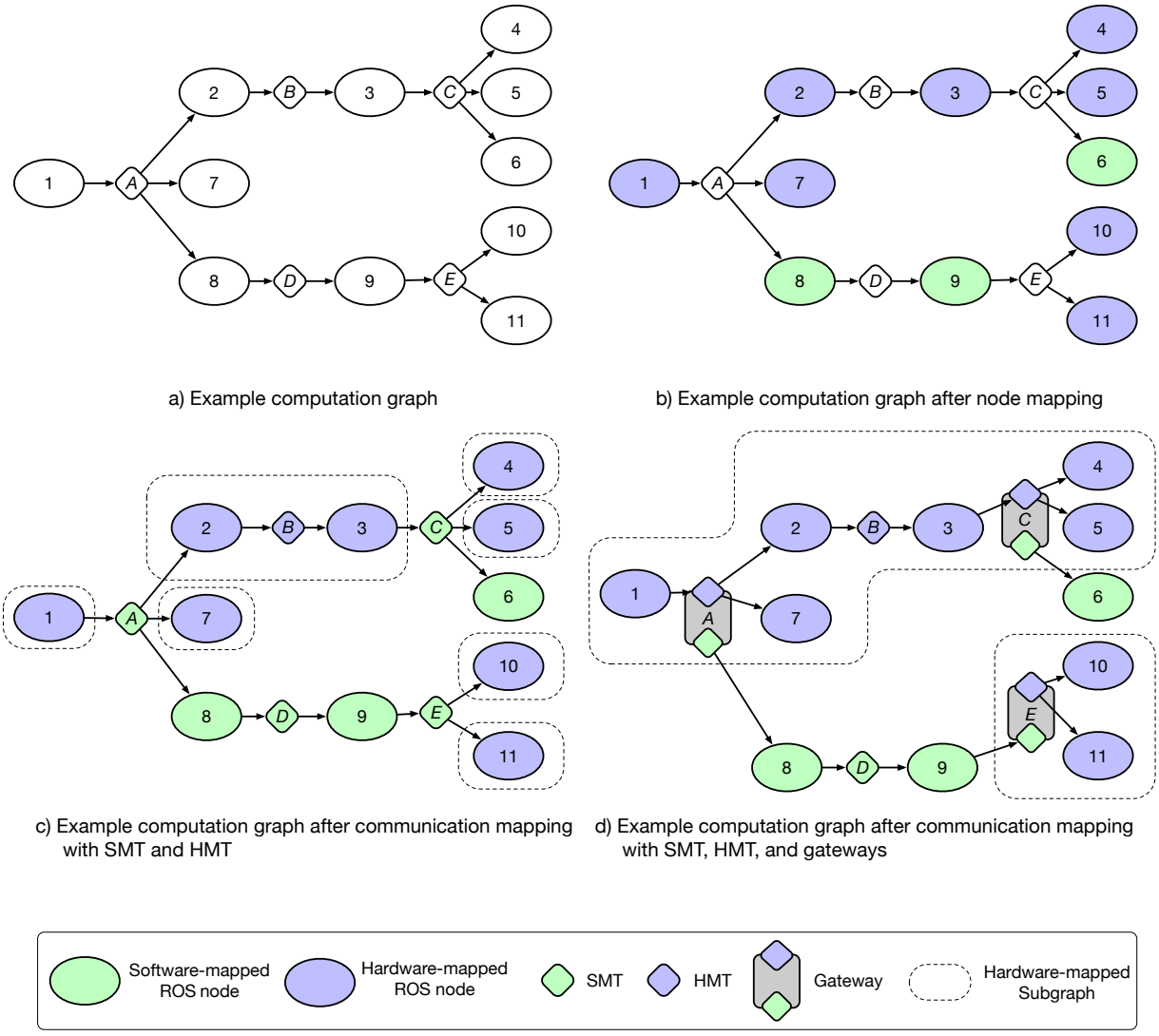

Figure 5.12: Step-wise example for node and communication mapping. Taken from [49]

$\mathcal{T}, y \in \mathcal{N}\}$. Additionally, we define for each topic $t \in \mathcal{T}$ the set of publishing and subscribing ROS nodes as $\mathcal{E}^t_{pub} = \{(x,y) \in \mathcal{E}_{pub} \,|\, y = t\}$ and $\mathcal{E}^t_{sub} = \{(x,y) \in \mathcal{E}_{sub} \,|\, x = t\}$.

In our design flow, mapping a ROS application to a configurable SoC comprises two subsequent steps: (i) node mapping and (ii) communication mapping. Figure 5.12 presents these steps on an exemplary computation graph.

Starting from the original computation graph in Figure 5.12(a), the node mapping step assigns each node $n \in \mathcal{N}$ to either a hardware or a software implementation. We denote the set of nodes mapped to hardware as $\mathcal{N}_{HW}$ and the set of nodes mapped to software as $\mathcal{N}_{SW}$ and, obviously, the node mapping must satisfy $\mathcal{N} = \mathcal{N}_{HW} \cup \mathcal{N}_{SW}$. We currently envision that the developer decides whether to map a specific node to software or hardware. This decision will depend on, e.g., whether an accelerated or energy-efficient

implementation is desirable and available for the node and whether there is logic capacity left in the FPGA. Figure 5.12(b) shows the result of an exemplary node mapping phase, where $\mathcal{N}_{HW} = \{1,2,3,4,5,7,10,11\}$ and $\mathcal{N}_{SW} = \{6,8,9\}$.

The second step is communication mapping, where we assign each topic $t \in \mathcal{T}$ an implementation in software, hardware, or as a gateway. We denote the set of topics mapped to software as $\mathcal{T}_{SW}$, the set of topics mapped to hardware as $\mathcal{T}_{HW}$, and the set of topics mapped to a gateway as $\mathcal{T}_{GW}$. Obviously, the communication mapping must satisfy $\mathcal{T} = \mathcal{T}_{SW} \cup \mathcal{T}_{HW} \cup \mathcal{T}_{GW}$.

The default mapping for topics is to the software since software-mapped topics are the most flexible and can connect any number of software and HMNs. However, if all topics of our running example were actually mapped to software, there are overall ten edges to and from HMNs, three of them are in $\mathcal{E}_{pub}$, and seven are in $\mathcal{E}_{sub}$. Each of these edges will lead to data transfer buffered in the SoC's main memory to or from the configurable logic. When all HMNs execute in parallel, which is a desired scenario and the motivation for hardware acceleration, all these transfers will have to share the available memory bandwidth of ReconROS' MEMIF.

Therefore, we optimize the communication mapping by identifying three cases. First, we search for topics in the node-mapped computation graph for which all the publishers and subscribers are mapped to software. For such topics, the standard software implementation is selected. Formally, we check for each topic $t \in \mathcal{T}$ the following condition: $(\forall (x,t) \in \mathcal{E}^t_{pub} : x \in \mathcal{N}_{SW}) \wedge (\forall (t,y) \in \mathcal{E}^t_{sub} : y \in \mathcal{N}_{SW})$. In our example, the condition only holds for topic D.

Second, we identify topics in the node-mapped computation graph for which all the publishers and subscribers are mapped to hardware. Such topics will then be mapped to hardware, realized with dedicated hardware components of the fpgaDDS layer [48]. Hardware-mapped topics provide much higher communication bandwidth and reduced latency than software-mapped topics. Formally, we check for each topic $t \in \mathcal{T}$ the following condition: $(\forall (x,t) \in \mathcal{E}^t_{pub} : x \in \mathcal{N}_{HW}) \wedge (\forall (t,y) \in \mathcal{E}^t_{sub} : y \in \mathcal{N}_{HW})$. In our example, the condition only holds for topic B.

The remaining topics, for which neither of the above conditions holds, connect software and HMNs. Figure 5.12(c) shows the resulting computation graph if software realizes such topics. The resulting communication mapping is $\mathcal{T}_{SW} = \{A,C,D,E\}$ and $\mathcal{T}_{HW} = \{B\}$. The figure further indicates subgraphs that are mapped to hardware with dashed lines, and it can be seen that the number of edges crossing the software-hardware boundary is reduced from 10 to eight, two of them are in $\mathcal{E}_{pub}$, and six are in $\mathcal{E}_{sub}$.

However, inspecting the topics A, C, and E reveals that they connect to more than one HMN of subscribers. Thus, according to the characterization discussed in Section 5.3.3, it is beneficial, at least for larger message sizes, to implement these topics as gateways. Figure 5.12(d) shows the resulting computation graph mapping with $\mathcal{T}_{SW} = \{D\}$, $\mathcal{T}_{HW} = \{B\}$, and $\mathcal{T}_{GW} = \{A,C,E\}$. Again, the figure indicates subgraphs that are mapped to

hardware with dashed lines, and the number of edges crossing the software-hardware boundary is finally reduced to 3, one from gateway A to the SMN 8, one from gateway C to the SMN 6, and the last one from the SMN 9 to the gateway E.

5.5 EVALUATION

This section reports on experiments with fpgaDDS. First, we describe the experimental setup, followed by experiments with HMNs to demonstrate the benefit of using fpgaDDS. Finally, we present measurements on a real-world application of autonomous driving.

All HMNs have been implemented in C++ and synthesized to hardware with the high-level synthesis tool Xilinx Vitis 2021.2. The computing platform is a ZCU104 evaluation board comprising an UltraScale+ MPSoC FPGA running Ubuntu Linux 18.04, ReconROS, and ROS 2 dashing. The ReconROS infrastructure and the HMNs run at 100 MHz. For the autonomous driving example, we additionally use a desktop PC with an Intel Core i5-8000 CPU running Ubuntu Linux 18.04 and ROS 2 dashing, connected to the FPGA evaluation board via Gigabit Ethernet.

In the experiment, we have implemented two HMNs connected by a topic and measured the time it took from the start of sending to the end of receiving. Data transfers have been repeated 1000 times, and the average values are reported. Table 5.1 presents the resulting transfer times t_{avg} and their standard deviation σ for HMNs using the software-mapped CycloneDDS and the hardware-mapped fpgaDDS for different message sizes. The table also lists the speedups achieved by fpgaDDS. The speedups show that the overhead for using CycloneDDS dominates for small data sizes. However, the speedup converges close to 2× for more extensive data sizes because data has to be transferred twice over the MEMIF compared to one needed transmission via the HMT. However, the measurements for CycloneDDS represent the most optimistic case. With higher utilization of the MEMIF, the transmission time increases as the nodes share MEMIFs bandwidth.

Message Size	CycloneDDS t_{avg} (σ) [ms]	fpgaDDS t_{avg} (σ) [ms]	Speedup
3 kB	0.06 (0.11)	<0.01 (<0.01)	17.21×
12 kB	0.07 (0.06)	0.02 (<0.01)	4.71×
50 kB	0.19 (0.08)	0.06 (<0.01)	3.11×
196 kB	0.63 (0.04)	0.24 (<0.01)	2.54×
786 kB	2.34 (0.03)	0.98 (<0.01)	2.38×
3146 kB	9.24 (0.04)	3.93 (<0.01)	2.35×

Table 5.1: Communication times and speedup for CycloneDDS and fpgaDDS for different message sizes.

Increasing the number of publishers and subscribers increases the transfer time for CycloneDDS. For fpgaDDS, the transfer time only increases with more publishers since the HMT arbitrates the incoming messages. However, adding more subscribers in fpgaDDS does not influence the transfer time.

5.6 CHAPTER CONCLUSION

This chapter presents approaches for the optimization of communication between nodes. Firstly, the extensions for the support of zero-copy transmissions reduce communication overhead. For the validation of these extensions, we refer to the evaluation of the AutonomROS project in Chapter 6 (cf. Table 6.5), in which significant performance improvements are achieved through the use of zero-copy communication. Secondly, fpgaDDS and its gateway extension reduce communication overheads between HMNs and, therefore, relax the interface between hardware and software while preserving the programming model of ROS 2. Besides the evaluation in this chapter, the Turtlebot design example (cf. Section 6.2) demonstrates the advantages of fpgaDDS and gateways in a more extensive application and validates them for real-world scenarios. The formal representation of our communication mapping methodology 5.5 can be used for automatic mapping in our framework in future work.

6

RECONROS CASE STUDIES

This chapter reports on different example applications realized using ReconROS.

The chapter starts with a Ball-on-plate demonstrator application in Section 6.1, which was first mentioned in the master project of Christoph Rueting [102]. In the follow-up master's thesis [46], we have used parts of the implementation and extended the physical setup by two additional platforms. The author also extended the hardware-software codesign by a Kalman filter in hardware and software and video input, output, and processing. Finally, the implementation was redesigned using ReconROS and used in the ReconROS TRETS publication [50]. The application demonstrates ReconROS applicability for distributed systems across multiple SoCs.

The second demonstrator application in Section 6.2 of this chapter presents an example of an autonomous driving architecture. The driving architecture can follow a street lane and handle traffic lights during the drive so that it can stop for red traffic lights and start when the light becomes green again. The architecture was first mentioned in the IRC publication [52]. Further extensions and adaptations followed for publication at the IROS [48] and ICRAI [49] conference, for which hardware-mapped topics were used in the first step, and the architecture was then expanded to include a gateway and an ORB-SLAM component. The Turtlebot architecture presents a real-world use-case scenario benefiting from the communication optimizations such as fpgaDDS and gateways described in Chapter 5.

Finally, this chapter closes with Section 6.3 about AutonomROS. The AutonomROS architecture was designed during a student project group at Paderborn University. The architecture based on the powerful open-source ROS 2 package Navigation 2 shows ReconROS-based hardware application in combination with standard ROS 2 packages. The descriptions in this section follow the publication accepted for publication at the IRC 2023 [47]. This application shows the compatibility of ReconROS with standard ROS 2 packages such as Nav2 and the advantages of zero-copy communication (cf. Section 5.1) together with hardware acceleration.

6.1 BALL ON PLATE DEMONSTRATOR

To showcase the suitability of ReconROS for distributed hardware-accelerated ROS 2 applications, we present the mechatronics model [46] shown in Fig-

ure 6.1, that we have physically implemented. Recently, there have been approaches for including reconfigurable hardware into distributed embedded systems, e.g., ReCoNets [37], LMGS [13] or RSS [18], but these approaches are not compatible with existing and widely-used software abstractions for creating distributed robotics systems.

The model comprises three ball-on-plate stations that can balance a mechanical platform such that a ball thrown onto the platform does not fall off. To this end, we employ a Stewart platform [107] that allows the system to move an object in six degrees of freedom, including linear translations in x, y, and z direction but also three rotations (pitch, roll, and yaw). Stewart platforms perfectly suit high-dynamic mechatronics applications such as flight simulators or telescopes. In our setup, we drive six servo motors by pulse-width modulated signals to adjust corresponding angles between the motor axes and the legs connecting to the platform, resulting in the desired movement. To capture the position (x, y) of the ball on the platform, we use a resistive touchscreen mounted on the surface of the platform.

Additionally, each ball-on-plate station has a monitor, and a camera captures all stations.

The computing infrastructure includes three ZedBoards, as outlined in Figure 6.1. Each ZedBoard has a Xilinx Zynq-7020 platform FPGA and runs Ubuntu 18.04 and ReconROS based on ROS 2 dashing. The servo actuators and touchscreen sensors are connected to ZedBoard-Main, the camera is connected to ZedBoard-1, and the monitor inputs on the three ball-on-plate stations are driven by a ZedBoard each. All compute platforms are connected to an Ethernet network.

6.1.1 *Architecture*

The overall ROS 2 application splits into two parts: the control of the ball-on-plate stations and a video processing chain. Figure 6.2 shows all involved ROS 2 nodes with their communication objects. The control loop for a ball-on-plate station comprises the four ROS 2 nodes *Touch*, *Control*, *Inverse*, and *Servo*. The *Touch* node starts a new control cycle by reading the actual position of the ball on the platform. This information is scaled and sent to the *Control* node that implements a PID controller and a Kalman filter to determine the desired rotations for the platform concerning the x and y axes. The subsequent *Inverse* node applies inverse kinematics transformations to determine the required angle for each of the six servo motors. Finally, the *Servo* node converts the angles into pulse width modulated signals to drive the motors.

The video processing chain includes ROS 2 nodes for video input, *HDMI in*, processing, *Filter*, and video output, *HDMI out*. The HDMI interface implementation includes mechanisms for transporting image data from and to the main memory without processor interaction by using AXI video direct memory access (VDMA).

All ROS 2 nodes use publish/subscribe mechanisms to communicate with topics shown in Figure 6.2.

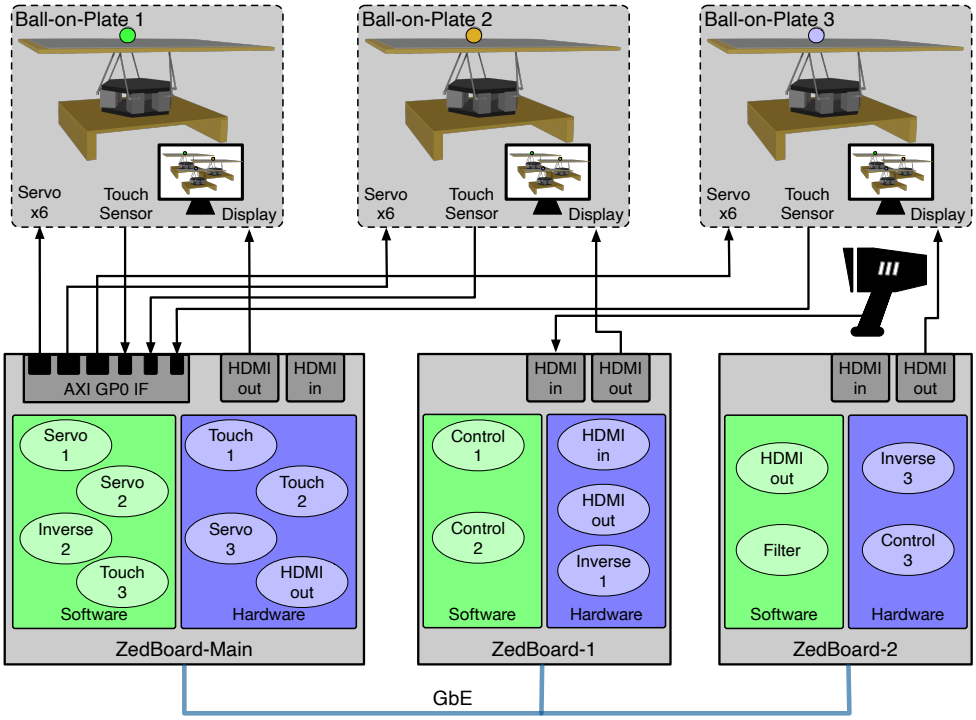

Figure 6.1: Mechatronics model based on three ball-on-plate stations with Stewart platforms. Adapted from [50].

6.1.2 *Evaluation*

We have realized all ROS 2 nodes in software and hardware. Table 6.1 lists the raw node runtimes. The hardware implementations of the inverse kinematics and the filter nodes can exploit low-level parallelism and achieve speedups. All other nodes are either more control-flow intensive, exhibit little computation, or are bound by the memory bandwidth and are thus better mapped to software.

Given that both software and hardware implementations for the ROS 2 nodes are available, developers can efficiently distribute the nodes across the boards in the network, change the mapping of nodes in the project configuration files, and rebuild the system. One specific example of such a mapping of nodes is indicated in Figure 6.1. With this mapping, the sampling time of the *Touch* node and, thus, the control loop could be set to 20 *ms*, which results in relatively smooth movements of the Stewart platforms. Table 6.2 lists the resources required for this specific mapping, including the actual HMNs, the necessary ReconROS infrastructure, and the components needed for the HDMI input and output interfaces.

The implemented system is not a hard real-time system with a guaranteed sampling period of 20 ms. Creating a hard real-time system would require

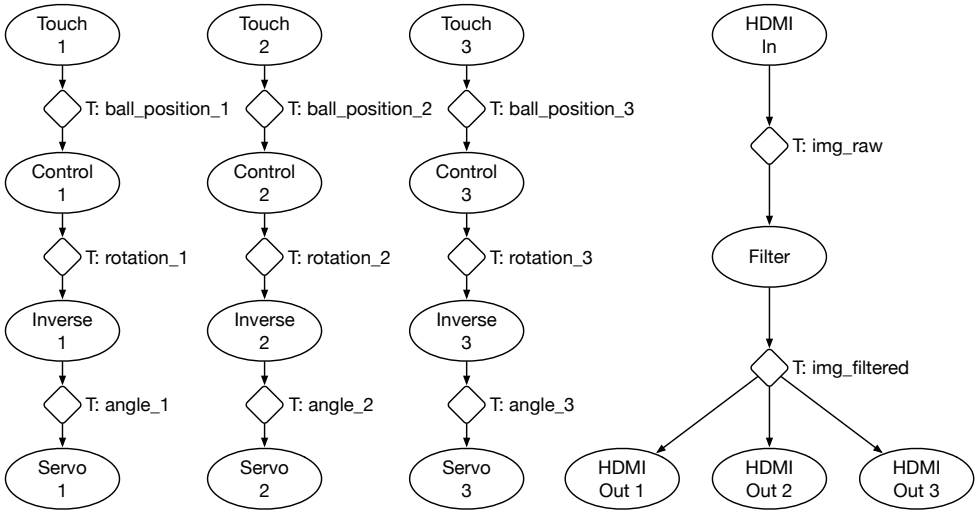

Figure 6.2: ROS 2 application with node and communication objects for the mechatronics model shown in Figure 6.1. Adapted from [50].

ROS 2 node	$t_{raw-node-SW}$ [ms]	$t_{raw-node-HW}$ [ms]	$S_{raw-node}$
Servo	0.001	< 0.001	$\approx 1\times$
Control	0.017	0.030	$0.57\times$
Inverse	1.430	0.196	$7.30\times$
Touch	0.001	< 0.001	$\approx 1\times$
HDMI In	5.160	18.460	$0.28\times$
HDMI Out	4.590	18.400	$0.25\times$
Filter	37.530	22.280	$1.68\times$

Table 6.1: Runtimes for the raw ROS 2 nodes of the mechatronics example in software and hardware. Taken from [50].

modifying ReconROS and the underlying ROS 2 and Linux layers and substituting Ethernet communication with a real-time version, which is not part of this thesis.

Moreover, this example also does not address the optimization of the mapping of nodes between hardware and software and across the FPGA board. However, to demonstrate the trade-offs involved, we have created three mappings of the mechatronics application and measured the processing times of the three control loops for 300 sampling periods. Figure 6.3 displays the relative frequencies for the resulting processing times for all three control loops, i.e., for the three Stewart platforms (columns 1-3) and three different ROS 2 mappings (rows 1-3). The figure shows the processing time frequencies from 0 to 20 ms and provides the percentage of missed deadlines, where the deadline has been set to 20 ms.

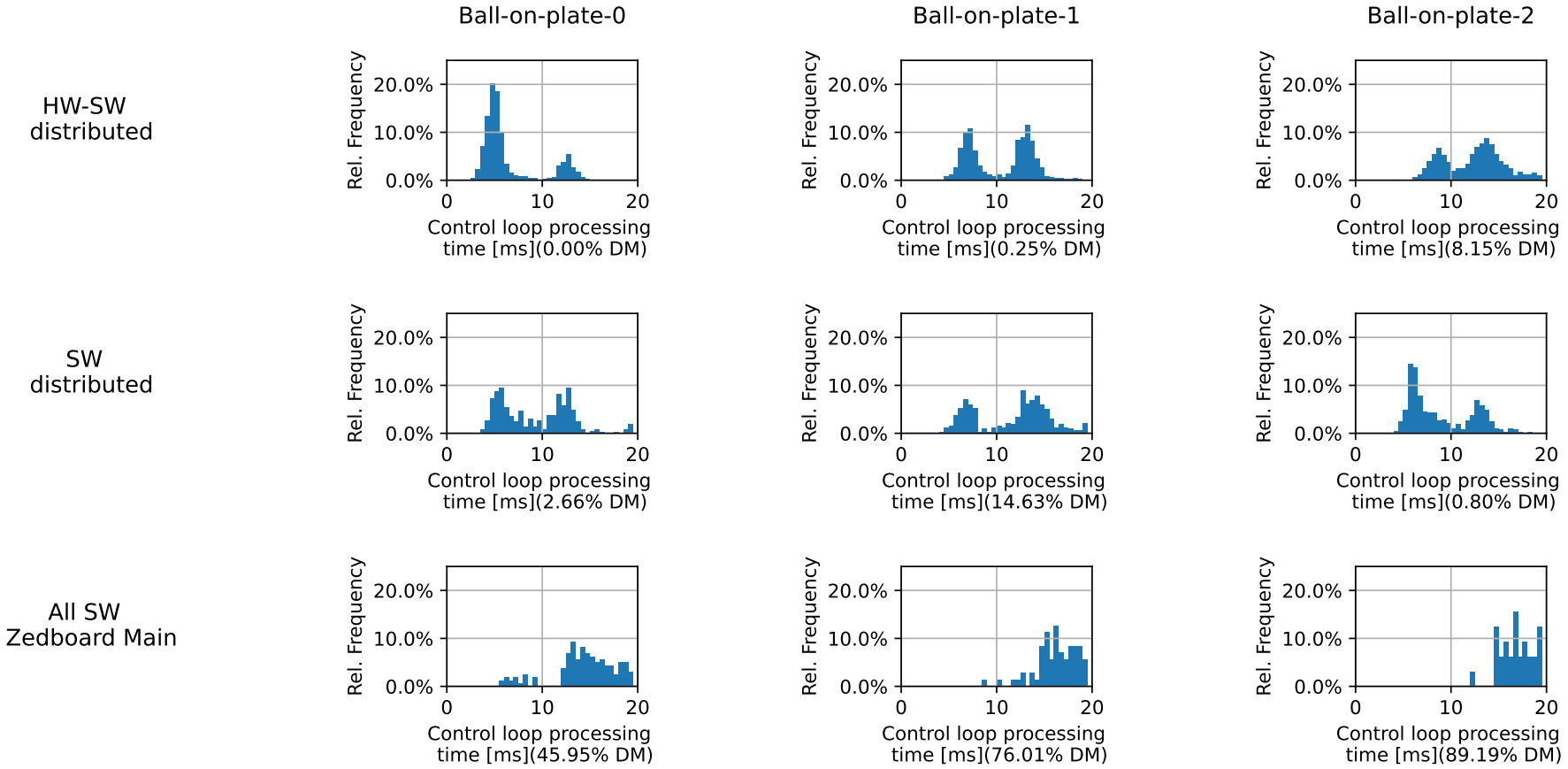

Figure 6.3: Relative frequencies of measured processing times for the three control paths (Touch→Control→Inverse→Servo) and three different node mappings (DM = Deadline Missed [%]). Taken from [50].

Board	FPGA	Slice LUTs	DSP	BRAM
Zedboard Main	Zynq-7020	13467 (25.31%)	0 (0.00%)	3 (2.14%)
Zedboard 1	Zynq-7020	13235 (24.88%)	77 (35.00%)	3 (2.14%)
Zedboard 2	Zynq-7020	13031 (24.49%)	77 (35.00%)	3 (2.14%)

Table 6.2: Resource usage and utilization (in % of the Xilinx Zynq 7020) for the three involved FPGA boards. Resource figures are reported for slice LUTs, DSP, and BRAM. Taken from [50].

The first row uses the mapping from Figure 6.1, which distributes the nodes over all three FPGA boards, software, and hardware. This mapping reaches all deadlines for platform 1, misses 0.25% of the deadlines for platform 2, and 8.15% for platform 3. The second row shows the same distribution of nodes across the three FPGA boards but maps all nodes to software. In this case, the fraction of missed deadlines is relatively low on platforms 1 and 3, and a value of 14.63% somewhat higher on platform 2. Finally, the mapping of row 3 places all nodes in software on Zedboard-Main with the result that most of the deadlines are missed. The Stewart platforms for mappings 1 and 2 move relatively smoothly, but for mapping 3, the platforms show very jerky movements, making the application unusable.

6.2 TURTLEBOT 3 AUTORACE

In this application example, we elaborate on an architecture for an autonomous driving application. The Turtlebot auto race challenge inspires this application, which follows a street lane and handles traffic lights during the drive. The application was used in an example application in several publications [48, 49, 52] and was extended step-wise. The first version of the application was presented in the IRC publication [52] to demonstrate a use case of the ReconROS executor for a real-world scenario. In this implementation, the ReconROS executor changed between two states of the robot: driving or waiting for a green traffic light. Then, the application was redesigned for the usage of fpgaDDS to optimize the computational performance of the application. Since fpgaDDS supports static-mapped hardware nodes only, the ReconROS executor was removed. In its final implementation, the application was extended by the popular ORB-SLAM 3 algorithm [16] to demonstrate the benefits of gateways for hardware-mapped topics.

6.2.1 *Architecture*

The target robot is the Turtlebot 3 platform, comprising a 0.3-megapixel color camera (640x480 pixels) for lane and traffic light detection.

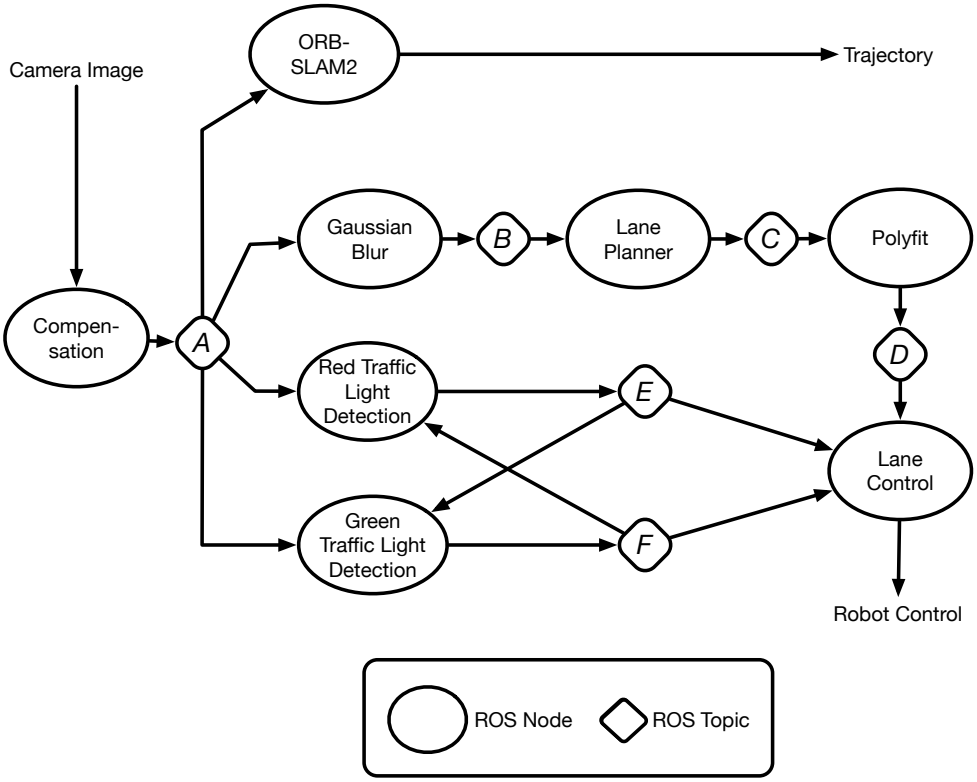

Figure 6.4: Computation graph for the autonomous vehicle example. Taken from [49].

Figure 6.4-a displays the computational graph for the application. The only input of the computation graph is the camera image of the robot. The *Image Compensation* node calculates a histogram of the input image and uses it to remove outlier pixels. The resulting image is then published to *Topic A*. The *Gaussian Blur* node subscribes to the image data, applies a low-pass filter, and forwards the result to the image projection node through *Topic B*. The *Image Projection* node applies a warp transformation for an orthogonal view of the lane, then resizes the image to 1000x600 pixels. *Lane Planner* computes color space conversation to the HSV color space. The node generates a mask for yellow and white pixels from the transformed image by checking pixel-wise for specific ranges. After combining both masks, the perspective of the combined mask is transformed into a bird's view. The resulting image is published on the *Topic C*. The *Polyfit* node subscribes to that topic and generates an approximation of the lane out of it, solving a least-squares problem. Second, we have extended the computation graph compared to former versions by a node implementing the popular ORB-SLAM 3 algorithm [16] for creating a map of the environment. Without using gateways for hardware-mapped topics, the subscription of the output of the *Image Compensation* node would result in falling back to standard ROS 2 communication for the *Topic A* and, therefore, potentially slow down the application due to memory transfer overheads.

In addition to the *Gaussian Blur* node, the *Red Traffic Light Detection* and the *Green Traffic Light* nodes subscribe to *Topic A*. Both nodes transform the image into the HSV colorspace and then detect a red or green traffic light. If a red traffic light is detected, the node publishes a stop command to *Topic E*, which leads to a stop of the robot by the lane control node. Additionally, that command activates green traffic light detection. When a green light is detected, the node publishes a start command to *Topic F*, which activates the lane control node again.

6.2.2 *Evaluation*

We have implemented and evaluated the application example in a hardware-in-the-loop (HiL) environment. We have used Gazebo under Ubuntu 20.04 for the application evaluation with ROS 2 galactic running on a desktop PC connected via Gigabit Ethernet to a ZCU104 evaluation board. The setup for this experiment is shown in Figure 6.5.

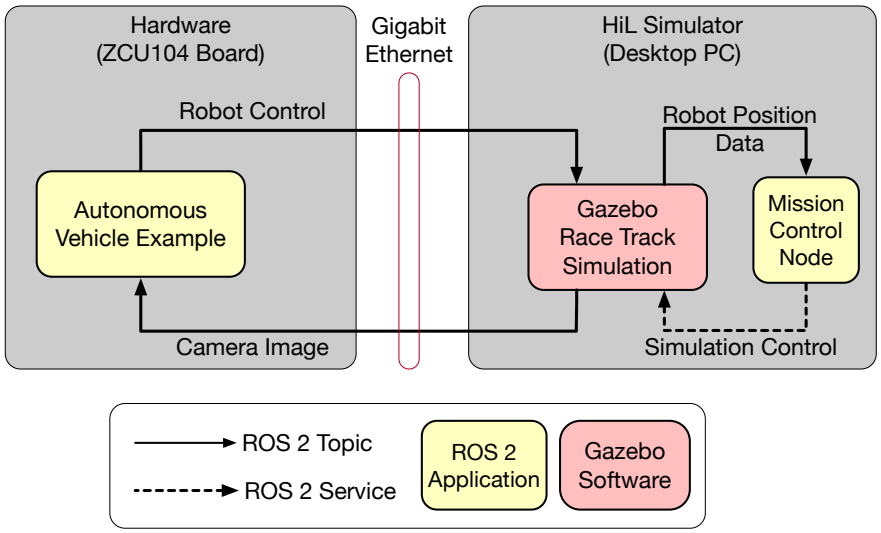

Figure 6.5: Overview of the HiL simulation environment for the application. Taken from [52].

The implementation of the control algorithm of the robot was partly inspired by the official Turtlebot Autorace 2020 ROS package [117], but has been entirely rewritten in C++. The desktop PC with Intel Core i5-8000 CPU performs the task of a HiL simulator and runs Gazebo [31], an open-source simulator for 3D robotics applications, to simulate the vehicle's environment on a racetrack. Similar to the desktop PC, the evaluation board runs Ubuntu 20.04 with ROS 2 galactic and ReconROS. The ZCU104 FPGA board acts as the robotic control board and executes the applications shown in Figure 6.4.

The simulator provides several interfaces for interacting with the robot and the robot's environment for modifying and controlling during runtime. However, we have changed the standard simulated environment of the robot to an urban environment by adding buildings. The resulting environment

Figure 6.6: Simulated Gazebo environment based on a modified version of the official Turtlebot Autorace 2020 challenge. In addition to the environment, the housing of the Turtlebot 3 was simplified to achieve a better simulation performance.

is shown in Figure 6.6. The buildings on the sides of the roads allow the ORB-SLAM3 algorithm to detect remarkable key points for creating a map.

The *Image Compensation* node of Figure 6.4 subscribes to camera image data from the Gazebo simulation, and the *Lane Control* node publishes robot control data through gigabit ethernet to the simulator. A Python-based mission control node subscribes to the robot's actual position data and derives simulation control commands for simulation control.

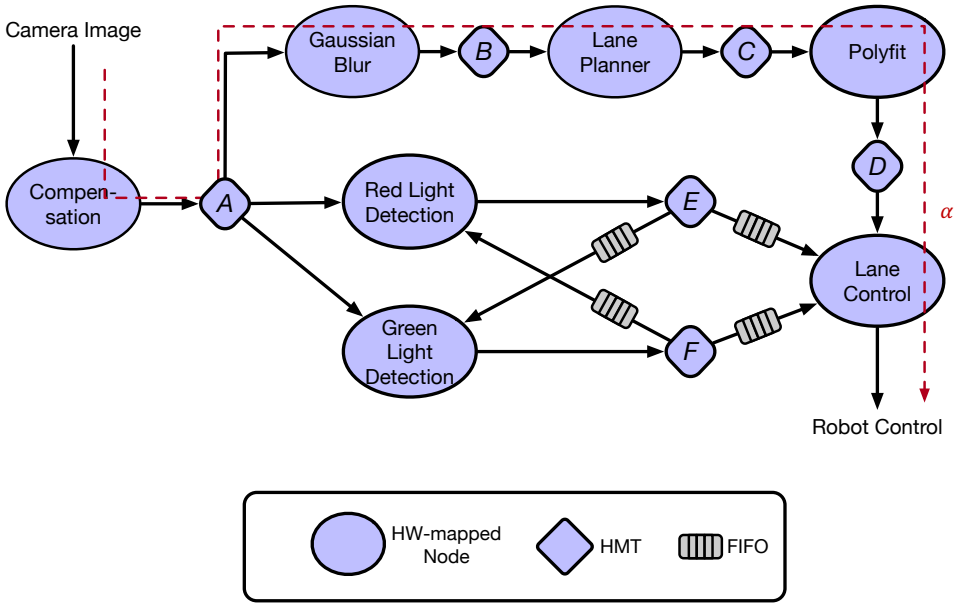

Figure 6.7: Architecture graph for the autonomous vehicle example using fpgaDDS without ORB-SLAM3. Taken from [48].

For the demonstration of the performance of fpgaDDS, we have implemented the computation graph without the ORB-SLAM node. Since the ORB-SLAM3 node is too complex for a redevelopment in hardware, it will be removed for the first step of the demonstration. Therefore, we have implemented the ROS 2 application of Figure 6.4 in three different versions: (i) Software, where nodes run on the CPU based on CycloneDDS for communication, (ii) hardware, where all nodes are implemented using high-level synthesis but still use CycloneDDS for communication, and (iii) hardware, where the nodes use our fpgaDDS. The architecture graph for the third version is sketched in Figure 6.7. In addition, the subscribers of HMT *Topic E* and *F* are equipped with buffers to enable asynchronous communication via these HMTs.

The software implementation does not use any reconfigurable logic resources. Table 6.3 summarizes the resource utilization of the reconfigurable fabric for the hardware implementations. The table lists the resource types lookup-tables (CLBs), dedicated memory blocks (BRAM, UltraRAM (URAM)), and arithmetic function blocks (DSP). The central insight from Table 6.3 is that fpgaDDS does not require more logic resources than the version with CycloneDDS. The higher logic consumption is because the AXIS-based communication architecture requires relatively few logic resources. CycloneDDS, on the other hand, demands substantially more CLBs for implementing in-node buffers.

Implementation	CLBs	BRAM	URAM	DSPs
Hardware (CycloneDDS)	134252 (58%)	250.5 (80%)	95 (99%)	230 (13%)
Hardware (fpgaDDS)	44939 (20%)	255.0 (82%)	91 (95%)	230 (13%)

Table 6.3: Resource utilization of the hardware implementations (% of the used XCZU7EV-2FFVC1156)

To evaluate the performance of the different communication middlewares, we have measured the execution times of the node chain α indicated as a red dashed line in Figure 6.7. The node chain starts with the image compensation node and ends with the lane control node. We have selected this chain of nodes since it subsumes most nodes regularly executed in the lane-following mode of the vehicle and feeds the topics periodically with message data. In contrast, topics *E* and *F* receive data only in the event of detected red or green traffic lights.

We have determined the execution of time node chain α by calculating the difference between the publishing time of the lane control node and the subscription time of the image compensation node. Again, we repeat the experiments 1000 times and report on the average and the standard deviation in Table 6.4. Additionally, the table presents the speedups achieved over a pure software implementation. For example, implementing the ROS nodes in

hardware but keeping the software-based CycloneDDS results in a speedup of 2.48×. Mapping also communication to hardware with fpgaDDS gives a speedup of 13.34×. Furthermore, with fpgaDDS, the standard deviation, which relates to jitter, reduces by two orders of magnitude.

Implementation	t_{avg} (σ) [ms]	Speedup
Software (CycloneDDS)	274.44 (14.96)	1.00×
Hardware (CycloneDDS)	110.72 (07.73)	2.48×
Hardware (fpgaDDS)	20.58 (00.13)	13.34×

Table 6.4: Execution times and speedups for node chain α of Figure 6.7 and different implementation variants.

In the following, we will consider the architecture from Figure 6.4, including the ORB-SLAM 3 node. Without gateways, communication on topic A would now lead to a fallback to standard ROS 2 communication, since now not only HMNs interact with the topic.

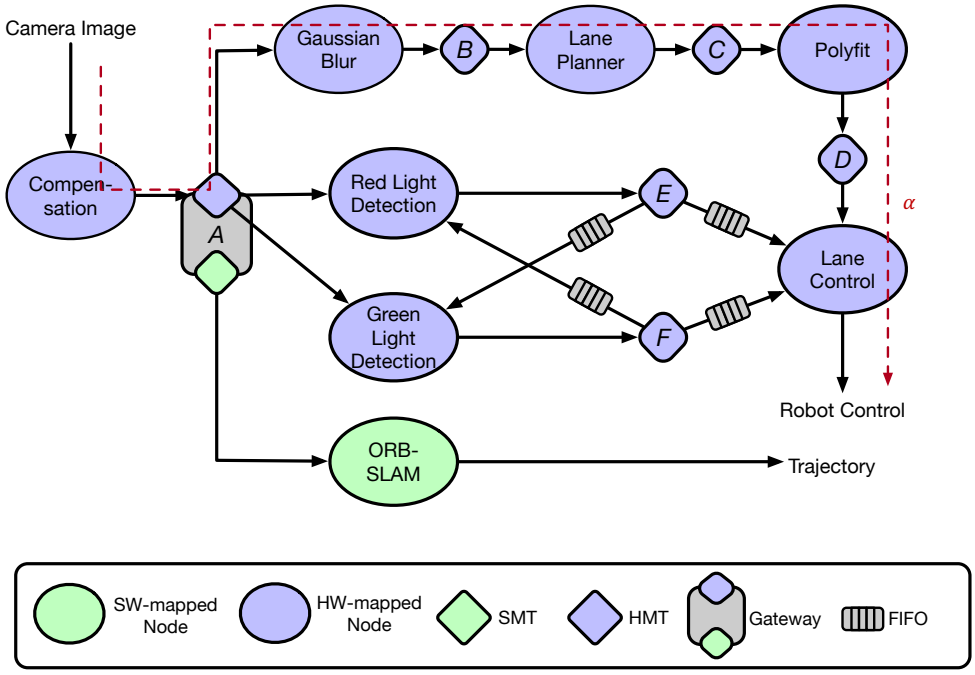

Figure 6.8: Architecture graph for the autonomous vehicle example including ORB-SLAM3 using fpgaDDS and gateways. Taken from [49].

For the mentioned showcase, we have implemented the computation graph in two versions: The first version uses a hardware-mapped topic gateway to map topic A into the hardware domain. We use standard ROS 2 communication for the second version. For both versions, we have measured

the execution time of the node chain α (red dotted line in Figure 6.8). For the implementation using standard ROS 2 communication for topic A, the mean execution time for the node chain is 28.264 ms with a standard deviation of 3.21 ms. For the implementation leveraging the proposed gateway, we have measured a mean execution time of 20.270 ms and a standard deviation of 0.239 ms. In summary, we achieved an average speedup of 1.4× and a one-order of magnitude reduced standard deviation and, therefore, validated the concept of gateways in a real-world example scenario.

6.3 AUTONOMROS

This section presents AutonomROS, an autonomous driving 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 SMNs, and ROS 2 nodes completely mapped to hardware (HMNs). On the one hand, AutonomROS uses zero-copy communication based on Iceoryx (cf. Section 5.1) for ReconROS, which improves the performance of shared-memory inter-process communication between HMNs and SMNs. On the other hand, AutonomROS demonstrates hardware acceleration for the functions of 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 SMNs, and nodes accelerated in hardware.

6.3.1 *Architecture*

Figure 6.9 shows the top-level overview of the architecture of the AutonomROS autonomous driving unit. Besides 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.

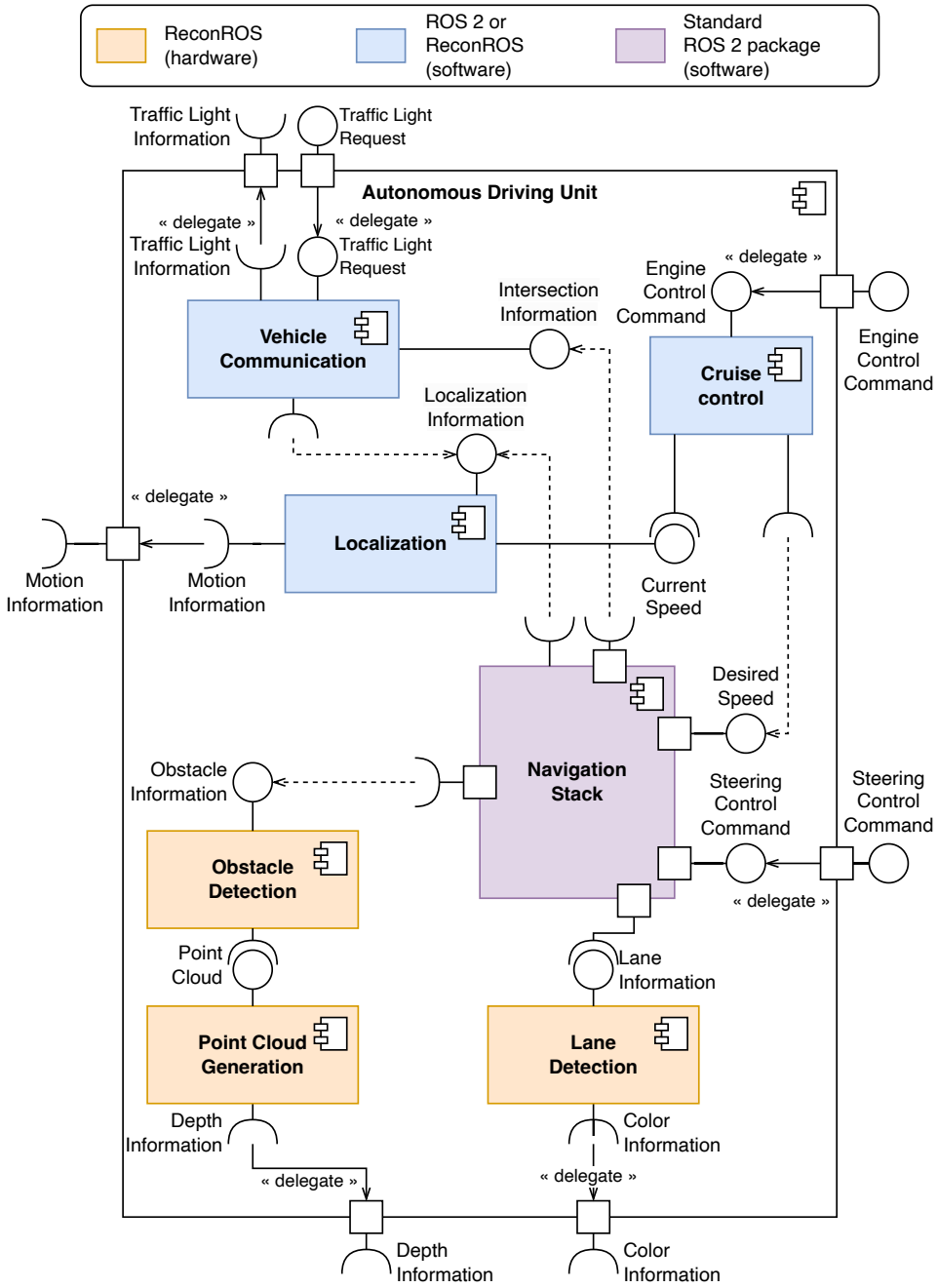

Figure 6.9: Architecture of the AutonomROS autonomous driving unit. Hardware-accelerated components are highlighted in blue color. Taken from [47].

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 during this chapter. 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 implemented as ROS 2 software nodes. Finally, the component *Navigation Stack* is based on Nav2, an open-source ROS 2 package.

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 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 6.1. 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) [15]. The camera's wrapper node publishes the matrix as a ROS 2 `CameraInfo` message. 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.

$$P = \begin{bmatrix} fx & 0 & cx & Tx \\ 0 & fy & cy & Ty \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{6.1}$$

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 6.2 - 6.4, which are taken from the description of the `CameraInfo` message of ROS [15].

$$X = \frac{u - cx \cdot w - Tx}{fx} \tag{6.2}$$

$$Y = \frac{v - cy \cdot w - Ty}{fy} \tag{6.3}$$

$$Z = w \tag{6.4}$$

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.

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 [122].

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.

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 6.5.

$$\begin{bmatrix} N & \dots & \sum_{k=0}^{N} x_n^k \\ \vdots & \ddots & \vdots \\ \sum_{n=0}^{N} x_n^k & \dots & \sum_{k=0}^{N} x_n^{2k} \end{bmatrix} \cdot \vec{a} = \begin{bmatrix} \sum_{n=0}^{N} y_n \\ \vdots \\ \sum_{n=0}^{N} y_n x_n^k \end{bmatrix} \tag{6.5}$$

For implementing the *Lane Detection* component, we have determined that the second-order polynomial shown in Equation 6.6 is suitable.

$$f_l(x) = a_2 \cdot x^2 + a_1 \cdot x + a_0 \tag{6.6}$$

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 6.6).

Using the resulting 30 coordinates (y_i, x_i), the equation system 6.5 is solved again for a target polynomial function $f_t(x)$ (Equation 6.7).

$$f_t(x) = a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0 \tag{6.7}$$

6.3.2 *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.

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). We used the quad-core CPU and the programmable logic for our evaluations. The board runs Ubuntu 20.04 with ROS 2 galactic and ReconROS.

We have mounted the evaluation board on a model car platform shown in Figure 6.10. 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.

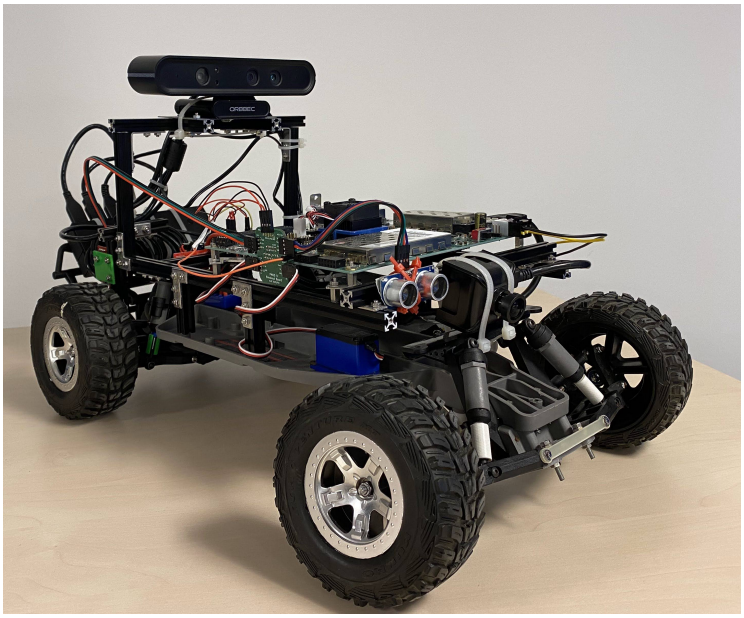

Figure 6.10: Model car platform. Taken from [47].

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.

Table 6.5 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×, and the turnaround time decreased by 7.5× compared to standard ROS 2 communication. Adding

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 6.5: Performance measurements for the hardware-accelerated components of AutonomROS. The table shows min/max values collected in multiple measurements. Taken from [47].

hardware acceleration without a zero-copy communication middleware gives a minimal advantage. Second, using Iceoryx and hardware acceleration further reduces the CPU load, while the achieved FPS is bound by the camera's maximum frame 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 HMN, the CPU utilization of the component decreases significantly by a factor of 12.5×, and the turnaround time is about 1.6× 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 significantly reduces the CPU load and turnaround times.

The main conclusion of the measurements reported in Table 6.5 is that for intra-platform communication, using a zero-copy communication middleware such as Iceoryx is of utmost importance to maintain performance.

Moreover, adequate hardware acceleration relies on such a middleware. The reported measurements are only for two ROS 2 nodes. Mapping the overall AutonomROS unit of Figure 6.9 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.

6.4 CHAPTER CONCLUSION

In this chapter, we have presented three advanced application examples that make use of ReconROS. The ball-on-plate application uses a control algorithm to demonstrate the ability to implement distributed applications, which can be used beyond the robotics domain. ReconROS, therefore, represents an alternative to distributed FPGA applications such as [13, 18, 37]. Our implementation for the Turtlebot 3 platform realizes a real-world use case for ReconROS in combination with fpgaDDS and gateways. It shows that intra-FPGA communication can significantly accelerate the overall application. Finally, AutonomROS validates the compatibility of ReconROS to accelerated selected functionality in combination with state-of-the-art ROS 2 packages such as Nav2 and shows the benefits of zero-copy communication in hardware-accelerated applications.

7

CONCLUSION AND FUTURE WORK

Robotic computing on reconfigurable logic has become a research subject in recent years. Using reconfigurable hardware in robotics applications promises a faster and more energy-efficient computation for several applications compared to related computing platforms such as multi-core CPUs and GPUs. However, the standardized integration of hardware accelerators into ROS-based software applications and the preservation of the common ROS programming model is essential to this research area and for developers' acceptance. Therefore, this thesis presents ReconROS, a novel approach for integrating reconfigurable hardware into ROS-based robotics applications. The overall contribution comprises three main components:

- A new approach to integrating reconfigurable hardware into ROS-based applications was introduced with the ReconROS framework, a combination of ROS 2 and ReconOS. ReconROS enables the mapping of ROS 2 nodes either to software running on a (multi-core) CPU or entirely in the reconfigurable hardware. Compared to previously presented approaches, ReconROS firstly allows the implementation of ROS 2 nodes entirely in hardware and offers several advantages, such as a consistent programming model and more extensive support for ROS 2 communication paradigms such as services and actions. Additionally, the framework benefits from features inherited from ReconOS, such as hardware access to the virtual address area.

- While the ReconROS framework leverages static mapping to hardware, the concept of the ReconROS executor proposes hardware callbacks as a generalization of callbacks for event-driven programming and therefore enables the dynamic mapping of ROS 2 nodes to hardware. Hardware callbacks provide (sub-)functionality of ROS 2 nodes and are loaded into a reconfigurable slot after a specific event happens during runtime. Following this approach, the ReconROS executor acts as a scheduler and placer by collecting ready-to-execute callbacks to determine a schedule and assigning the callbacks to a reconfigurable slot. The executor abstracts scheduling and placement for the developer, making this technology accessible to more users.

- Due to the CPU-centric architecture of ROS, all hardware acceleration approaches for ROS suffer from limited data transmission bandwidths between the main memory and the accelerator. In order to tackle

this bottleneck, this thesis proposes fpgaDDS, an intra-FPGA data distribution service enabling the mapping of ROS 2 topics entirely to hardware. Due to the generation of a dedicated streaming network per ROS 2 topic in the FPGA, the communication bottleneck is reduced or entirely removed. As an extension, gateways increase the application area for hardware-mapped topics to ROS 2 topics with interacting nodes in hardware and software. The ROS programming model is also preserved here, such as using standardized message descriptions for communication.

For all three components, improvements in the form of runtime reductions were measured and presented in this thesis. Furthermore, three case studies demonstrate the applicability of this thesis and archive performance improvements using one or more proposed components.

However, ReconROS' broad approach to integrating reconfigurable hardware into ROS-based robotics architectures opens up additional research areas that can be explored in future work. The following list suggests some possible research directions:

- In this thesis, a methodology was presented that maps a ROS computation to the reconfigurable hardware. However, this approach already requires a ready mapping of ROS 2 nodes to software or hardware. This mapping problem could be solved autonomously in future work by solving ROS 2 nodes according to known methods such as exact ILP (integer linear programming) or even heuristic approaches. The user could give needed information about the nodes, or the tool flow determines the information automatically.

- Due to the consistent programming model provided by ReconROS, robotics developers can easily migrate ROS 2 nodes from hardware to software and vice versa. Dynamic task migration could exploit this property during runtime in future work.

- As shown in this thesis, reconfiguration overheads in a range of tens of milliseconds prevent the usage of hardware callbacks for several applications, e.g., control systems with cycle times in the same time range. In order to open up these areas of application, scheduling and placement would have to be much more sophisticated. Therefore, future work could take up work on improvements on the ReconROS executor to further reduce the reconfiguration overheads, e.g., by (speculative) pre-loading of hardware callbacks in the reconfigurable slots.

- fpgaDDS supports the mapping of ROS 2 topics with a fixed setting of quality-of-service parameters to hardware. However, since ROS 2 supports additional communication paradigms and a much larger set of QoS settings, future work could investigate how fpgaDDS could be extended for support of ROS 2 services and actions and additional QoS parameters.

- In this thesis, the ReconROS Executor and fpgaDDS have been presented as two separate contributions without intersection. Therefore, the combination, i.e., the use of hardware-mapped topics for hardware callbacks, would be a valuable extension of the presented approaches, enabling efficient usage of resources with simultaneous efficient communication.

BIBLIOGRAPHY

[1] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl. "ReconOS: An Operating System Approach for Reconfigurable Computing." In: *IEEE Micro* 34.1 (2014), pages 60–71. DOI: 10.1109/MM.2013.110.

[2] Andreas Agne, Marco Platzner, Christian Plessl, Markus Happe, and Enno Lübbers. "ReconOS." In: *FPGAs for Software Programmers*. Edited by Dirk Koch, Frank Hannig, and Daniel Ziener. Cham: Springer International Publishing, 2016, pages 227–244. ISBN: 978-3-319-26408-0. DOI: 10.1007/978-3-319-26408-0_13.

[3] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. "hthreads: a hardware/software co-designed multithreaded RTOS kernel." In: *2005 IEEE Conference on Emerging Technologies and Factory Automation*. Volume 2. 2005, 8 pp.–338. DOI: 10.1109/ETFA.2005.1612697.

[4] Bahar Asgari, Ramyad Hadidi, Nima Shoghi Ghaleshahi, and Hyesoon Kim. "PISCES: Power-Aware Implementation of SLAM by Customizing Efficient Sparse Algebra." In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pages 1–6. DOI: 10.1109/DAC18072.2020.9218550.

[5] Takuya Azumi, Yuya Maruyama, and Shinpei Kato. "ROS-lite: ROS Framework for NoC-Based Embedded Many-Core Platform." In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pages 4375–4382. DOI: 10.1109/IROS45743.2020.9340977.

[6] Mrinal R. Bachute and Javed M. Subhedar. "Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms." In: *Machine Learning with Applications* 6 (2021), page 100164. ISSN: 2666-8270. DOI: 10.1016/j.mlwa.2021.100164.

[7] Sinan Barut, Marco Boneberger, Pouya Mohammadi, and Jochen J. Steil. "Benchmarking Real-Time Capabilities of ROS 2 and OROCOS for Robotics Applications." In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pages 708–714. DOI: 10.1109/ICRA48506.2021.9561026.

[8] Christophe Bédard, Pierre-Yves Lajoie, Giovanni Beltrame, and Michel Dagenais. "Message Flow Analysis with Complex Causal Links for Distributed ROS 2 Systems." In: *Robot. Auton. Syst.* 161.C (2023). ISSN: 0921-8890. DOI: 10.1016/j.robot.2022.104361.

[9] Kaiwalya Belsare et al. "Micro-ROS." In: *Robot Operating System (ROS): The Complete Reference (Volume 7)*. Edited by Anis Koubaa. Cham: Springer International Publishing, 2023, pages 3–55. ISBN: 978-3-031-09062-2. DOI: 10.1007/978-3-031-09062-2.

[10] Tobias Betz, Maximilian Schmeller, Andreas Korb, and Johannes Betz. "Latency Measurement for Autonomous Driving Software Using Data Flow Extraction." In: *2023 IEEE Intelligent Vehicles Symposium (IV)*. 2023, pages 1–8. DOI: 10.1109/IV55152.2023.10186686.

[11] Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn B. Brandenburg. "Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems." In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2021, pages 264–277. DOI: 10.1109/RTAS52030.2021.00029.

[12] Tobias Blaß, Daniel Casini, Sergey Bozhko, and Björn B. Brandenburg. "A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance." In: *2021 IEEE Real-Time Systems Symposium (RTSS)*. 2021, pages 41–53. DOI: 10.1109/RTSS52674.2021.00016.

[13] Christophe Bobda, Kevin Cheng, Felix Mühlbauer, Klaus Drechsler, Jan Schulte, Dominik Murr, and Camel Tanougast. "Enabling self-organization in embedded systems with reconfigurable hardware." In: *International Journal of Reconfigurable Computing* (2009). DOI: 10.1155/2009/161458.

[14] Christian Brugger, Lorenzo Dal'Aqua, Javier Alejandro Varela, Christian De Schryver, Mohammadsadegh Sadri, Norbert Wehn, Martin Klein, and Michael Siegrist. "A quantitative cross-architecture study of morphological image processing on CPUs, GPUs, and FPGAs." In: *2015 IEEE Symposium on Computer Applications Industrial Electronics (ISCAIE)*. 2015, pages 201–206. DOI: 10.1109/ISCAIE.2015.7298356.

[15] *CameraInfo Message Specification*. http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/CameraInfo.html. Accessed: 2023-11-16.

[16] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, José M. M. Montiel, and Juan D. Tardós. "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM." In: *IEEE Transactions on Robotics* 37.6 (2021), pages 1874–1890. DOI: 10.1109/TRO.2021.3075644.

[17] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B. Brandenburg. "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling." In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Edited by Sophie Quinton. Volume 133. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019,

6:1–6:23. ISBN: 978-3-95977-110-8. DOI: 10.4230/LIPIcs.ECRTS.2019.6.

[18] Kevin Cheng, Ali Akbar Zarezadeh, Felix Muhlbauer, Camel Tanougast, and Christophe Bobda. "Auto-reconfiguration on self-organized intelligent platform." In: *2010 NASA/ESA Conference on Adaptive Hardware and Systems*. 2010, pages 309–316. DOI: 10.1109/AHS.2010.5546243.

[19] Hiroyuki Chishiro, Kazutoshi Suito, Tsutomu Ito, Seiya Maeda, Takuya Azumi, Kenji Funaoka, and Shinpei Kato. "Towards Heterogeneous Computing Platforms for Autonomous Driving." In: *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. 2019, pages 1–8. DOI: 10.1109/ICESS.2019.8782446.

[20] Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. "PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2." In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2021, pages 251–263. DOI: 10.1109/RTAS52030.2021.00028.

[21] Lennart Clausing and Marco Platzner. "ReconOS64: A Hardware Operating System for Modern Platform FPGAs with 64-Bit Support." In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pages 120–127. DOI: 10.1109/IPDPSW55747.2022.00029.

[22] *Colcon Project Website*. https://colcon.readthedocs.io/en/released/. Accessed: 2023-11-16.

[23] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. "An Introduction to High-Level Synthesis." In: *IEEE Design & Test of Computers* 26.4 (2009), pages 8–17. DOI: 10.1109/MDT.2009.69.

[24] Ürün Dogan, Johann Edelbrunner, and Ioannis Iossifidis. "Autonomous driving: A comparison of machine learning techniques by means of the prediction of lane change behavior." In: *2011 IEEE International Conference on Robotics and Biomimetics*. 2011, pages 1837–1843. DOI: 10.1109/ROBIO.2011.6181557.

[25] Marcel Eckert, Dominik Meyer, Jan Haase, Bernd Klauer, et al. "Operating system concepts for reconfigurable computing: review and survey." In: *International Journal of Reconfigurable Computing* 2016 (2016). DOI: doi.org/10.1155/2016/2478907.

[26] *Eclipse Cyclone DDS*. https://github.com/eclipse-cyclonedds/cyclonedds. Accessed: 2023-02-28.

[27] Marc Eisoldt, Marcel Flottmann, Julian Gaal, Steffen Hinderink, Juri Vana, Marco Tassemeier, Marc Rothmann, Thomas Wiemann, and Mario Porrmann. "ReconfROS: An approach for accelerating ROS nodes on reconfigurable SoCs." In: *Microprocessors and Microsystems* 94 (2022), page 104655. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2022.104655.

[28] Marc Eisoldt, Steffen Hinderink, Marco Tassemeier, Marcel Flottmann, Juri Vana, Thomas Wiemann, Julian Gaal, Marc Rothmann, and Mario Porrmann. "ReconfROS: Running ROS on Reconfigurable SoCs." In: *Proc. 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*. DroneSE and RAPIDO '21. Budapest, Hungary: Association for Computing Machinery, 2021, 16–21. ISBN: 9781450389525. DOI: 10.1145/3444950.3444959.

[29] J. Fernandez, B. Allen, P. Thulasiraman, and B. Bingham. "Performance Study of the Robot Operating System 2 with QoS and Cyber Security Settings." In: *2020 IEEE International Systems Conference (SysCon)*. 2020, pages 1–6. DOI: 10.1109/SysCon47679.2020.9275872.

[30] *FreeRTOS Website*. https://www.freertos.org/index.html. Accessed: 2023-11-16.

[31] *Gazebo Project Website*. https://gazebosim.org/home. Accessed: 2023-11-16.

[32] Pablo Ghiglino and Guillermo Sarabia. "The ring-buffer ROS2 executor: a novel approach for real-time ROS2 Space applications." In: *2023 IEEE Space Computing Conference (SCC)*. 2023, pages 80–85. DOI: 10.1109/SCC57168.2023.00021.

[33] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Pearson, 2018. ISBN: 9780133356724.

[34] *Harris Corner Detection*. https://de.mathworks.com/help/visionhdl/ug/corner-detection.html. Accessed: 2023-09-04.

[35] Chris Harris, Mike Stephens, et al. "A combined corner and edge detector." In: *Alvey vision conference*. Volume 15. 50. Citeseer. 1988, pages 10–5244. DOI: 10.5244/C.2.23.

[36] Kento Hasegawa, Kazunari Takasaki, Makoto Nishizawa, Ryota Ishikawa, Kazushi Kawamura, and Nozomu Togawa. "Implementation of a ROS-Based Autonomous Vehicle on an FPGA Board." In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pages 457–460. DOI: 10.1109/ICFPT47387.2019.00092.

[37] C. Haubelt, D. Koch, and J. Teich. "ReCoNet: modeling and implementation of fault tolerant distributed reconfigurable hardware." In: *Proc. 16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003*. 2003, pages 343–348. DOI: 10.1109/SBCCI.2003.1232851.

[38] Jeffrey Ichnowski et al. "FogROS2: An Adaptive Platform for Cloud and Fog Robotics Using ROS 2." In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, pages 5493–5500. DOI: 10.1109/ICRA48891.2023.10161307.

[39] Aws Ismail and Lesley Shannon. "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators." In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. 2011, pages 170–177. DOI: 10.1109/FCCM.2011.48.

[40] Martin Israel, Manuel Mende, and Stefan Keim. "UAVRC, a generic MAV flight assistance software." In: *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 40 (2015), pages 287–291. DOI: 10.5194/isprsarchives-XL-1-W4-287-2015.

[41] Xabier Iturbe, Khaled Benkrid, Ahmet T. Erdogan, Tughrul Arslan, Mikel Azkarate, Imanol Martinez, and Antonio Perez. "R3TOS: A reliable reconfigurable real-time operating system." In: *2010 NASA/ESA Conference on Adaptive Hardware and Systems*. 2010, pages 99–104. DOI: 10.1109/AHS.2010.5546274.

[42] D.E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Pearson Education, 1998. ISBN: 9780321635785.

[43] Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthé, Till Smejkal, and Gerhard Fettweis. "Latency Analysis of ROS2 Multi-Node Systems." In: *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. 2021, pages 1–7. DOI: 10.1109/MFI52462.2021.9591166.

[44] Takahisa Kuboichi, Atsushi Hasegawa, Bo Peng, Keita Miura, Kenji Funaoka, Shinpei Kato, and Takuya Azumi. "CARET: Chain-Aware ROS 2 Evaluation Tool." In: *2022 IEEE 20th International Conference on Embedded and Ubiquitous Computing (EUC)*. 2022, pages 1–8. DOI: 10.1109/EUC57774.2022.00010.

[45] Daniel Pinheiro Leal, Midori Sugaya, Hideharu Amano, and Takeshi Ohkawa. "Automated Integration of High-Level Synthesis FPGA Modules with ROS2 Systems." In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pages 292–293. DOI: ICFPT51103.2020.00052.

[46] Christian Lienen. "Implementing a Real-time System on a Platform FPGA operated with ReconOS." Master's thesis. September 2019.

[47] Christian Lienen, Mathis Brede, Daniel Karger, Kevin Koch, Dalisha Logan, Janet Mazur, Alexander Philipp Nowosad, Alexander Schnelle, Mohness Waizy, and Marco Platzner. *AutonomROS: A ReconROS-based Autonomonous Driving Unit*. 2023.

[48] Christian Lienen, Sorel Horst Middeke, and Marco Platzner. "FPGADDS: An Intra-FPGA Data Distribution Service for ROS 2 Robotics Applications." In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2023, pages 6261–6266. DOI: 10.1109/IROS55552.2023.10341921.

[49] Christian Lienen, Alexander Philipp Nowosad, and Marco Platzner. "Mapping and Optimizing Communication in ROS 2-based Applications on Configurable System-on-Chip Platforms." In: *Proceedings of the 9th International Conference on Robotics and Artificial Intelligence (ICRAI) (Accepted for Publication)*. 2023.

[50] Christian Lienen and Marco Platzner. "Design of Distributed Reconfigurable Robotics Systems with ReconROS." In: *ACM Transactions on Reconfigurable Technology and Systems* 15.3 (2022). ISSN: 1936-7406. DOI: 10.1145/3494571.

[51] Christian Lienen and Marco Platzner. "Event-Driven Programming of FPGA-accelerated ROS 2 Robotics Applications." In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. 2022, pages 615–623. DOI: 10.1109/DSD57027.2022.00088.

[52] Christian Lienen and Marco Platzner. "Task Mapping for Hardware-Accelerated Robotics Applications using ReconROS." In: *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*. 2022, pages 148–155. DOI: 10.1109/IRC55401.2022.00033.

[53] Christian Lienen, Marco Platzner, and Bernhard Rinner. "ReconROS: Flexible Hardware Acceleration for ROS2 Applications." In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pages 268–276. DOI: 10.1109/ICFPT51103.2020.00046.

[54] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. "Edge Computing for Autonomous Driving: Opportunities and Challenges." In: *Proceedings of the IEEE* 107.8 (2019), pages 1697–1716. DOI: 10.1109/JPROC.2019.2915983.

[55] Weizhuang Liu, Bo Yu, Yiming Gan, Qiang Liu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. "Archytas: A Framework for Synthesizing and Dynamically Optimizing Accelerators for Robotic Localization." In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. 2021, 479–493. ISBN: 9781450385572. DOI: 10.1145/3466752.3480077.

[56] Yanqi Liu, Can Eren Derman, Giuseppe Calderoni, and R. Iris Bahar. "Hardware Acceleration of Robot Scene Perception Algorithms." In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD '20. 2020. ISBN: 9781450380263. DOI: 10.1145/3400302.3415766.

[57] Enno Lübbers and Marco Platzner. "ReconOS: An RTOS Supporting Hard-and Software Threads." In: *2007 International Conference on Field Programmable Logic and Applications*. 2007, pages 441–446. DOI: 10.1109/FPL.2007.4380686.

[58] Enno Lübbers and Marco Platzner. "ReconOS: Multithreaded Programming for Reconfigurable Computers." In: *ACM Transactions on Embedded Computing Systems* 9.1 (2009), 8:1–8:33. DOI: 10.1145/1596532.1596540.

[59] Steve Macenski, Tom Moore, David V. Lu, Alexey Merzlyakov, and Michael Ferguson. "From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2." In: *Robotics and Autonomous Systems* 168 (2023), page 104493. ISSN: 0921-8890. DOI: 10.1016/j.robot.2023.104493.

[60] Steve Macenski, Alberto Soragna, Michael Carroll, and Zhenpeng Ge. "Impact of ROS 2 Node Composition in Robotic Systems." In: *IEEE Robotics and Automation Letters* 8.7 (2023), pages 3996–4003. DOI: 10.1109/LRA.2023.3279614.

[61] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild." In: *Science Robotics* 7.66 (2022). DOI: 10.1126/scirobotics.abm6074.

[62] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the Performance of ROS2." In: *Proceedings of the 13th International Conference on Embedded Software*. EMSOFT '16. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2016. ISBN: 9781450344852. DOI: 10.1145/2968478.2968502.

[63] Víctor Mayoral-Vilches. *Kria Robotics Stack*. https://www.xilinx.com/applications/industrial/robotics/wp540-kria-robotics-stack.html. Accessed: 2022-01-13. 2021.

[64] Víctor Mayoral-Vilches and Giulio Corradi. "Adaptive Computing in Robotics, Leveraging ROS 2 to Enable Software-Defined Hardware for FPGAs." In: *arXiv preprint arXiv:2109.03276* (2021). DOI: 10.48550/arXiv.2109.03276.

[65] Víctor Mayoral-Vilches, Sabrina M. Neuman, Brian Plancher, and Vijay Janapa Reddi. "RobotCore: An Open Architecture for Hardware Acceleration in ROS 2." In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2022, pages 9692–9699. DOI: 10.1109/IROS47612.2022.9982082.

[66] Sorel Horst Middeke. "Design and Realization of Optimized Intra-FPGA ROS 2 Communication." Master's thesis. Paderborn University, December 2022.

[67] Ryota Miyagi, Naofumi Takagi, Sho Kinoshista, Masashi Oda, and Hideki Takase. "Zytlebot : FPGA integrated ros-based autonomous mobile robot." In: *2021 International Conference on Field-Programmable Technology (ICFPT)*. 2021, pages 1–4. DOI: 10.1109/ICFPT52863.2021.9609883.

[68] Ren Morita and Katsuya Matsubara. "Dynamic Binding a Proper DDS Implementation for Optimizing Inter-Node Communication in ROS2." In: *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2018, pages 246–247. DOI: 10.1109/RTCSA.2018.00043.

[69] Raúl Mur-Artal and Juan D. Tardós. "ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras." In: *IEEE Transactions on Robotics* 33.5 (2017), pages 1255–1262. DOI: 10.1109/TRO.2017.2705103.

[70] Sean Murray, Will Floyd-Jones, George Konidaris, and Daniel J. Sorin. "A Programmable Architecture for Robot Motion Planning Acceleration." In: *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Volume 2160-052X. 2019, pages 185–188. DOI: 10.1109/ASAP.2019.000-4.

[71] Sean Murray, Will Floyd-Jones, Ying Qi, Daniel J Sorin, and George Dimitri Konidaris. "Robot Motion Planning on a Chip." In: *Robotics: Science and Systems*. Volume 6. 2016. DOI: 10.15607/rss.2016.xii.004.

[72] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J. Sorin. "The microarchitecture of a real-time robot motion planning accelerator." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pages 1–12. DOI: 10.1109/MICRO.2016.7783748.

[73] Yasuhiro Nitta, Sou Tamura, and Hideki Takase. "A Study on Introducing FPGA to ROS Based Autonomous Driving System." In: *2018 International Conference on Field-Programmable Technology (FPT)*. 2018, pages 421–424. DOI: 10.1109/FPT.2018.00090.

[74] Yasuhiro Nitta, Sou Tamura, Hidetoshi Yugen, and Hideki Takase. "ZytleBot: FPGA Integrated Development Platform for ROS Based Autonomous Mobile Robot." In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pages 445–448. DOI: 10.1109/ICFPT47387.2019.00089.

[75] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, and Debbie Marr. "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC." In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pages 1–4. DOI: 10.1109/FPL.2016.7577314.

[76] Frederik Falk Nyboe, Nicolaj Haarhøj Malle, and Emad Ebeid. "MPSoC4Drones: An Open Framework for ROS2, PX4, and FPGA Integration." In: *2022 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2022, pages 1246–1255. DOI: 10.1109/ICUAS54217.2022.9836055.

[77] 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: *2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*. 2019, pages 33–36. DOI: 10.1109/RoSE.2019.00014.

[78] Takeshi Ohkawa, Kazushi Yamashina, Hitomi Kimura, Kanemitsu Ootsu, and Takashi Yokota. "FPGA components for integrating FPGAs into robot systems." In: *IEICE TRANSACTIONS on Information and Systems* 101.2 (2018), pages 363–375. DOI: 10.1587/transinf.2017RCP0011.

[79] Takeshi Ohkawa, Kazushi Yamashina, Takuya Matsumoto, Kanemitsu Ootsu, and Takashi Yokota. "Architecture Exploration of Intelligent Robot System Using ROS-Compliant FPGA Component." In: *Proceedings of the 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*. RSP '16. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2016, 72–78. ISBN: 9781450345354. DOI: 10.1145/2990299.2990312.

[80] Takeshi Ohkawa, Kazushi Yamashina, Takuya Matsumoto, Kanemitsu Ootsu, and Takashi Yokota. "Automatic generation tool of FPGA components for robots." In: *IEICE transactions on Information and Systems* 102.5 (2019), pages 1012–1019. DOI: 10.1587/transinf.2018RCP0004.

[81] *OpenCV Project Website*. https://opencv.org/. Accessed: 2023-11-16.

[82] *PYNQ Project Repository*. https://github.com/Xilinx/PYNQ. Accessed: 2023-11-16.

[83] Bo Peng, Atsushi Hasegawa, and Takuya Azumi. "Scheduling Performance Evaluation Framework for ROS 2 Applications." In: *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. 2022, pages 2031–2038. DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00302.

[84] Brian Plancher and Scott Kuindersma. "A performance analysis of parallel differential dynamic programming on a gpu." In: *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*. Springer. 2020, pages 656–672. DOI: 10.1007/978-3-030-44051-0_38.

[85] Brian Plancher, Sabrina M. Neuman, Thomas Bourgeat, Scott Kuindersma, Srinivas Devadas, and Vijay Janapa Reddi. "Accelerating Robot Dynamics Gradients on a CPU, GPU, and FPGA." In: *IEEE Robotics and Automation Letters* 6.2 (2021), pages 2335–2342. DOI: 10.1109/LRA.2021.3057845.

[86] Brian Plancher, Sabrina M. Neuman, Radhika Ghosal, Scott Kuindersma, and Vijay Janapa Reddi. "GRiD: GPU-Accelerated Rigid Body Dynamics with Analytical Gradients." In: *2022 International Conference on Robotics and Automation (ICRA)*. 2022, pages 6253–6260. DOI: 10.1109/ICRA46639.2022.9812384.

[87] A. Podlubne and D. Göhringer. "FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs." In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2019. DOI: 10.1109/ReConFig48160.2019.8994719.

[88] Ariel Podlubne and Diana Göhringer. "A Survey on Adaptive Computing in Robotics: Modelling, Methods and Applications." In: *IEEE Access* 11 (2023), pages 53830–53849. DOI: 10.1109/ACCESS.2023.3281190.

[89] Ariel Podlubne, Johannes Mey, René Schöne, Uwe Aßmann, and Diana Göhringer. "Model-Based Approach for Automatic Generation of Hardware Architectures for Robotics." In: *IEEE Access* 9 (2021), pages 140921–140937. DOI: 10.1109/ACCESS.2021.3119061.

[90] L. Puck, P. Keller, T. Schnell, C. Plasberg, A. Tanev, G. Heppner, A. Roennau, and R. Dillmann. "Distributed and Synchronized Setup towards Real-Time Robotic Control using ROS2 on Linux." In: *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*. 2020, pages 1287–1293. DOI: 10.1109/CASE48305.2020.9217010.

[91] L. Puck, P. Keller, T. Schnell, C. Plasberg, A. Tanev, G. Heppner, A. Roennau, and R. Dillmann. "Performance Evaluation of Real-Time ROS2 Robotic Control in a Time-Synchronized Distributed Network." In: *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*. 2021, pages 1670–1676. DOI: 10.1109/CASE49439.2021.9551447.

[92] *QEMU Project Website*. https://www.qemu.org/. Accessed: 2023-11-16.

[93] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip 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)*. 2019, pages 1–8. DOI: 10.1109/ICESS.2019.8782524.

[94] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. "ROS: an open-source Robot Operating System." In: *ICRA workshop on open source software*. Volume 3. 3.2. Kobe, Japan. 2009, page 5.

[95] *RISC-V Website*. https://riscv.org/. Accessed: 2023-11-16.

[96] *ROS 2 Quality-of-Service Settings Documentation*. https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Quality-of-Service-Settings.html. Accessed: 2023-11-16.

[97] *ROS Hardware Acceleration Working Group*. https://github.com/ros-acceleration. Accessed: 2023-11-16.

[98] *ROS Project Website*. https://www.ros.org. Accessed: 2023-11-16.

[99] *ReconOS Project Website*. http://www.reconos.de/. Accessed: 2023-11-16.

[100] Christina Rödel, Susanne Stadler, Alexander Meschtscherjakov, and Manfred Tscheligi. "Towards Autonomous Cars: The Effect of Autonomy Levels on Acceptance and User Experience." In: *Proceedings of the 6th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. AutomotiveUI '14. Seattle, WA, USA: Association for Computing Machinery, 2014, 1–8. ISBN: 9781450332125. DOI: 10.1145/2667317.2667330.

[101] Edward Rosten and Tom Drummond. "Machine Learning for High-Speed Corner Detection." In: *Proceedings of European Conference on Computer Vision*. Edited by Aleš Leonardis, Horst Bischof, and Axel Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pages 430–443. ISBN: 978-3-540-33833-8. DOI: 10.1007/11744023_34.

[102] Christoph Rüthing. "Self-Adaptation in Programmable Automation Controllers based on Hybrid Multi-Cores." Master's thesis. Paderborn University, August 2015.

[103] Jacob Sacks, Divya Mahajan, Richard C. Lawson, Behnam Khaleghi, and Hadi Esmaeilzadeh. "RoboX: An End-to-End Solution to Accelerate Autonomous Control in Robotics." In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pages 479–490. DOI: 10.1109/ISCA.2018.00047.

[104] Yukihiro Saito, Futoshi Sato, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. "ROSCH:Real-Time Scheduling Framework for ROS." In: *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2018, pages 52–58. DOI: 10.1109/RTCSA.2018.00015.

[105] Ashutosh Singandhupe and Hung Manh La. "A Review of SLAM Techniques and Security in Autonomous Driving." In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 2019, pages 602–607. DOI: 10.1109/IRC.2019.00122.

[106] Jan Staschulat, Ingo Lütkebohle, and Ralph Lange. "The rclc Executor: Domain-specific deterministic scheduling mechanisms for ROS applications on microcontrollers: work-in-progress." In: *2020 International Conference on Embedded Software (EMSOFT)*. 2020, pages 18–19. DOI: 10.1109/EMSOFT51651.2020.9244014.

[107] D. Stewart. "A Platform with Six Degrees of Freedom." In: *Proceedings of the Institution of Mechanical Engineers*. Volume 180. 1. 1965, pages 371–386. DOI: 10.1243/PIME_PROC_1965_180_029_02.

[108] B. Strohmer, A. BØgild, A. S. SØrensen, and L. B. Larsen. "ROS-Enabled Hardware Framework for Experimental Robotics." In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2019. DOI: 10.1109/ReConFig48160.2019.8994770.

[109] Yuhei Sugata, Takeshi Ohkawa, Kanemitsu Ootsu, and Takashi Yokota. "Acceleration of Publish/Subscribe Messaging in ROS-Compliant FPGA Component." In: *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART2017)*. Bochum, Germany: ACM, 2017. ISBN: 9781450353168. DOI: 10.1145/3120895.3120904.

[110] Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. "Navion: A 2-mW Fully Integrated Real-Time Visual-Inertial Odometry Accelerator for Autonomous Navigation

of Nano Drones." In: *IEEE Journal of Solid-State Circuits* 54.4 (2019), pages 1106–1119. DOI: 10.1109/JSSC.2018.2886342.

[111] Yuhei Suzuki, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. "Real-Time ROS Extension on Transparent CPU/GPU Coordination Mechanism." In: *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. 2018, pages 184–192. DOI: 10.1109/ISORC.2018.00035.

[112] Hideki Takase, Tomoya Mori, Kazuyoshi Takagi, and Naofumi Takagi. "MROS: A Lightweight Runtime Environment for Robot Software Components onto Embedded Devices." In: *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. HEART '19. Nagasaki, Japan: Association for Computing Machinery, 2019. ISBN: 9781450372558. DOI: 10.1145/3337801.3337815.

[113] Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. "Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors." In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. 2020, pages 231–243. DOI: 10.1109/RTSS49844.2020.00030.

[114] Russell Tessier, Kenneth Pocek, and André DeHon. "Reconfigurable Computing Architectures." In: *Proceedings of the IEEE* 103.3 (2015), pages 332–354. DOI: 10.1109/JPROC.2014.2386883.

[115] P. Thulasiraman, Z. Chen, B. Allen, and B. Bingham. "Evaluation of the Robot Operating System 2 in Lossy Unmanned Networks." In: *2020 IEEE International Systems Conference (SysCon)*. 2020, pages 1–8. DOI: 10.1109/SysCon47679.2020.9275849.

[116] *Turtlebot 3 Autonomous Driving Challenge*. https://emanual.robotis.com/docs/en/platform/turtlebot3/autonomous_driving/. Accessed: 2023-11-16.

[117] *Turtlebot 3 Autorace 2020 Challenge*. https://github.com/ROBOTIS-GIT/turtlebot3_autorace_2020. Accessed: 2023-11-16.

[118] Tomohisa Uchida. "Hardware-Based TCP Processor for Gigabit Ethernet." In: *IEEE Transactions on Nuclear Science* 55.3 (2008), pages 1631–1637. DOI: 10.1109/NSSMIC.2007.4436337.

[119] Onur Ulusel, Christopher Picardo, Christopher B. Harris, Sherief Reda, and R. Iris Bahar. "Hardware acceleration of feature detection and description algorithms on low-power embedded platforms." In: *Proceedings 2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pages 1–9. DOI: 10.1109/FPL.2016.7577310.

[120] Stylianos I. Venieris and Christos-Savvas Bouganis. "fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs." In: *IEEE Transactions on Neural Networks and Learning Systems* 30.2 (2019), pages 326–342. DOI: 10.1109/TNNLS.2018.2844093.

[121] Kizheppatt Vipin and Suhaib A Fahmy. "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq." In: *IEEE Embedded Systems Letters* 6.3 (2014), pages 41–44. DOI: 10.1109/LES.2014.2314390.

[122] *Vitis HLS Library*. https://github.com/Xilinx/Vitis_Libraries.git. Accessed: 2023-11-16.

[123] Zishen Wan, Ashwin Lele, Bo Yu, Shaoshan Liu, Yu Wang, Vijay Janapa Reddi, Cong Hao, and Arijit Raychowdhury. "Robotic Computing on FPGAs: Current Progress, Research Challenges, and Opportunities." In: *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 2022, pages 291–295. DOI: 10.1109/AICAS54282.2022.9869951.

[124] Ying Wang, Xuegong Zhou, Lingli Wang, Jian Yan, Wayne Luk, Chenglian Peng, and Jiarong Tong. "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12 (2013), pages 2179–2192. DOI: 10.1109/TVLSI.2012.2231101.

[125] Hongxing Wei, Zhenzhou Shao, Zhen Huang, Renhai Chen, Yong Guan, Jindong Tan, and Zili Shao. "RT-ROS: A real-time ROS architecture on multi-core processors." In: *Future Generation Computer Systems* 56 (2016), pages 171–178. ISSN: 0167-739X. DOI: 10.1016/j.future.2015.05.008.

[126] Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. "Monocular-SLAM-based navigation for autonomous micro helicopters in GPS-denied environments." In: *Journal of Field Robotics* 28.6 (2011), pages 854–874. DOI: 10.1002/rob.20412.

[127] Richard Welch, Daniel Limonadi, and Robert Manning. "Systems engineering the Curiosity Rover: A retrospective." In: *2013 8th International Conference on System of Systems Engineering*. 2013, pages 70–75. DOI: 10.1109/SYSoSE.2013.6575245.

[128] *XRT Project Repository*. https://github.com/Xilinx/XRT. Accessed: 2023-11-16.

[129] *Xillinux Project Website*. https://xillybus.com/xillinux. Accessed: 2023-11-16.

[130] Kazushi Yamashina, Hitomi Kimura, Takeshi Ohkawa, Kanemitsu Ootsu, and Takashi Yokota. "CReComp: Automated Design Tool for ROS-Compliant FPGA Component." In: *IEEE 10th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2016*. IEEE, 2016, pages 138–145. ISBN: 9781509035304. DOI: 10.1109/MCSoC.2016.47.

[131] Kazushi Yamashina, Takeshi Ohkawa, Kanemitsu Ootsu, and Takashi Yokota. *Proposal of ROS-compliant FPGA Component for Low-Power Robotic Systems*. 2015. DOI: 10.48550/arXiv.1508.07123. arXiv: 1508.07123 [cs.AR].

[132] Yuqing Yang and Takuya Azumi. "Exploring Real-Time Executor on ROS 2." In: *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*. 2020, pages 1–8. DOI: 10.1109/ICESS49830.2020.9301530.

[133] Evşen Yanmaz, Saeed Yahyanejad, Bernhard Rinner, Hermann Hellwagner, and Christian Bettstetter. "Drone networks: Communications, coordination, and sensing." In: *Ad Hoc Networks* 68 (2018), pages 1–15. DOI: 10.1016/j.adhoc.2017.09.001.

[134] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. "A Survey of Autonomous Driving: Common Practices and Emerging Technologies." In: *IEEE Access* 8 (2020), pages 58443–58469. DOI: 10.1109/ACCESS.2020.2983149.

[135] *Zephyr Project Website*. https://zephyrproject.org/. Accessed: 2023-11-16.

[136] *eCOS Project Website*. http://ecos.sourceware.org/. Accessed: 2023-11-16.

[137] *eProsima Fast DDS*. https://github.com/eProsima/Fast-DDS. Accessed: 2023-02-28.

[138] *iceoryx - true zero-copy inter-process-communication*. https://github.com/eclipse-iceoryx/iceoryx. Accessed: 2023-02-28.

[139] *mros2 Project Repository*. https://github.com/mROS-base/mros2. Accessed: 2023-11-16.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/