RECONROS: Flexible Hardware Acceleration for ROS2 Applications

Christian Lienen
Department of Computer Science
Paderborn University
Germany
christian.lienen@upb.de

Marco Platzner
Department of Computer Science
Paderborn University
Germany
platzner@upb.de

Bernhard Rinner
Institute of Networked and Embedded Systems
University of Klagenfurt
Austria
bernhard.rinner@aau.at

***Abstract*—Mobile robotics applications process large amounts of data in real-time and require compute platforms that provide high performance and energy-efficiency. FPGAs are well-suited for many of these applications, but there is a reluctance in the robotics community to use hardware acceleration due to increased design complexity and a lack of consistent programming models across the software/hardware boundary. In this paper we present RECONROS, a framework that integrates the widely-used robot operating system (ROS) with ReconOS, which features multithreaded programming of hardware and software threads for reconfigurable computers. This unique combination gives ROS2 developers the flexibility to transparently accelerate parts of their robotics applications in hardware. We elaborate on the architecture and the design flow for RECONROS and report on a set of experiments that underline the feasibility and flexibility of our approach.**

***Index Terms*—Robotics, FPGA, ROS2, ReconOS**

I. INTRODUCTION

Mobile robotics involves challenging tasks, especially when drones or autonomous vehicles move in unstructured and GPS-denied environments. In such scenarios, estimating the local position and orientation (pose) is a fundamental task. Data from different onboard sensors must be analyzed and fused over time to achieve a robust pose estimation. Cameras are widely used on mobile robots, and advanced computer vision algorithms such as visual simultaneous localization and mapping (SLAM) represent a key component for the pose estimation [1]. Collaborative state estimation techniques are deployed more recently where data from other (nearby) robots is exploited to improve accuracy, and robustness [2].

Resource-efficiency is a fundamental challenge of all these techniques since large amounts of data must be processed in real-time [3]. Compared to implementations on CPUs and GPUs, FPGAs have been shown to offer higher performance and higher energy-efficiency for many of the involved tasks, e.g., for vision kernels [4], for morphological image processing functions [5], for feature detection and description algorithms [6], and for convolutional neural network inference [7]. However, despite the demonstrated advantages of FPGAs, their proliferation into the robotics domain is still limited for several reasons. On one hand, FPGA design and, all the more, software/hardware co-design are arguably more challenging than embedded software development. On the other hand, robotics engineers and application developers are typically not trained in FPGA circuit or hardware/software co-design.

High level synthesis (HLS) tools that use standard C/C++ for describing behavior and (semi-)automatically take such descriptions to FPGA hardware increase productivity and are thus highly useful, but a consistent programming model for implementing software and hardware functions 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 very often leads to a re-development of substantial parts of the application.

In our work, we take up a very popular programming environment in the robotics domain, the robot operating system (ROS). ROS is a middleware layer that models applications as set of communicating nodes. We present RECONROS as a novel integration of ROS with ReconOS [8]. ReconOS provides an architecture and programming model to enable shared memory multi-threading for software and hardware threads. 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.

The remainder of the paper is organized as follows: Section II provides an overview over ROS and related approaches for integrating hardware accelerators into ROS. Section III elaborates on different approaches for accelerating ROS applications, before Section IV details RECONROS with its architecture and design flow. In Section V, we present experiments to demonstrate the feasibility and flexibility of RECONROS. Finally, Section VI concludes the paper and gives an outlook to future work.

II. BACKGROUND AND RELATED WORK

In this section, we first briefly introduce the robot operating system (ROS) and then analyze and compare related approaches for integrating FPGA hardware acceleration into ROS.

A. The Robot Operating System (ROS)

The Robot Operating System (ROS)[1] is an open-source middleware on top of Linux for robotics applications that was originally developed by William Garage and is now coordinated by the Open Robotic Foundation. ROS comprises a multitude of libraries and an infrastructure for building and reusing robot-related software modules. The ROS programming paradigm splits larger software architectures into nodes, which use certain communication mechanisms for information exchange.

The decomposition into nodes promises code reusability and modularity for robot architectures. Available communication mechanisms comprise (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, and (iii) actions, which make use of services but here the client can receive regular feedback about the server's progress.

ROS2 is the latest release of ROS. In earlier versions only one ROS node per Linux process was supported. This prevented the use of shared memory communication when both ROS nodes are mapped to the same compute node. While this limitation was mitigated through the support of so-called nodelets, with ROS2 multiple ROS nodes can natively run within one Linux process and there is support for shared memory communication. ROS2 is built on top of an exchangeable communication layer, the data distribution service (DDS). DDS is an industry standard for decentralized communication and available from different vendors. Compared to older ROS versions, the use of DDS provides better configurability and improves properties such as scalability, reliability and durability [9].

Another important element of ROS are ROS messages, which are multi-layered combinations of built-in data types such as integers, floats and strings. Besides predefined message types, e.g., for images or 3D point clouds, custom messages can be created. Since the length of a message might vary during runtime, the ROS2 middleware supports dynamic memory allocation for messages.

B. Related Approaches for ROS-FPGA Integration

In the last years, a few approaches have been presented that integrate reconfigurable hardware accelerators into a ROS software architecture. Yamashina et al. [10] proposed so-called ROS-compliant FPGA components. A ROS node is implemented in software and accesses the hardware component, i.e., the accelerator, via a software wrapper. Communication within the ROS network is completely 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 remote procedure call, realized in Xilinux. In [11], the automated design tool cReComp (creator for 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 eReComp, 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.

In follow-up work, Sugata et al. [12] identify the communication times between ROS nodes as bottlenecks and aim to reduce these times through implementing the ROS publish/subscribe messaging in hardware. In their system, communication is divided into two phases: the connection establish phase, which is supported by software, and the data communication phase that is realized by two network stacks implemented in FPGA hardware. This reduces the communication time between nodes by 50 percent. Ohkawa et al. [13] extend this work by using high level synthesis (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.

While [12], [13] migrate almost a complete ROS node to hardware, Podlubne and Göhringer [14] go one step further and propose a methodology for full-hardware implementation of a number of ROS nodes. Their hardware designs comprise four parts: the ROS application nodes that use publish/subscribe communication, a so-called application-to-ROS converter, a communication interface, and a manager. Basically, the application-to-ROS converter serializes the ROS-based IP traffic on an AXI bus, the communication interface handles the AXI messages and sends them to a TCP/IP stack to connect to external ROS nodes, and the manager coordinates the communication between the ROS nodes and the TCP/IP stack. Conceptually, the application-to-ROS converter must reside in hardware, but the communication interface and the manger could also be mapped to the processing system of the platform FPGA. However, the main feature of this methodology is the option to implement one or more ROS nodes fully in hardware and map them to reconfigurable logic without the need of using a processor. Likewise, any application implemented in reconfigurable hardware can be made ROS-compatible. Furthermore, the presented implementation can use dynamic custom ROS messages.

Strohmer et al. [15] presented a ROS-enabled hardware framework for experimental robotics. They use the programmable logic on a Xilinx Zynq-7000 for signal conditioning and partition the available CPU cores into a non real-time part running Linux with ROS and a real-time part running control algorithms. A distributed network of FPGAs can extend the

[1]https://www.ros.org

signal conditioning part using TosNet, which provides memory access across multiple nodes by memory mirroring.

III. Design Considerations

The goal of this work is provide developers of ROS2-based robotic applications with a flexible means to utilize programmable logic for hardware acceleration. On the level of ROS2 applications, there are several schemes for such an integration, which are sketched in Figure 1. Figure 1(a) shows a scheme where some parts of a ROS2 node, typically runtime-consuming functions, are mapped to one or several accelerators in programmable logic. The semantics of the communication between the ROS2 node and the accelerators is that of a remote procedure call (RPC). In Figure 1(b), a hardware accelerator is shared between several ROS2 nodes. Communication semantics is still RPC, but the implementation is more involved since proper arbitration between the accesses of the ROS2 nodes is required. The third scheme shown in Figure 1(c) is the most advanced and allows to map complete ROS2 nodes to hardware. Essentially, the hardware accelerator is turned into a ROS2 node. In this scheme, all ROS2 nodes can communicate via the ROS2 publish/subscribe mechanism, 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 ROS2 nodes use exactly the same ROS2 functions. Furthermore, arbitrary combinations of the schemes are also conceivable.

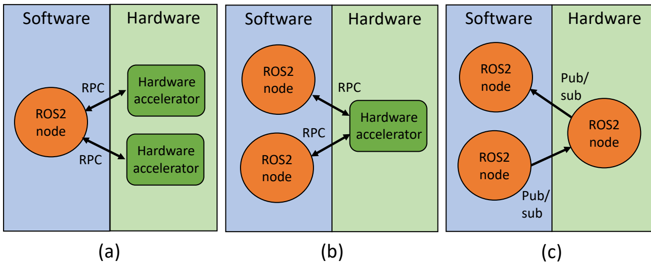

Fig. 1: Different schemes for integrating ROS2 node with hardware accelerators

Naturally, ReconROS allows developers to implement accelerators and ROS2 nodes in hardware that directly connect to peripherals such as A/D converters or non-USB cameras in reconfigurable logic or directly connect several ROS2 nodes, all without any software interaction. While such schemes are sometimes followed to maximize performance in concrete robotics applications, there are also two possible counter-arguments: First, flexibility is reduced since directly connected peripherals can not be accessed by other ROS2 nodes, and much less so when the ROS2 nodes are mapped to different compute nodes in a distributed system. Second, many sensors and actors come with standardized interfaces and corresponding drivers, e.g., USB, for which the use of an existing, software-accessible peripheral of the compute platform is much easier than to implement suitable interfaces and protocol stacks in hardware.

Characteristic	[10], [11], [12], [13]	[14]	[15]	ReconROS
ROS version	1	1	1	2
Support of hw/sw co-designed ROS nodes	✓	✗	✓	✓
Multiple ROS nodes per FPGA	✗	✓	✗	✓
Consistent hw/sw programming model	✗	✗	✗	✓
Memory access for hw accelerators	✗	✗	✗	✓
Support of arbitrarily large ROS messages	✗	✗	✗	✓

TABLE I: Comparison of approaches for integrating hardware accelerators with ROS

ReconROS[2] integrates the ROS2 middleware with the ReconOS/Linux architecture and programming model for hardware/software multithreading on platform FPGAs and can realize all schemes shown in Figure 1 and their combinations. On 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, ROS2 allows for declaring several ROS2 nodes within one Linux process. Therefore, in the first and second scheme of Figure 1, each hardware accelerator is encapsulated by a ReconOS hardware thread. In contrast to related work, ReconROS hardware accelerators can communicate with the ROS2 software nodes not only by passing data in an RPC manner, but can also use shared memory communication in the Linux virtual address space, which is more efficient when larger data structures have to be passed. In such a case, pointers to arbitrarily large ROS2 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 scheme two is straight-forward to realize. In the most advanced scheme shown in Figure 1(c), ReconOS hardware threads implement complete ROS2 nodes and allow them to access operating system functions and also ROS2 communication primitives, using the whole set of standard and even custom-defined ROS messages.

Table I compares ReconROS with related approaches. In contrast to all other approaches, ReconROS leverages the more future-oriented ROS2 version which promises improved scalability and real-time properties. Hardware acceleration of a ROS node mostly implies to partition the node and implement it as hardware/software co-design. This is followed by all approaches except [14], which maps complete ROS nodes to hardware. Mapping several ROS nodes to hardware is possible in [14] and ReconROS. A consistent hardware/software programming model, the memory access for hardware accelerators, and the support of arbitrarily large ROS messages are unique features of ReconROS.

[2]https://github.com/Lien182/ReconROS

IV. RECONROS

In this section, we present the architecture of RECONROS, followed by the design flow and an example that shows the programming interface.

A. Hardware/Software Architecture

RECONROS inherits most of its hardware architecture from the underlying ReconOS [8], [16]. Figure 2 shows an example architecture with two hardware ROS2 nodes (threads) and several software ROS2 nodes (threads). The hardware threads are mapped to reconfigurable slots and are connected to the Linux operating system kernel running on the CPU via the operating system interface (OSIF) and to shared memory via the memory interface (MEMIF). A so-called operating system finite state machine (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 light-weight 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 an 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. The RECONROS stack extends the existing set of ReconOS objects, such as semaphores or mailboxes, with the four new objects *rosnode*, *rossub*, *rospub*, and *rosmsg*. These objects relate to ROS2 nodes, ROS2 publisher, ROS2 subscriber, and ROS2 messages, and can be created when configuring a ROS2 application.

The RECONROS API abstracts the standard ROS2 API and allows ReconOS threads to access the objects of the RECONROS stack. As indicated in Figure 2, the RECONROS API is available for both software and hardware threads and currently 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. Software threads can not only access the RECONROS API but also the standard ROS2 API to utilize a richer set of functions. For hardware threads, the set of three functions implemented in the RECONROS API is sufficient to implement ROS2 hardware nodes that receive data, process it, and send it back. However, due to the flexibility of the underlying ReconOS system any ROS2 function can be made available for hardware threads.

In contrast to related work, our ROS2 hardware nodes can access shared memory and thus implement 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 is 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 interface provides higher data rates due to the used AXI high performance interface of the processing system. Second, the transmission of the data can be done without using the processing system, which leads to more potential for parallel execution of software and hardware threads.

Figure 3 exemplifies the sequence of events when a hardware ROS2 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 ROS2 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 unblocks the delegate thread ④, which in turn sends the message pointer via the OSIF back to the hardware thread ⑤. 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 stores the message in the main memory. Then, it sends a `ROS_publish` command and the message pointer via the OSIF interface to its delegate thread, which executes the command.

B. Design Flow

Designing a RECONROS application starts with creating the RECONROS configuration file and the implementations for the software and hardware threads. The configuration file specifies the ROS2 objects and their dependencies as well as the implementation architecture including the number of reconfigurable slots, used RECONROS primitives, threads, mapping of threads to slots, and settings for the toolchain.

The basic element of each RECONROS application is the *rosnode* object, which represents a ROS2 node in the network. A ROS2 node is equipped with one or more subscriber (*rossub*) and / or one or more publisher (*rospub*) objects attached to specific topics. In addition, each *rossub* / *rospub* gets a reference to an instance of a ROS message *rosmsg* of a specific type. Declarations of *rosmsg* objects are specified by their group, e.g., sensor_msgs, and type, e.g., Image.

The actual tool flow for a RECONROS project extends the original ReconOS tool flow [8]. Based on the configuration file, the ReconOS development kit creates a hardware project and a software project. The hardware project contains the RECONROS infrastructure, comprising, among others, the OSIFs, the MEMIFs, and the hardware threads. The software project includes the delegate threads, all necessary initialization functions for the ReconOS primitives but also the ROS2 middleware. In the following build step, the software binaries and the FPGA configuration bitstream are generated.

We provide the same RECONROS API for software threads written in C and for hardware threads described in VHDL or C/C++ for high-level synthesis. For hardware threads created

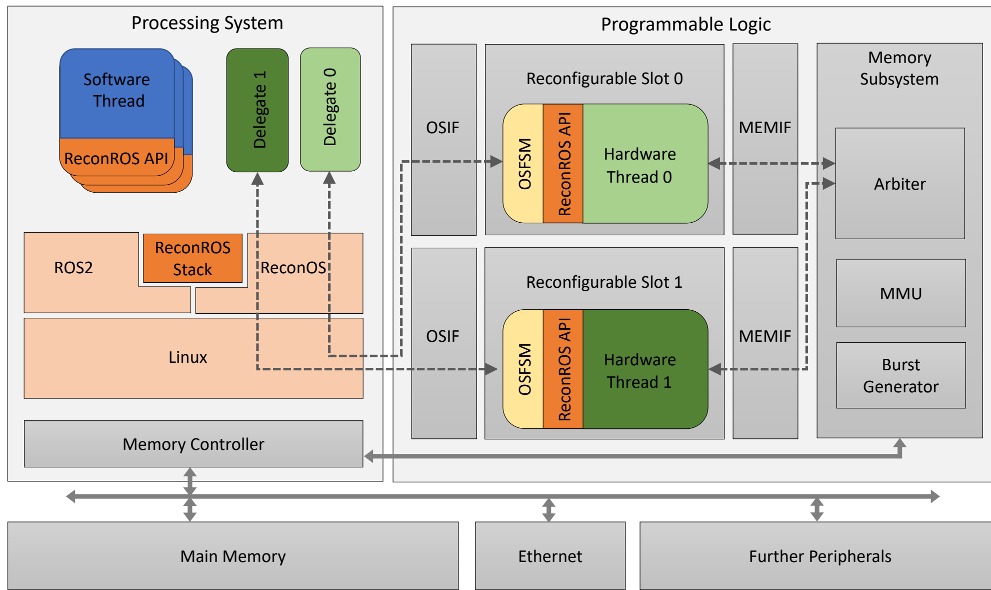

Fig. 2: RECONROS architecture with two hardware ROS2 nodes (threads) and several software ROS2 nodes (threads)

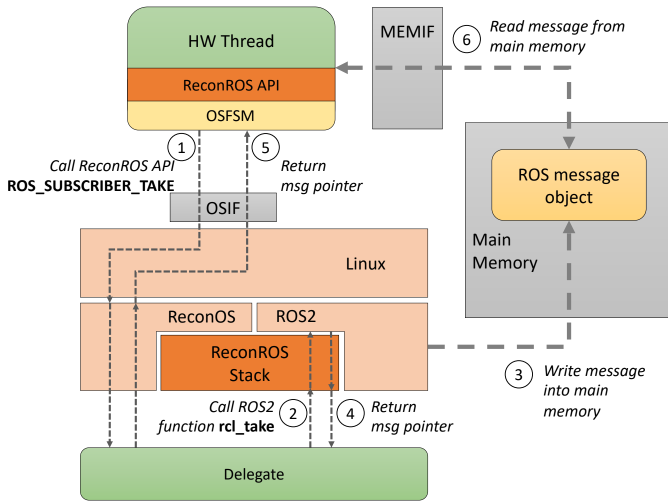

Fig. 3: Sequence of events when a ROS2 hardware node calls the *ROS_SUBSCRIBER_TAKE* function from the RECONROS API

with HLS, we pass the ROS message layout information stored in ROS message header files on to the HLS translation process. This allows for a straight-forward way of writing hardware threads that can access only relevant parts of a ROS message through their MEMIFs, without having to load whole messages into the FPGA.

C. Example ROS2 Application

Figure 4 outlines a ROS2 application comprising three nodes. The first node captures images from a camera and publishes them to the topic `/image_raw`. The second node subscribes to this topic, reads and filters the images, and publishes the result to the topic `/image_filtered`. The third node reads and displays the filtered images.

Generally, the ROS2 application can run either on a single platform FPGA or on multiple platform FPGAs in a distributed ROS2 setup. Independently of the setup, the filter node of this application is to be a Sobel filter described in HLS and mapped to hardware. Thus, the RECONROS configuration file must specify the needed primitives. Listing 1 shows the ROS-related part of the configuration file for this node. The first line of the listing defines the message type as `Image`. Line two instantiates the ROS node `node_2` with the name `"filter"`. The following two lines equip the filter node with a publisher and a subscriber object. Each of them specifies a message type and the corresponding topic. Additionally, the subscriber object includes a sleep time (10000 μs) for which the node will wait for new messages.

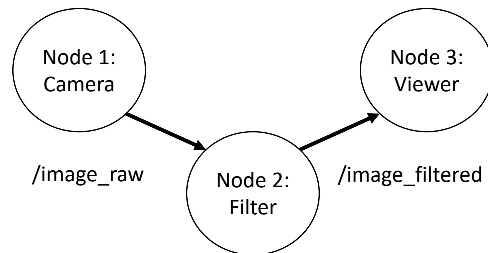

Fig. 4: Example ROS2 application

Listing 1: RECONROS configuration example

```
image_msg = rosmsg,sensor_msgs,msg,Image
node_2 = rosnode,"filter"
sub=rossub,node_2,image_msg,"/image_raw",10000
pub=rospub,node_2,image_msg,"/image_filtered"
```

Listing 2 presents the main HLS code for the implementation of the filter node. Using the RECONROS API, the processing loop starts with a blocking subscription for a new message containing an image. When a message becomes available, the function `ROS_SUBSCRIBER_TAKE` returns a pointer to the message data structure. With the help of the `OFFSETOF` macro, another pointer to the address of the message's payload is determined. Using this payload pointer, the image location can then be read from main memory via the `MEM_READ` macro. Subsequently, the image data is read into a `ram` structure within the FPGA, a Sobel filter is executed on the image, and the filtered image is written back to main memory via the `MEM_WRITE` macro. Finally, the node publishes the filtered data to the `/image_filtered` topic.

Listing 2: HLS implementation example

```
while(1) {
    uint32 msg_ptr =
    ROS_SUBSCRIBER_TAKE(resources_subdata,
        resources_image_msg );
    uint32 payload_pt_addr = OFFSETOF(
        sensor_msgs__msg__Image, data.data)
        + msg_ptr;
    MEM_READ(payload_pt_addr, payload_ptr,
        BLOCK_SIZE);
    MEM_READ(payload_ptr, ram, BLOCK_SIZE);
    Sobel_Filter(ram);
    MEM_WRITE(ram, payload_ptr, BLOCK_SIZE);
    ROS_PUBLISHER_PUBLISH(resources_pubdata,
    resources_image_msg);
}
```

V. EXPERIMENTS

In this section, we describe a set of experiments to demonstrate the feasibility and flexibility of RECONROS. First, we report on communication time measurements, then we demonstrate the flexible mapping of ROS2 nodes to either software and hardware and, finally, we present a hardware/software co-designed ROS2 node.

For all experiments, our setup consists of a desktop PC with an Intel Core i5 processor connected via Gigabit ethernet to a Mini-ITX 7Z100 board containing a Xilinx Zynq-7100 platform FPGA. On both platforms, we run ROS2 Dashing on Ubuntu 18.04 LTS. All applications use the same C/C++ source for software and hardware implementations. All software implementations have been compiled with optimizations level O3, and all hardware implementations have been created with HLS without any optimizations except for the Sobel filter.

A. Communication Times

To characterize communication times, we have implemented a ping-pong RECONROS application with two ROS2 nodes distributed onto a desktop PC and the Zynq board as shown in Figure 5. The ROS2 node on the PC publishes messages to the topic `T:/send`. The ROS2 node on the Zynq subscribes to these messages, creates copies of them in local memory, and publishes them to the topic `T:/recv`. The ROS2 node on the PC measures the roundtrip time of the transmission as $t_{round} = t_{end} - t_{start}$.

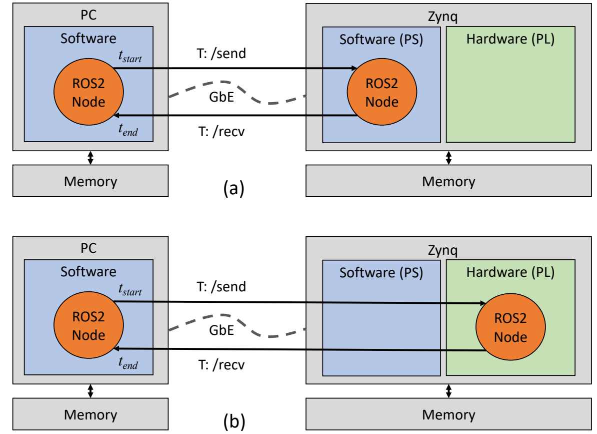

Fig. 5: RECONROS ping-pong application

Table II presents the measured roundtrip times for different message sizes and for the ROS2 node on the Zynq mapped to either software ($t_{round-SW}$), shown in Figure 5(a), or hardware ($t_{round-HW}$), shown Figure 5(b). Mapping the Zynq-based ROS2 node to hardware (PL) results in slightly increased roundtrip times compared to mapping it to software (PS). This effect is due to the fact that the hardware ROS2 node copies the message into its local memory (BRAM) first, before it writes it back to main memory, and this process is slower than the ARM CPU's memory copy operation. The last column shows t_{single}, the time to transmit a message single way, from the PC to the ARM core on the Zynq. This time has been determined by subtracting the runtime of the software-mapped ROS2 node from $t_{round-SW}$ and dividing by two.

Message size [Byte]	$t_{round-SW}$ [ms]	$t_{round-HW}$ [ms]	t_{single} [ms]
4	1.75	1.76	0.88
8 k	10.45	11.00	4.62
1 M	141.44	161.63	68.22
6 M	485.39	555.43	228.15

TABLE II: Ping-pong communication times

The RECONROS communication times in Table II are in the same order as that of related work [12], where communication times were measured between a ROS node on a PC and a ROS node on an ARM within a Zynq for messages of size 1 Mbyte and 6 MByte, albeit on a different ROS version. Importantly, the two versions of the RECONROS ping-pong application in Figure 5(a) and (b) are semantically identical. Switching from one to the other simply requires a change in the RECONROS configuration file and to start the software or hardware ROS node, respectively, in the application's main routine.

B. ROS2 Hardware Nodes

We have implemented three applications to demonstrate the flexibility of RECONROS in mapping ROS2 nodes to either software or hardware. All applications use the setup shown in Figure 5: A ROS2 node mapped to the PC publishes data to a topic and receives processed data on another topic. The processing itself is done by a ROS2 node on the Zynq platform, either by a ROS2 software node mapped to the ARM core (software version) or by a ROS2 hardware node mapped to reconfigurable logic (hardware version). The applications are:

1) Inverse kinematics: This application computes control signals for driving a servo motor that sets a joint angle based on a desired position and orientation of a robotic manipulation platform. The application is part of a larger mechatronic system [17] for controlling the movements of a Stewart platform [18] with six degrees of freedom. The computation involves coordinate transformations and an iterative implementation of the $\mathrm{arctan}()$ function. The ROS2 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 ROS2 output messages is also a 32 Bit unsigned integer containing a 10 Bit unsigned integer which is the pulse width coded control signal for the motor.

2) Number sorting: This application sorts an array of 32 Bit unsigned integers based on the odd-even transposition sort algorithm [19]. The algorithm is based on a comparator network that employs n stages with n comparisons each to sort n numbers. The ROS2 node on the PC generates random numbers and publishes messages comprising 2048 numbers as an array. The Zynq-based ROS2 node sorts the data and sends it back.

3) Sobel filter: This application implements a Sobel image filter [20] operating on three channels (RGB) of dimension 640×480. The filter applies two filter kernels on each channel of the image and calculates the absolute value of the dot product as an approximation for the geometric mean. The ROS2 input and output messages are of the type `Image` from the ROS2 sensor message package.

RECONROS application	Slice LUTs	DSP	BRAM
Inverse kinematics	4802 (1.73%)	17 (0.84%)	3 (0.40%)
Number sorting	10396 (3.75%)	0 (0.00%)	2 (0.26%)
Sobel filter	13625 (4.91%)	0 (0.00%)	10 (1.32%)
ORB-SLAM2	194585 (70.15%)	26 (1.4%)	82 (11%)

TABLE III: Resource usage and utilization (in % of the Xilinx Zynq 7100) for the implemented RECONROS applications

Rows 2-4 of Table III display the resource usage and utilization for the three applications and rows 2-4 of Table IV the execution times for the Zynq-bound ROS2 nodes, which are either mapped to the ARM core (software runtime) or to reconfigurable logic (hardware runtime). The inverse kinematics application achieves a speedup of 6, the number sorting application does not benefit from hardware mapping, and the Sobel filter is accelerated by a factor of 2. It has to be noted that the goal of these experiments has been to test and demonstrate the feasibility and flexibility of RECONROS rather than achieving high speedups through hardware acceleration. Hence, with the exception of an HLS unroll pragma for the Sobel filter no optimizations have been applied for HLS. There is obviously a certain potential to improve the speedups for the hardware accelerators, in particular for the number sorting application where more parallelism can be exploited.

RECONROS application	Software runtime	Hardware runtime	Speedup
Inverse kinematics	1.2 ms	0.2 ms	6.00
Number sorting	176.0 ms	342.0 ms	0.50
Sobel filter	44.0 ms	22.6 ms	2.00
ORB-SLAM2	0.77 fps	0.96 fps	1.25

TABLE IV: Runtimes for the software and hardware versions of the implemented RECONROS applications and achieved frame rates (in frames per second, fps) for ORB-SLAM2

C. Hardware-accelerated ROS2 SLAM

As an example of a larger RECONROS application that utilizes a hardware-accelerated ROS2 node instead of a ROS2 node fully mapped to hardware, we have implemented an ORB-SLAM2 [21] algorithm. ORB-SLAM2 is a state-of-the-art feature-based simultaneous localization and mapping (SLAM) algorithm for robotics, which is designed to operate in real-time in in-door and out-door environments. Due to its exchangeable input processing step and the use of key points for localization, ORB-SLAM2 supports a wide range of input data formats such as monocular images, stereo images, and RGB-D images. The input processing step transforms the input image into a map of features.

ORB-SLAM2 relies on corners as features and employs the FAST [22] algorithm for corner detection. FAST detects corners by inspecting a 16 pixel circle around a pixel of interest and checking whether at least nine contiguous pixels in that circle are darker or brighter than a given threshold. FAST is well-suited for parallelization since the computations are independent for all pixels. Our hardware/software co-designed version of the ORB-SLAM2 RECONROS application implements the compute-intensive FAST algorithm in reconfigurable logic, while the subsequent steps, i.e., (i) tracking of the camera by matching features from the camera to the local map, (ii) managing the local map, and (iii) detecting large loops and accumulating drift correction due to pose-graph optimization, are mapped to software and are executed on the ARM core.

Figure 6 sketches the architecture of the co-designed ORB-SLAM2 implementation. The ROS2 node receives incoming data by subscribing to the topic `T:/stereo`, which includes two subtopics with messages of type `Image` that correspond to two stereo channels (`T:/stereo/left/image_raw` and `T:/stereo/right/image_raw`). The software part of the ROS2 node writes the stereo images to main memory ① and calls the hardware accelerator to execute a FAST operation. This involves passing a pointer to the images in memory and information about the image size to the accelerator via the

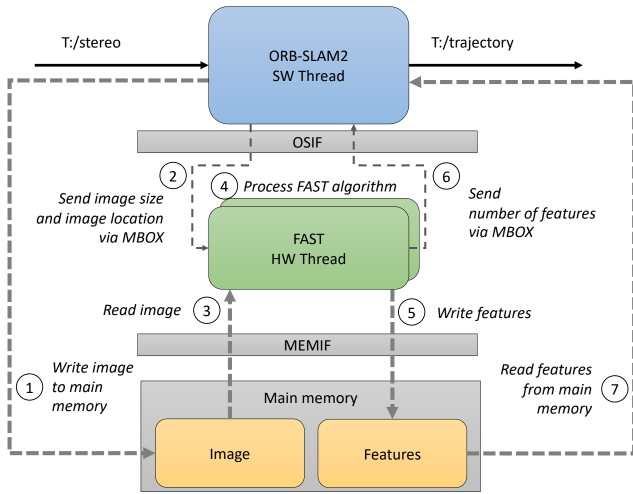

Fig. 6: RECONROS ORB-SLAM2 ROS Node

ReconOS mailbox communication primitive MBOX ②. The accelerator actually comprises two ReconOS hardware threads, one for each channel of the stereo stream. The hardware threads load the images into FPGA-internal memory ③, process them ④, and write the extracted features back to main memory ⑤. The number of extracted features are then sent back via ReconOS mailbox communication to the software part ⑥, which retrieves the features from main memory ⑦, proceeds with its computation and finally publishes the calculated trajectory to a corresponding ROS2 topic.

We have tested the ORB-SLAM2 application on the KITTI data set [23] to ensure its correct functionality. The hardware accelerator has been created with HLS, again without optimization. The resulting resource usage and utilization is shown in row 5 of Table III. The averaged execution time for the software application is $1.3s$ per image frame, and the hardware-accelerated application with two hardware threads achieves $1.04\ s$ per image frame. This results in the frame rates shown in row 5 of Table IV and a speedup of 1.25.

VI. CONCLUSION AND FUTURE WORK

In this paper we have presented RECONROS, a novel approach that enables developers of ROS2 robotics applications to leverage the performance and energy-efficiency of FPGA implementations. RECONROS bases on ReconOS and allows for flexible hardware acceleration of ROS2 nodes through an API that supports a consistent programming model for ROS2 nodes across hardware/software boundaries, while preserving 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.

In future work we want to also provide the ROS2 communication patterns services and actions to hardware nodes. Since in distributed ROS networks not all compute nodes might be equipped with platform FPGAs, we plan to investigate the feasibility of a ROS2 node offering acceleration-as-a-service. Furthermore, we want to leverage partial reconfiguration available with ReconOS [8] to manage the reprogrammable hardware resources more efficiently, for example by configuring ROS2 hardware nodes on demand. Finally, we plan to deploy an FPGA-based compute board on small scale drones to demonstrate the benefits of ROS-based flexible onboard hardware acceleration in drone applications such as surveillance and search and rescue [24].

REFERENCES

[1] S. Weiss, D. Scaramuzza, and R. Siegwart, "Monocular-SLAM-based navigation for autonomous micro helicopters in GPS-denied environments," *Journal of Field Robotics*, vol. 28, no. 6, pp. 854–874, 2011.

[2] R. Jung, C. Brommer, and S. Weiss, "Decentralized Collaborative State Estimation for Aided Inertial Navigation," in *Proc. International Conference on Robotics and Automation*, 2020.

[3] E. Yanmaz, S. Yahyanejad, B. Rinner, H. Hellwagner, and C. Bettstetter, "Drone networks: Communications, coordination, and sensing," *Ad Hoc Networks*, vol. 68, pp. 1–15, 2018.

[4] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels," in *Proc. 2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[5] C. Brugger, L. Dal'Aqua, J. A. Varela, C. D. Schryver, M. Sadri, N. Wehn, M. Klein, and M. Siegrist, "A quantitative cross-architecture study of morphological image processing on CPUs, GPUs, and FPGAs," in *Proc. 2015 IEEE Symposium on Computer Applications Industrial Electronics (ISCAIE)*, 2015, pp. 201–206.

[6] O. Ulusel, C. Picardo, C. B. Harris, S. Reda, and R. I. Bahar, "Hardware acceleration of feature detection and description algorithms on low-power embedded platforms," in *Proc. 2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.

[7] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 326–342, 2019.

[8] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2014.

[9] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the Performance of ROS2," in *Proc. 13th International Conference on Embedded Software*, ser. EMSOFT '16. Association for Computing Machinery, 2016.

[10] K. Yamashina, T. Ohkawa, K. Ootsu, and T. Yokota, "Proposal of ROS-compliant FPGA component for low-power robotic systems (retraction notice)," in *Proc. International Conference on Intelligent Earth Observing and Applications 2015*, vol. 9808, 2015, p. 98082N.

[11] K. Yamashina, H. Kimura, T. Ohkawa, K. Ootsu, and T. Yokota, "CReComp: Automated Design Tool for ROS-Compliant FPGA Component," in *Proc. IEEE 10th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2016*. IEEE, 2016, pp. 138–145.

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

[13] T. Ohkawa, Y. Sugata, H. Watanabe, N. Ogura, K. Ootsu, and T. Yokota, "High level synthesis of ROS protocol interpretation and communication circuit for FPGA," in *Proc. 2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*, 2019, pp. 33–36.

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

[15] B. Strohmer, A. BØgild, A. S. SØrensen, and L. B. Larsen, "ROS-Enabled Hardware Framework for Experimental Robotics," in *Proc. 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019.

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

[17] C. Lienen, "Implementing a Real-time System on a Platform FPGA operated with ReconOS," Master's thesis, 2019.

[18] D. Stewart, "A Platform with Six Degrees of Freedom," in *Proc. of the Institution of Mechanical Engineers*, vol. 180, no. 1, 1965, pp. 371–386.
[19] D. Knuth, *The Art of Computer Programming: Volume 3: Sorting and Searching*. Pearson Education, 1998.
[20] R. Gonzalez and R. Woods, *Digital Image Processing*. Pearson, 2018.
[21] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.
[22] E. Rosten and T. Drummond, "Machine Learning for High-Speed Corner Detection," in *Proc. European Conference on Computer Vision*, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443.
[23] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, 2013.
[24] J. Scherer and B. Rinner, "Multi-Robot Persistent Surveillance With Connectivity Constraints," *IEEE Access*, vol. 8, pp. 15 093–15 109, 2020.