Model-based Generation of Hardware/Software Architectures with Hybrid Schedulers for Robotics Systems

Ariel Podlubne, Johannes Mey, Andreas Andreou, Sergio Pertuz, Uwe Aßmann and Diana Göhringer

Abstract—Robotic systems compute data from multiple sensors to perform several actions (e.g., path planning, object detection). FPGA-based architectures for such systems may consist of several accelerators to process compute-intensive algorithms. Designing and implementing such complex systems tends to be an arduous task. This work proposes a modeling approach to generate architectures for such applications, compliant with existing robotics middlewares (e.g., ROS, ROS2). The challenge is to have a compact, yet expressive description of the system with just enough information to generate all required components and to integrate existing algorithms. The system model must be application-independent and leverage FPGA advantages, such as concurrency, energy efficiency, and acceleration due to custom designs, surpassing software-based solutions. Previous work mainly focused on individual accelerators rather than all components involved in a system and their interactions. The proposed approach exploits the advantages of model-driven engineering and model-based code generation to produce all components, i.e., message converters as middleware interfaces and wrappers to integrate algorithms. Data type and data flow analysis are performed to derive the necessary information to generate the components and their connections. Six different schedulers are proposed to cover multiple scenarios. Solutions to several identified challenges for generating entire systems from such models are evaluated using four different use cases.

Index Terms—HW/SW Co-Design, Robotics, Code Generation, Model-Based, Embedded Hardware, FPGAs.

1 Introduction and Motivation

The range of robotic applications has been increasing lately, from manufacturing [1], collaborative robots interacting with humans [2], biomedicine [3], drones [4] as well as mobile robots [5], to name a few. As the range of applications expands, robotic platforms are becoming more complex. This complexity arises from the requirement to concurrently process heterogeneous data from multiple sensor types to meet real-time constraints. An architecture should facilitate the development of robotic systems by providing helpful constraints on the design and implementation of the desired application without being overly restrictive [6]. Field Programmable Gate Arrays (FPGAs) provide the flexibility to reprogram them with custom designs, suited for targeted applications. However, designing FPGA-based architectures for such systems tends to be an arduous process as it requires low-level hardware (HW) knowledge and a long and complex design process. Even though the proven advantages of FPGAs (i.e., concurrency, lower energy consumption, acceleration due to custom HW design) for robotic applications [7], [8], [9], porting them from software (SW) to embedded HW platforms or accelerating parts requires the creation of suitable interfaces. This often means the re-design of several parts of the applications. Lastly, the manual interconnection of multiple components for complex applications (i.e., multiple accelerators) turns into an error-prone process. Therefore, this work proposes a modeling approach to automatically generate and deploy architectures for robotic applications in FPGAs. The *research questions* to answer are how to generate all required components for such architectures from a *holistic model* and how that model should be defined. This brings some requirements: (R1) the description should be compact, concise, but expressive enough containing the necessary information to *derive* the system's components and their relations. (R2) the approach must be application-independent, and (R3) it must exploit the benefits of FPGAs over SW solutions. Finding the optimal trade-off among these requirements poses significant challenges. Existing solutions address specific subsets of these requirements, but this study aims to fill the gap by providing a comprehensive solution that covers all of them. The tree main identified challenges are: (CH1) Obtain the explicit and derive the implicit information from the *system specification*. (CH2) The *system specification* has to be a *compact and meaningful* description so writing it is not as cumbersome as deploying the system manually. (CH3) There has to be an *understanding* of the specifications of interfaces to generate the compliant components and the relations among each other. Addressing these requirements

- *A. Podlubne, A. Andreou, S. Pertuz and D. Göhringer are with the Chair of Adaptive Dynamic Systems, TU Dresden, Germany., E-mail: {ariel.podlubne, andreas.andreou, sergio.pertuz, diana.goehringer}@tu-dresden.de*
- *J. Mey and U. Aßmann are with the Chair of Software Technology, TU Dresden, Germany and the 6G-life project., E-mail: {johannes.mey, uwe.assmann}@tu-dresden.de*
- *A. Podlubne, U. Aßmann and D. Göhringer are with the Centre for Tactile Internet with Human-in-the-Loop (CeTI), TU Dresden, Germany*

This work has been funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy - EXC 2050/1 - Project ID 390696704 - Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) *of Technische Universität Dresden, and by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt". Joint project 6G-life, project identification number: 16KISK001K.*

needs sophisticated tooling, specification design, and static and runtime analysis. These go beyond the scope of configuration and scripting-based approaches found in related works, which only tackle specific aspects of the problem. Hence, the *main contributions* of this work are:

- `Model Analysis`: A comprehensive *analysis* of the *system specification* to *derive* the *holistic model* that includes all the components to generate, their interfaces and how they all interact among each other.
- `Interfaces`: Wrappers for accelerators based on middleware specifications to ease their integration.
- `Scheduler`: Multiple algorithms tailored for hybrid HW/SW architectures.
- `Design Space Exploration`: Evaluation framework to decide the most suitable scheduler for each application.

This work builds upon our previous one [10], extending the pool of scheduling algorithms and incorporating Domain Space Exploration (DSE) for evaluation and selection of the most suitable algorithm for each application. To the best of the authors' knowledge, no comparison of multiple HW IP-based scheduling algorithms, including the one presented in this work, has been conducted in the given context. Furthermore, our novel contribution of generating the complete system from a simple description replaces the arduous process of the traditional FPGA flow, allowing roboticists, who may not be FPGA experts, to leverage the benefits of FPGAs.

2 Related Work

Research over the last years showed the potential advantages of FPGAs over Central Processing Units (CPUs) and Graphics Processing Units (GPUs) concerning performance, energy consumption and latency for efficient implementations of robotics applications [7], [8], [9]. An essential aspect of FPGA acceleration for robotic applications is how to integrate accelerators into software-centric robotics systems. Most approaches rely on the Robot Operating System (ROS), which has become the mainstream middleware used by roboticists over the years. It is an open-source middleware that runs on top of Linux and provides an off-the-shelf solution to deploy algorithms easily in a SW distributed system. It consists of *nodes* where computation is performed, and they exchange *messages* with each other over *topics*. Different approaches have been proposed to combine FPGAs and ROS, improving the computational power of robotic systems. However, most of them focus on a single dedicated solution, often neglecting the integration aspects. The extended capabilities of the new ROS2 version finally enable real-time support and the specification of Quality of Service (QoS) settings for publishers and subscribers. Given the significance of communication between SW and HW in meeting these requirements, there is a need for flexible message scheduling to address potential bottlenecks.

Early approaches [11], [12], [13], [14] focused on HW/SW Co-Design techniques to accelerate parts of SW applications. They proposed the automatic generation of interfaces between HW and SW, partitioning ROS message specifications. They aimed to minimize the communication time between these two and use High Level Synthesis (HLS) to increase productivity. These solutions were not integral, as they mainly focused on particular applications rather than general architectures.

Shi et al. [15] presented a heterogeneous platform based on OpenCL to enable researchers and engineers without FPGA expertise to develop heterogeneous computing applications efficiently. They mainly targeted the most active research fields in robotics, namely Simultaneous Localization and Mapping (SLAM), motion planning, and Convolutional Neural Network (CNN). Even though it simplifies the process by relying on OpenCL, the authors mainly focused on particular kernel generation. The concurrent execution of multiple kernels as a holistic system is left for future work.

Lienen et al. [16] presented an event-based programming approach that leverages [17], a framework to map ROS2 nodes to SW or reconfigurable HW. It is based on callbacks to partially reconfigure pre-allocated slots for either SW or HW executions. There may be a limiting factor for specific applications with time constraints smaller than the reconfiguration time needed for each callback. The work was extended by introducing a reconfigurable slot model [18]. This enables dynamic loading and execution of HW-mapped ROS2 callback nodes instead of statically placing them in reconfigurable logic. The scheduling algorithm follows a First In First Out (FIFO) approach for registering callbacks, which can make retrieved data unusable, as discussed in Section 4.3. This motivates the proposed scheduler schemes in this work. The evaluation includes six accelerators, but the system's scalability remains unclear with a large number of accelerators.

A complete HW architecture and the generation of interfaces compatible with robotics middlewares is presented in [19]. The main advantage of this work is the automatic generation of HW components based on message specifications to interface HW accelerators. The automatic integration of accelerators is solved in the current work, eliminating the need for manual implementation. The proposed model-based approach facilitates a comprehensive understanding of the desired system via model analysis. This is particularly crucial for *simplyfing* the generation and deployment of complex systems, such as those involving multiple interconnected accelerators with full middleware-based interfaces.

CPU+FPGA scheduling has been previously studied [20], [21], [22], mainly focusing on one specific algorithm for a given use case. [23] proposed heterogeneous resource-elastic scheduling for maximizing the utilization of CPU and FPGA resources by dynamically scaling the resource allocation for tasks. Unlike these approaches, our work provides different algorithms that can be seamlessly integrated with DSE. This enables the selection of the optimal algorithm for each use case, particularly in scenarios involving data exchange between SW and HW. The scaling factor of the system is relevant and it is also evaluated.

3 Code Generation Workflow

A typical robotics system consists of various components, including CPUs, accelerators, and *converters* that serve as interfaces between them. The concepts shown in this work follow the Zynq device model with a Processing System (PS) and a Programmable Logic (PL) sharing data via Direct

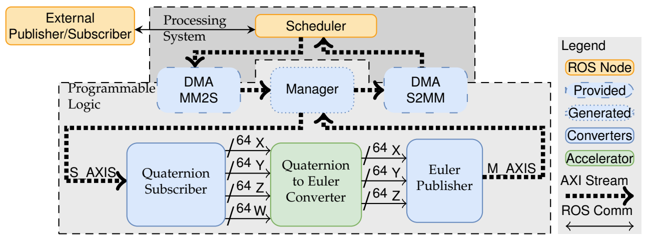


Fig. 1: Quaternion to Euler converter with ROS interfaces

Memory Access (DMA). However, this is not limited as the CPU support can be extended (e.g., soft-cores) or removed if not needed. An example is shown in Figure 1, which consists of a *subscriber converter* to receive a quaternion, an *accelerator* to compute the conversion to Euler angles, and a *publisher converter* to broadcast the result. AXI Stream (AXIS) slave (S_AXIS) and master (M_AXIS) connect to DMA through a *Manager* to schedule transactions between PS and PL [19]. Understanding the characteristics of each component and their interactions is essential to automatically generate the necessary artifacts and deploy robotics systems according to a given specification. Listing 1 shows how to describe such a system for the proposed workflow. The interfaces of the *accelerator* (Line 12 and Line 15) include a *message* type. Wrappers are generated to provide the desired signals corresponding to that message type for the components performing the computation. In this case, the *accelerator* is an HLS component (Line 7), so the equations for the conversion are defined in a *.cpp* file (Line 8), leavriging tools such as Vivado or Vitis HLS. VHDL is also supported and further HW description languages can be added with *templates*, as explained below. A SW implementation is also possible by changing the *type* to ROS-SW. How to specify all components and how they interact with each other is shown from Line 17. Similar to the accelerators, it is necessary to define the message types for *publishers* and *subscribers* . Lastly, the output of each component must be declared as *outgoing*, defining the destination block. Like so, in a compact specification, the characteristics of accelerators, their interfaces, and how to establish the communication for incoming and outgoing data have been defined. Multiple components are involved in such architectures besides the converters and accelerators. They are the ones that depend on the integrity of the system (i.e., Manager, DMA), depending on how many converters and accelerators are involved. These components are not part of the *system specification* as they are not generated, but their configuration is derived from it. Additionally, *tailored scripts* are needed to deploy the entire architecture. The workflow of the proposed toolchain is shown in Figure 2, with Listing 1 as an example of a *system specification*.

Model Analysis: The information to generate the different converters, wrappers for accelerators, and tailored scripts is deduced *only* from the *system specification*. All required information that is not explicitly defined (e.g., total components to manage transactions between PS and PL) is derived by doing data-type and data-flow analysis of the message types and connections of the components. All individual connections (at signal level) among all blocks, based on the specification and their interfaces, are also inferred. All this *derived* information is expressed in an

```
1  project:
2    name: RotationConverterNode
3    fpgaPart: xc7z020clg400-1 #FPGA family is derived
4  # Definition of accelerator types
5  accelerators:
6    - name: QuaternionToEuler
7      type: HLS # can be HLS, VHDL or ROS-SW
8      sources: ./QuaternionToEulerConverter.cpp
9      interface:
10       input:
11         - middleware: ROS
12           message: geometry_msgs/Quaternion
13       output:
14         - middleware: ROS
15           message: geometry_msgs/Point
16 # Definition of all components and their relations
17 blocks:
18   - name: Quaternion_sub        # converter
19     type:                       # ROS > accelerator
20       middleware: ROS
21       mode: subscriber
22       message: geometry_msgs/Quaternion
23     outgoing:                   # can have many destinations
24       - name: QuatToEuler_acc
25   - name: QuatToEuler_acc       # accelerator of the type
26     type: QuaternionToEuler     # defined in line 7 and it
27     outgoing:                   # can be used multiple times
28       - name: Euler_pub
29   - name: Euler_pub             # converter
30     type:                       # accelerator > ROS
31       middleware: ROS
32       mode: publisher
33       message: geometry_msgs/Point
```

Listing 1: System specification for a Quaternion to Euler system

extended and detailed version of the *system specification*, as a *template configuration* for the *template engine*.

Template Engine: The *template engine*[1] along with templates are used to generate the *intermediate artifacts*. A *template* is a generic source code that resembles the expected artifact. It is expanded with given specifications (*template configuration*) accordingly to the needs (e.g., names, bit widths). These templates are included in the toolchain. They are *coded once* and are re-used for any *system specification*. There are multiple ones involved, according to the *intermediate artifact* to generate. These can be for HLS or VHDL sources (e.g., converters) or *tailored scripts* as configurations for *vendor dependent* tools to generate the expected components. New *templates* can be added to the toolchain with ease to extend it for new components, additional HW description methods (e.g., Verilog), or scripts for different vendors. This is proven in [19] by extending support from ROS1 to ROS2. The main difference lies in the serialization of messages in ROS2 to improve memory alignment and influences the logic of the generated VHDL converters. It is worth noting that most proposed HW components remain independent of the

1. Mustache—Logic-Less Templates, https://mustache.github.io

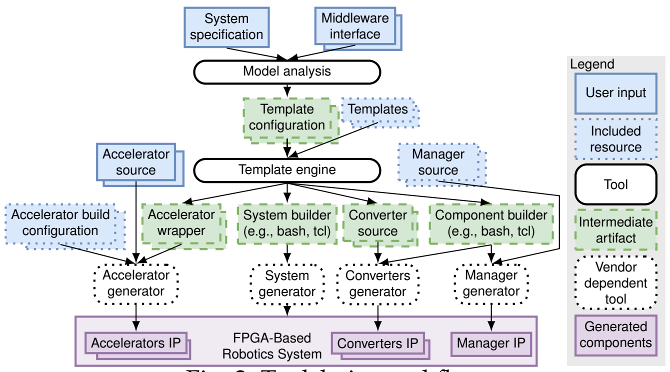


Fig. 2: Toolchain workflow

```
1  accelerators:
2    - name: GrayScale
3      type: HLS
4      sources: ./grayScale.cpp
5      interface:
6        output:                # Same for input
7          - middleware: ROS    # (simplified due to space)
8            message: sensor_msgs/Image
9            include: ["height", "width", "data"]
10 blocks:
11   - name: ImgFilter_sub      # converter
12     type:                    # ROS > accelerator
13       middleware: ROS
14       mode: subscriber
15       message: sensor_msgs/Image
16     outgoing:
17       - name: ImgFilter_pub
18         exclude: ["height", "width", "data"]
19       - name: GrayScale
20         include: ["height", "width", "data"]
21   - name: GrayScale_acc      # accelerator of the type
22     type: GrayScale          # defined in line 2
23     outgoing:
24       - name: ScaleDownNearest_acc
25         include: ["height", "width", "data"]
26   - name: ImgFilter_pub      # converter
27       type:                  # accelerator > ROS
28         middleware: ROS
29         mode: publisher
30         message: sensor_msgs/Image
```

Listing 2: Snippet of the connections between accelerator and publisher converter

middlewares, as the middleware's influence is reflected *only* in the logic of the converters.

Generators: They take the *intermediate artifacts* to build and deploy the entire system. There are two types. Those that generate components (i.e., accelerators, converters, manager), and the *system generator* which does not generate components but uses them. The latter one takes a set of *tailored scripts* for each application and the information of a (vendor-dependent) targeted platform. It deploys all the generated components and the derived ones (e.g., Manager, DMA). Additionally, as their interactions have been derived (each individual signal), it connects all of them accordingly, as specified in the *template configuration*. This work uses bash scripts to handle different tcl scripts for Vivado, and Vivado and Vitis HLS tools. These tools are used to import the generated and provided sources (i.e., .cpp for the accelerators, .vhd for the converters and manager), and export them as IPs for automatically deploying the desired *holistic system*.

4 Code generation Challenges for HW/SW Architectures

The generation of the previously described architecture presents three main challenges, which are outlined below.

4.1 Concise Holistic Model

An important aspect is to have a concise but expressive description of the system (CH2), as shown in Listing 1. This means that there has to be a mechanism to *include* or *exclude* signals from one component to another. Examples of these are shown in Listing 2 (Line 18 and Line 20). These keywords are analyzed to determine *which signals* corresponding to a message specification (Line 15) should be connected to which component. They can be individual signals as well as submessages. Analyzing the structure of the message definition allows filtering and deriving the desired signals from one component to another.

4.2 Dynamic Frame Length

Listing 2 shows the specification of a system which contains an HLS accelerator of an image processing application compliant with a *sensor_msgs/Image* message from ROS. This message includes a string (i.e., frame_id) which varies with every new frame, and the image itself could also vary depending on the application (e.g., image upscaling/downscaling). Hence, the number of bytes for the *publisher* to transmit (frame length) can change dynamically. Figure 3 depicts the generated components for such system. It contains the *subscriber* and *publisher converters* (to send/receive the image message from/to the PS over DMA), and the image processing application itself provided by the user (Line 4). The transmission of the message through the *publisher converter* cannot start unless the total number of bytes (frame length) to transmit is known. Hence, the *frame length* component computes this at runtime. Considering the case that the message *may not transmit* all of its fields, or the ones containing fields that *change their length dynamically* (e.g., strings), the total length cannot be known at compile time. Therefore, a tailored component to obtain dynamically the frame length of each *publisher* is generated when needed and added as shown in Figure 3. SW implementations have access to large memory blocks, and the entire message is available constantly. This is not possible on the HW side as data is streamed, which makes it necessary to have a mechanism to compute the total bytes in each frame as they can change dynamically. The fields of a message that are involved in this computation are derived by analyzing Listing 2. This will provide the individual lengths of dynamically changing fields that are needed as *inputs* for this new component to compute the *publisher*'s frame length. The fixed-sized field lengths are calculated in the analysis, as these are known at compile time.

Algorithm 1 computes the length of a message from its contained fields using the helper methods FieldLength to compute the length of a field and TypeLength to compute the length of a type. Fields can be arrays of variable length not known before receiving a message. Thus, signals connected to AXIS must be used to obtain the length at runtime using the *signal()* function. Note that because arrays (and messages) can be nested, but their contents are not uniform, each information taken from a signal must be obtained at the right time during the reception of the message. This means the TotalLength can only be computed once the last size signal of an array within the message has been received. Because the size signals are evaluated at different times, parts of the message might need to be buffered [24], which

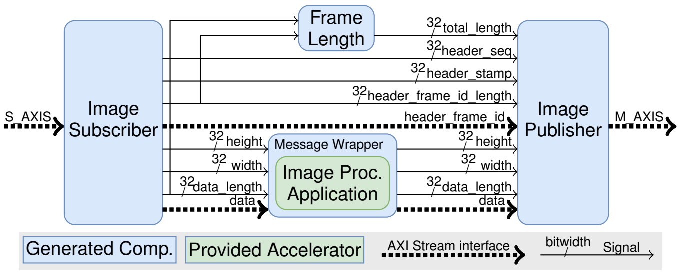


Fig. 3: Payload of an image publisher dynamically computed

is also inferred at the *Model Analysis* stage.

4.3 Scheduling Transactions between HW and SW

Addressing the communication between the HW and SW components is a crucial aspect when dealing with hybrid HW/SW systems. In our case, the SW part is based on ROS. Its default scheduling scheme to receive new messages is shown in Figure 4a. It consists of a *shared callback queue* for all subscribers. Hence, the callback queue needs to be read three times (retrieving B) before A can be read, in a FIFO manner. This can cause a message to no longer be usable for a given subscriber. Therefore, a modification to this scheme is proposed (Figure 4b) by using individual callback queues for each incoming message. This leads to the question of which *spinner thread* should get a hold of the DMA to exchange data from PS to PL. Similarly to the different SW threads, accelerators also compete to get a hold of the DMA to send data from PL to PS. Therefore, multiple scheduling algorithm are proposed and their detailed HW implementation is described in Section 6. The HW schedulers remain independent of the ROS version, as the converters are influenced by the middleware and responsible for providing the necessary interfaces.

5 The Model-Driven Code Generation Toolchain

After discussing the toolchain in Section 3 and three particular challenges in Section 4, this section explains the technical details of the implementation and argues why a model-driven approach is beneficial. Model-driven engineering [25], [26] offers a systematic and domain-oriented development approach using domain-specific models, model transformation and code generation to create comprehensible and maintainable SW. The toolchain shown in Figure 2 has two essential components: the *model analysis* and a set of provided resources and inputs that are used to construct the system using a template engine. In this case, a *logic-less* template approach is used with simple placeholders in the template rather than programmed instructions, which simplifies the definition of templates for domain experts. Therefore, all analysis and reasoning must happen *within* the tool.

The toolchain uses and extends the open-source solution provided by [19] and thus also uses a *grammar-based* modeling approach based on attribute grammars [27]. As opposed to other modeling approaches, grammars describe *trees* rather than models comprised of arbitrarily structured elements. This approach was used in [19] to derive all required information to generate the converters for individual messages; here, we include the middleware-based interfaces generated in [19] and extend the approach to the generation of the entire system. Thus, the analysis must be able to derive all relevant information for the creation from the *system specification* (e.g., in Listing 1) and the provided static resources. Attribute grammars are an approach to compute semantic properties of a language (or, in our case, a model) in a declarative and formalized way. In this case, the concept of *higher-order attributes* [28] is used, which additionally allows the computed properties to be entire new artifacts. For this, we employ *relational reference attribute grammars* [29], [30], which allow efficient linking of tree elements with cross-tree *relations*.

The three challenges identified in Section 4 are used to illustrate why such a model-based approach is a necessary and adequate solution to generate HW/SW architectures.

5.1 Tailored Information using Intermediate Representations

Since a major target of the proposed system is to have concise specifications (CH2), most required information to construct a complete system is only included *implicitly*. However, the employed template engine needs all information *explicitly* specified; thus, an analysis with computed attributes on the input model are used. However, doing this transformation in one step is difficult and does not allow for reuse (R2), since there are multiple template configurations to be created. Therefore, we employ multiple intermediate representations, i.e., models based on reference attribute grammars obtained using model transformation using higher-order attributes.

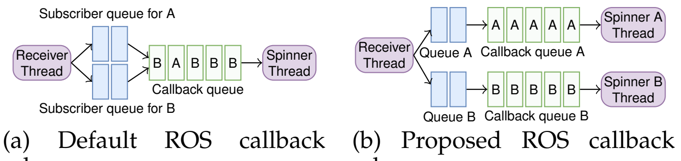
One example is an extended system specification model. As suggested in Section 4.1, to keep the input specification concise *and* the implementation efficient, signals connecting messages can be filtered using `include` and `exclude` hints. This is a shorthand for the specification of all required signals, which is only possible, because the contents and nestings of message types are analyzed. In the full *system specification*, the inclusion hints are expanded to contain a (potentially long) list of all individual fields to be included.

Algorithm 1 Computation of message length

```
function MESSAGELENGTH(Message $m$)
    $l := 4$
    for each Field $f$ in $m$ do
        $l := l + \text{FIELDLENGTH}(f)$
    return $l$

function FIELDLENGTH(Field $f$)
    $l := 0$
    if $f$ is no array then
        $l := l + \text{TYPELENGTH}(f)$
    else if $f$ is fixed-length array then
        for $i$ in range(array_length(f)) do
            $l := l + \text{TYPELENGTH}(index(f, i))$
    else if $f$ is variable-length array then
        $l := l + 4$
        for $i$ in range(signal(f, length)) do
            $l := l + \text{TYPELENGTH}(index(f, i))$
    return $l$

function TYPELENGTH(Field $f$)
    $l := 0$
    if type_of(f) is built-in type then
        $l := size\_of(t)$
    else if type_of(f) is message then
        $l := l + 4$
        for each Field $s$ in $f$ do
            $l := l + \text{FIELDLENGTH}(s)$
    return $l$
```



(a) Default ROS callback scheme

(b) Proposed ROS callback scheme

Fig. 4: ROS Scheduling Schemes

5.2 Simplifying Runtime Computation

The computation of the length of the message was already highlighted in Algorithm 1. It consists of two main functions used in a recursive process following the nested structure of a message definition. The first benefit of the chosen approach is that the algorithm can be simplified when considering the intermediate message representation from [19], which no longer contains fixed-length arrays and less nested messages, which have been flattened whenever possible. This removes the else branch in lines 10 to 12 of Algorithm 1 and reduces the nesting depth of the function calls. Secondly, since the signal data required in the algorithm are available at different times, function calls have to be inlined depending on the message type. So, again, type analysis is required. Finally, the signals required by the algorithm must be connected, which requires a data flow analysis, which can be performed using the attribute grammar approach [31]. Additionally, optimization can be applied if signals are known at compile time, e.g. when signals are not connected.

5.3 Benefits of Model Analysis in the Development Lifecycle

In addition to the aforementioned analysis and optimization steps, the use of a model-based, attribute grammar analysis approach allows for potential further analysis improving performance at development time, compile time and runtime. During development, the construction and verification of the system model can be aided by static analysis, aiding the developer with syntactic and semantic checks, code completion and suggestions, and refactoring support. During compile time, knowledge of the entire system can help with the generation of optimized code beyond the abilities of the FPGA compiler toolchain or optimizations for better resource utilization. One example for runtime benefits is the use of Worst Case Execution Time (WCET) analysis to ensure real-time guarantees in combination with Algorithm 1 to adapt the scheduling scheme dynamically, knowing the time left for the accelerator.

6 SCHEDULERS

The heterogeneous data transmission between PS and PL components are synchronized through different scheduling algorithms, proposed here to have different options, adaptable for each application. The algorithms presented here focus on data movement between CPUs and FPGAs, as the algorithms are meant to be fully implemented as IP cores, compared to related work that mostly focuses on accelerating mostly the compute-intensive parts. Despite the advancements in real-time scheduling in ROS2, which are independent of HW/SW architectures, it is important to note that the schedulers presented here focus specifically on addressing resource starvation issues when partitioning applications in a HW/SW Co-Design. In our case, ROS is used as the communication layer to send/received data between non-accelerated external components, and the PS-PL communication is after the callbacks.

A SW implementation is needed to schedule the transactions from PS to PL, and a HW counterpart for the PL to PS ones. The HW implementation details are described below, as one needs to consider the low-level signals that are not needed in SW. Particularly for this work, tasks represent the time each component can stream its data. In general, a scheduler has multiple inputs for the *requests* from the accelerators, meaning that they have data available to broadcast and they are ready to be scheduled. The scheduler's output is the *grant*, allowing only one accelerator to perform the transmission at a time.

As the AXIS is the chosen communication protocol, `tvalid` is used as *requests* and `tready` is used as *grants*. All proposed schedulers follow the same philosophy with different variations in how each computes the grants. Each accelerator that sets `tvalid` to `1` will get a *grant* as long as it is the only one with the highest priority. Only one accelerator can get the *grant* on each clock cycle. Therefore, it will be computed according to each algorithm when multiple accelerators have data to stream simultaneously. The end of each task is denoted with `tlast`, following the AXIS protocol.

There are four characteristics considered for the schedulers:

- ***Preemptive:*** a running task is paused when a higher priority task arrives and gives the grant. The first one resumes after the latter one completes.
- ***Non-Preemptive:*** a running task will not be interrupted until its execution is completed.
- ***Fixed Priorities:*** priorities are set at design time and kept for the entire runtime of the process.
- ***Dynamic Priorities:*** priorities are updated dynamically during runtime according to the scheduling algorithm.

There is no need for dynamic task allocation during runtime because the task-to-accelerator mapping is already fixed and determined at compiletime. Considering that the end goal is to generate all these components from an abstract description of the system, the core of implementing the different schedulers has to be generalizable. The adaptable statechart shown in Figure 5 serves as the base, comprising two types of states and transitions. Some states are static, among all schedulers, while others are customized to meet the needs of each algorithm, encompasing computations such as deadline and slack , as well as transition conditions. Therefore, only certain parts of the statechart differ from one scheduler to the other. A *priority table* is initialized at the beginning. The algorithm-dependent conditions are computed to use them for updating the *priority table* accordingly, depending on the algorithm. The updated priorities are used in the *Set Grant* superstate to find the maximum value (highest priority) to asses which accelerator will get the grant. The transitions within this superstate also depend on the algorithm, as each of them dictates how to react to new requests or internal conditions.

The automatic code generation process is simplified by following this statechart for all algorithms. However, as each algorithm has different characteristics and computes priorities differently, the *Compute Condition* state defines how this is done. There is no relation between accelerators and the schedulers, so there is data independency for all tasks.

Implicit deadlines were chosen for the Earliest Deadline Fist (EDF) and Least Slack Time (LST) algorithms presented below. This means, for this work, that relative deadline D_i

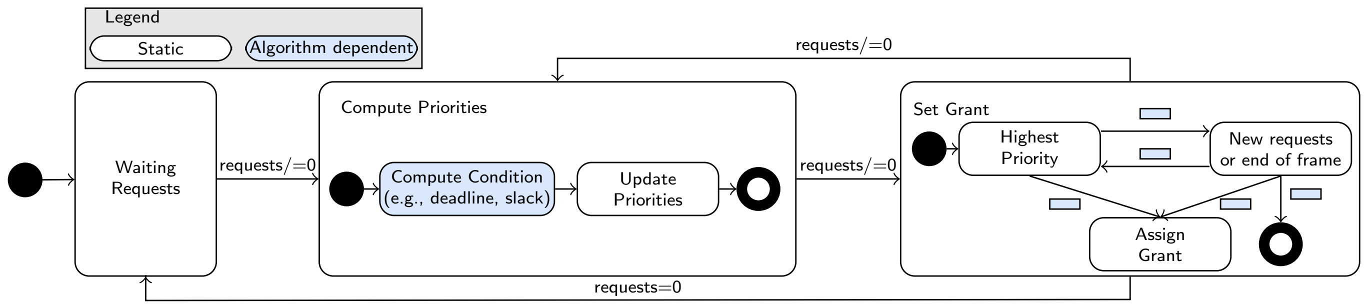


Fig. 5: Adaptable statechart for multiple schedulers

is not larger than the transfer time (T_i) plus the frequency F_i (number of clock cycles after its last transmission, and the availability of new data to be streamed) , i.e., $D_i \leq T_i + F_i$, for every task τ_i. Moreover, soft real-time constraints are assumed for the system, meaning that missed deadlines will not have severe consequences. Four different algorithms are proposed, two of them with two variations, making six schedulers in total. They are based on SW solutions with the corresponding adaptations to HW implementations and a chosen streaming interface. Below are the details of each of them:

Least Recently Used (LRU): this algorithm is mainly used to manage buffer memories and caches. It dynamically changes the priorities based on the accelerator that got the *grant* the latest. This one will be moved to the bottom of the priority list, ensuring all accelerators to get the *grant*. This guarantees that there will be no resource starvation. However, some accelerators will likely miss their deadlines. This is more evident the more accelerators are included because it takes N rounds (in the worst case) for an accelerator to be on top of the priority list. To avoid this, only the accelerators which set their *requests* are considered each time the priorities are evaluated.

Fixed Priority (FP): this is a static priority algorithm where priorities remain unchanged throughout the entire execution time. The VHDL model's entity defines the priorities based on its N-bit port (N is the amount of accelerators), with lower-priority signals assigned to the lower bits and higher-priority signals assigned to the higher bits.. Each $tvalid_i$ signal acts as *request* and determines the priority of each accelerator. The lower the N, the higher the priority. Accelerators are assigned indexes, and the one with the lowest index, which sets its *request* is assigned the *grant*. There are two variants for this algorithm: preemptive or non-preemptive. The difference lies in whether the *grant* may change at any clock cycle.

EDF: this is a dynamic priority scheduler. Priorities are updated dynamically on each clock cycle throughout the entire runtime. These updates depend on the state of the *requests* and the proximity of each deadline to the current time. Deadlines are decremented for each accelerator, with the request set to one on every clock cycle. Newly arrived requests are assigned a priority based on their implicit deadline. Two main computations are required. The first one is to decrement the deadlines of every accelerator to increase their priorities. As this is a dynamic priority scheduler, priorities are updated whenever a new *request* arrives to modify the *priority table*. There are two versions proposed. Firstly, the resources-optimized one (*ROEDF*) follows the statechart shown in Figure 5. Secondly, the latency-optimized version (*LOEDF*) is a slight modification of the generic statechart as some computations are merged into the same state. This reduces the number of state transitions, resulting in fewer clock cycles but comes at the cost of increased resource consumption. Thus, a tradeoff exists between resource utilization and latency.

LST: is also a dynamic priority scheduler. Contrary to EDF, this algorithm will evaluate the slack time of the accelerator requesting the grant on every clock cycle according to $s_i = d_i - a_i - c_i$, where s_i is the slack time (priority) of the accelerator acc_i with a deadline of d_i. The time acc_i sets its *request* is represented by a_i and c_i is the remaining execution time for the task. The accelerator holding the *grant* will get a_i incremented and C_i decremented by one on each clock cycle to *keep its slack time* until it completes its transmission or gets preemted by an accelerator with higher priority. For all other accelerators with *request* set to one, their arrival time will be incremented by one, resulting in a lower slack time on each clock cycle. This algorithm is the most complex one as it needs to keep track of the remaining processing time, deadlines, and arrival time of new requests, implying a higher resource consumption compared to the previous algorithms.

7 Evaluation

The fulfillment of the *requirements* and how the *challenges* are solved with the *contributions* listed in Section 1 are analyzed in Section 7, through four different use cases. The schedulers are evaluated in Section 7.2.

7.1 Code Generation

7.1.1 Quaternion to Euler

This use case addresses the challenge of obtaining *all* the information (explicit and implicit) from the *system specification* (CH1). Listing 1 shows that with only 33 lines of code, the system depicted in Figure 1 can be generated and deployed (R1). It can be seen that the input and output signals of the *Quaternion to Euler Converter* have not been individually specified. They have been defined by their message type (Line 12 and Line 15 in Listing 1). This means that all the signals that constitute such messages are generated (CH3). Even though they have not been explicitly defined, they are derived by analyzing the message type. The information derived (*template configuration*) also includes the integration of the components shown in Figure 1 to the PS via DMA, which is where native ROS is running to communicate with

TABLE 1: Execution time of hardware accelerated functions.

Function	Software*[ms]	Hardware+[ms]	Speedup
Quaternion to Euler	0.012884	0.003730	3.45
Gray Scale conversion	801.45	62.20	12.9
Scale Down Nearest	381.95	20.73	18.4
Integral	212.22	20.73	10.2
Robotic Arm Kinematics	0.017	0.008	2.12

*Cortex-A9 running at 666 MHz — +HLS IPs running at 100 MHz

external nodes. Additionally, a *wizard* is provided to avoid manually writing the system specification but generate it interactively. This further reduces the possibility of making mistakes in such an error-prone process. The subscriber and publisher take 35 and 28 clock cycles, respectively. The Quaternion to Euler Converter takes 373 clock cycles. Therefore, the interfaces are not an overhead with respect to the time it takes to perform the computation (8%, 6%, and 86%, respectively). Table 1 shows the speedup obtained with the Quaternion to Euler conversion in HW with respect to SW (R3) running on the PS.

7.1.2 Image Processing

An image processing use case, consisting of pipelined functions (i.e., RGB to Grayscale, Downscaling, and Integral computation), was generated. Listing 2 shows a snippet of the *system specification* used, defining the interfaces for the accelerators (CH3), which are targeted to be in HLS. It also includes which elements of each interface is connected to where. Table 1 shows the execution time of each function. They take images with an input resolution of 1920x1080 (full HD) scaled down to 640x480. A speedup of 12.9x, 18.4x and 10.2x respectively was achieved. In this case, the length of the images (and therefore the resulting AXIS frame) can change. Therefore, the frame length is dynamically computed, as shown in Figure 3. The component to compute it is obtained following Algorithm 1 and it only consumes 48 Lookup Tables (LUTs), as it is a purely combinational logic. In this case, the *sensor_msgs/Image* does not contain nested arrays or messages, so there is no need to buffer any signals to wait for their sizes.

7.1.3 Multi-type messages

A system consisting of multiple *converters* for different types of messages, namely *sensor_msgs/Image*, *sensor_msgs/LaserScan* and *geometry_msgs/TwistStamped* was generated. Each set of converters (one *publisher* and one *subscriber* for each type of message) had a pass-through component in between (considered as the accelerator). The aim of this use case is the evaluation of the scheduling proposed in Section 4.3. On the SW side, three different callback queues were set. They received three types of ROS messages with different lengths at different frequencies. Depending on the dynamically changing priority list of the LRU scheduler, transactions between PS to PL occurred. The evaluation of the HW counterpart for all schedulers is shown in Section 7.2.

7.1.4 Robotic Arm position estimation

A system to compute the forward kinematics of a 7 Degrees of Freedom (DoF) robotics arm[2] was generated. This sort of computation becomes relatively complex and proportional to the amount of DoF. This is particularly important when performing motion control by generating a trajectory without colliding with objects. The accelerator is based on the desired and measured joint state values (q and q_d), and the measured and desired end-effector spatial matrices (O_T_EE and O_T_EE_d), read from the franka_msgs/FrankaState message. The outputs are the pose of each joint as fourteen spatial matrices (T1 to T7 and T1_d to T7_d, based on q and q_d), and the medium square error (T_mse) of the calculated spatial matrices concerning O_T_EE and O_T_EE_d. The reason why the Lines of Code (LoC) for the *Generated Artifacts* (Table 2) is so large is due to the extend of the *franka_msgs/FrankaState* message. However, this is not a concern when writing the *system specification* as it only requires to include the elements that contain the joint states as the input interface of the HLS accelerator to compute the kinematic equations. Table 1 shows a speedup of 2x compared to the SW execution, which would be beneficial to perform collision detection by knowing the position of each joint (spatial matrices) as soon as possible.

2. https://frankaemika.github.io

7.1.5 Manual Vs. Generated Deployment

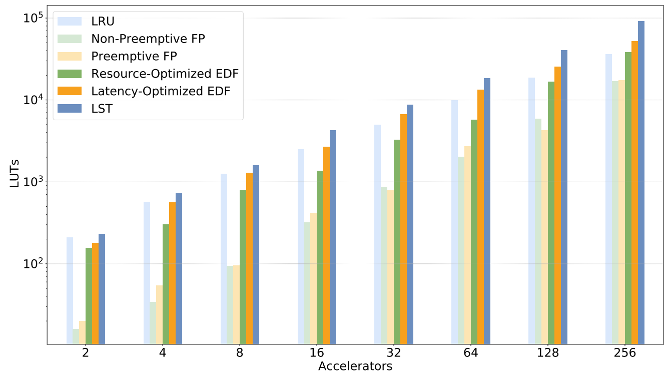
Table 2 compares the LoC that are manually written (or generated interactively via the wizard) of the *system specification* and of all *intermediate artifacts* for all use cases. Even though not all the artifacts would have to be manually written, the ratio between the LoC of the *system specification* and all the *intermediate artifacts* provides an indication of the effort for manual deployment in relation to the proposed workflow in this work. Evidently, the more complex the project becomes (more accelerators and converters, and more complex message specifications), the higher the effort. However, it is important to note that the effort escalates less when following our model-based approach than manual deployment.

7.2 Schedulers

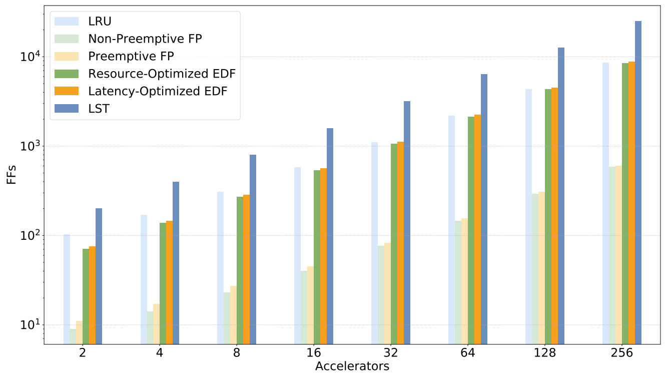
The accelerators competing to get a hold of the DMA have two parameters. One is the transfer time (T), in this case, representing the length of its payload to be streamed in bytes (one byte per clock cycle is transmitted). The other is the frequency (F), the number of clock cycles after its last transmission, and the availability of new data to be streamed. Each pair is called a set $S_i = \{T_i, F_i\}$, and the evaluation methodology followed for the schedulers consisted of a normal distribution for the generation of N sets for $M = \{2, 4, 8, 16, 128, 256\}$ accelerators. The N-sets constitute a dataset $D_M = \{(S_1, \sigma_1), \ldots, (S_N, \sigma_N)\}$, where σ is its standard deviation. Every algorithm is evaluated with the same dataset to understand the behavior of each scheduler for the same scenario. There are two types of exploration spaces (composed of the datasets). On the one hand, a large one with 200 sets, centered around $S_{large} = (\{100, 100\}, 50)$. Therefore, there will be evenly distributed sets between 50 and 150 for transfer time and frequency. This dataset gives a heterogeneous exploration space to have a general evaluation. On the other hand, the so-called "corner cases" are evaluated with four different datasets of ten sets each. They are centered around $S_{cc1} = (\{20, 20\}, 10)$, $S_{cc2} = (\{20, 180\}, 10)$, $S_{cc3} = (\{180, 20\}, 10)$ and $S_{cc4} = (\{180, 180\}, 10)$. These represent short and long transfer times and frequencies in extreme conditions, and as

TABLE 2: Lines of Code of Input vs. Generated Artifacts

Use Case	Input System Specification	Generated Artifacts: Template Configuration	Acc.Wrappers and Scripts	Converters and Scripts	System Components	Combined Artifacts	Generated to Input Ratio
Quaternion to Euler	35	99	22	459	102	682	19.48
Image Processing	83	136	34	692	107	969	11.67
Multi accelerator system	143	322	34	2320	172	2848	19.91
Robotic arm	45	1007	22	16540	307	17876	397.24



(a) Schedulers LUTs



(b) Schedulers FFs

Fig. 6: Schedulers' resource utilization

σ is small, there is homogeneity in these exploration spaces, focused on small areas around their centers.

The simulation time for the large dataset is $100us$ because the sets are heterogeneous enough, so it is a mix of large and short slack. The simulation time for the "corner cases" is $500us$ because when either the transfer time or frequency is large, there is less slack, so datasets with a large number of accelerators are preempted more (mainly the dynamic priority ones), so they need more time to complete their transactions. Hence, to have equal comparisons for all four corner cases, all of them have the same simulation time. These four ones do not require many datasets as their sets are homogeneous due to the small standard deviation. All simulations were performed at $100MHz$.

Different metrics are used to evaluate the proposed algorithm: scalability, schedulability, and performance.

7.2.1 Scalability

The design of the statechart and its derivations for the different scheduling algorithms is meant to rely only on LUTs and Flip-Flops (FFs). Figure 6 shows that both LUTs and FFs have linear behavior, which is desired when scaling up, so the resource consumption does not grow exponentially.

Table 3 shows the ratio for resource utilization between the two versions of EDF. It proves that, on average, the *ROEDF* algorithm shows a 5% reduction in FFs and a 40% decrease in LUTs. This proves a tradeoff between resource utilization and latency, as *ROEDF* consumes fewer resources but exhibits higher response time and lateness (as shown below) compared to *LOEDF*

7.2.2 Schedulability

The schedulability is studied to understand the abilities of the different algorithms to schedule tasks (to give accelerators the grant). It has a significant impact on the evaluation of the performance done below. The total number of accelerators that got the grant at least once are shown in Figure 8. These numbers can be further analyzed by distinguishing between accelerators that completed at least on transaction (full bars) and those that did not (striped bars) but still got the grant once. To further understand the schedulability, Figure 7 shows each algorithm's average preemptions per accelerator (per completed transactions).

There is a clear difference of LST to the other dynamic priority algorithms, as this one preempts accelerators at least four times more. The reason is that LST not only considers the time to the deadline but also takes into account when the request was set. Unlike EDF, this consideration greatly influences the slack and leads to more frequent priority updates, resulting in higher rate of accelerator preemption. These characteristics have significant implications, particularly when a large number of accelerators are present in the architecture (128 and 256). As a result, more accelerators get the grant. However, not all of them can complete their transactions in the simulation time set for the evaluation. It is worth noting that a longer simulation time would allow more accelerators to complete their transactions successfully. So, it is not a flaw of the scheduler but a restriction on the evaluation methodology [3].

The FP schedulers stand out in Figure 8 as they cannot give the grant to many accelerators, with a top of eight and just two can complete their transactions. This is expected as all accelerators have the same priorities during execution

3. The evaluation was done until 256, but it is not limited as larger values can be used.

TABLE 3: Resource-Latency EDF tradeoff

Accelerators	ROEDF/LOEDF LUTs	ROEDF/LOEDF FFs
2	0.87	0.95
4	0.54	0.95
8	0.62	0.95
16	0.51	0.96
32	0.49	0.96
64	0.43	0.95
128	0.66	0.96
256	0.73	0.97

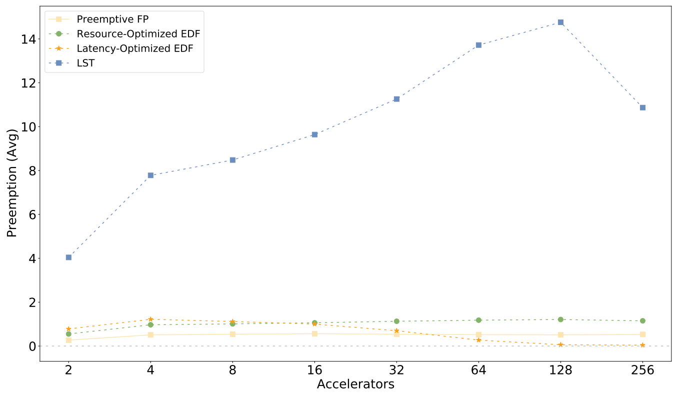


Fig. 7: Preemptions per accelerator (per completed transaction)

time, and the ones on the top of the priority list will be scheduled regularly. It can be seen in Figure 8 the differences between the two non-preemptive algorithms. The counterpart of the limitations of *NPFP* mentioned before can be seen with the LRU, with completed transactions for almost all accelerators that get the grant. However, as shown below, this has some drawbacks with its performance.

7.2.3 Performance

Multiple metrics are used to characterized the scheduling algorithms.

***Average response time*:** The *response time* (r_i) represents how long it took for an accelerator to get the *grant* since the moment it set its *request*. It is obtained with $r_i = g_i - a_i$, where g_i is the time when the grant is given, and $\mathrm{a_i}$ is the arrival time of the request. The *average response time* (r_{avg}) measured for n completed transactions, computed with Equation (1), is shown in Figure 10a.

$$r_{avg} = \sum_{i=0}^{n} \frac{r_i}{n} \tag{1}$$

It can be seen that both FP versions are the ones with the shortest response time, which would lead to think this is a good result. However, the schedulability of these two is the worst for all algorithms, as explained before, due to the small number of accelerators scheduled. As expected, LRU is the one with the worst results. This is not an issue as performance is not the main characteristic of this algorithm but to ensure accelerator schedulability. LST is the one that shows the best performance with the drawback that it takes a bit longer for all accelerators to complete their transactions. There is a clear difference between both EDF versions, being *LOEDF* the one with the shorter response time, approaching the same results as LST, but with the tradeoff of more significant resource consumption. ***Lateness*:** Denotes how much later than the deadline the data transmission was completed and is computed with $L_i = f_i - d_i$, where f_i is the time when accelerator acc_i finishes its transactions and d_i is its deadline. Negative lateness means the transmission was completed before its deadline.

The only requirement to measure the lateness is that accelerators must complete at least one transmission. Similar to the response time, the lateness (Figure 10b) shows that accelerators with both FP finish the transactions earlier, albeit at the cost of not scheduling a significant number of

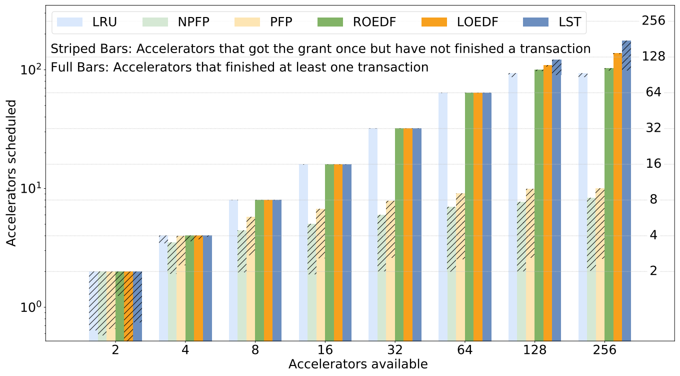


Fig. 8: Accelerators that completed at least one transaction or only got the grant and have not finished a transaction

them. Also, LRU has the largest lateness. For this metric, both EDF versions outperform LST since the latter one will preempt more accelerators, leading them to finish their transactions in a longer time. *LOEDF* shows better performance compared to *ROEDF* as intended. The reason for this is a shorter latency, which also translates to the smallest lateness for *LOEDF* among all dynamic priority schedulers.

The maximum lateness in any given system specification with multiple accelerators can be used to estimate the length of the buffers mentioned in Section 4.2.

***Communication Channel Utilization*:** This measurement quantifies the duration which any of the accelerators get the grant and transmits their data. To be fair with all schedulers, only the time which there are active requests is taken into account. Therefore, the utilization is measured according to Equation (2).

$$U = \sum_{i=0}^{M} \frac{acc_i}{t_{sim}} \tag{2}$$

where acc_i stands for the total time accelerator acc_i sets its request and was given the grant. t_{sim} stands for the total simulation time in clock cycles. The communication channel utilization measured is shown in Figure 9. Same as the other metrics, LRU is the one that performs the worst due to its design to avoid resource starvation, leading to long response time and lateness, which translates to less channel utilization. Both FP schedulers show a high communication channel utilization, but one has to keep in mind the low number of accelerators that are actually able to finish a transaction. However, as there is no time

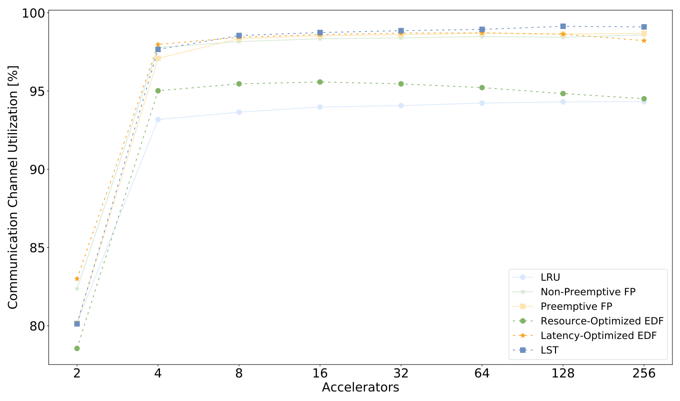
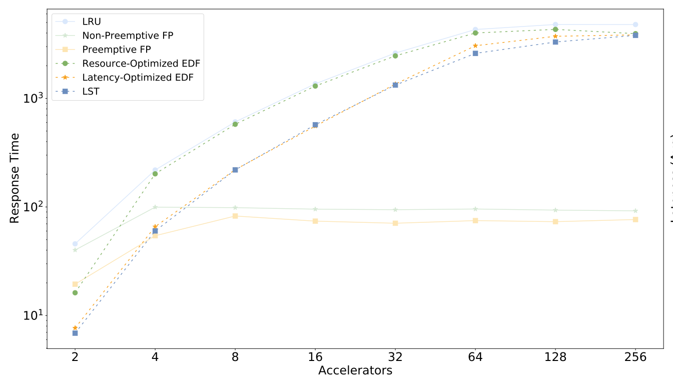
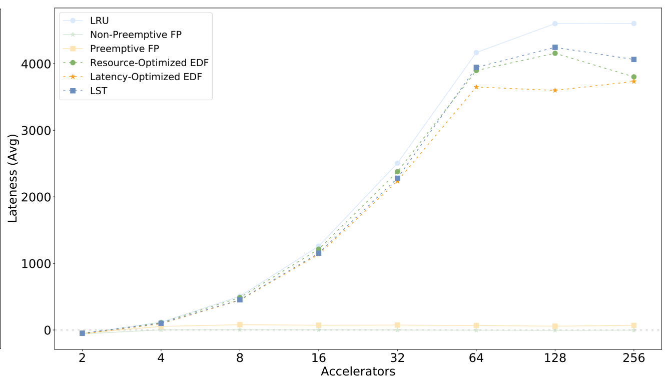


Fig. 9: Schedulers' communication channel utilization



(a) Average response time



(b) Lateness

Fig. 10: Schedulers' average response time and lateness

to update the priority table, this algorithm reacts fast to give the grant to accelerators. Regarding EDF, the resource-optimized version takes longer to give grants and also preempts the current accelerator holding the grant every time a new request arrives to recalculate the priorities, which translates into lower channel utilization compared to *LOEDF*. This directly impacts the channel utilization as accelerators can stream their data faster (in terms of when each can restart after being preempted). The last point is that as more accelerators get the grant with LST (Figure 8), the communication channel utilization is the highest for this algorithm.

7.2.4 Corner Cases

As mentioned previously, four cases with different transfer times and frequencies were evaluated to understand the behavior of the schedulers in these areas of the exploration space. The previously used metrics also help to understand the behavior of the algorithms for these cases.

Schedulability: In these four cases, the FP algorithms only schedule a low of number of accelerators as before. However, *all accelerators* for the dynamic priority schedulers finished a transaction at least once. The average preemptions per accelerator is impacted by the different transfer times and frequencies, as shown in Figure 11. In all cases, LST continues to be the algorithm that preempts most of the accelerators, and the preemptions increase significantly with larger transfer times, regardless of their frequency. This is clear because each accelerator requires to have the grant for more time to finish a transaction which causes more preemptions. Moreover, these four datasets have a small σ. Therefore, the possibility for *laxity ties* (two or more accelerators with the same priority constantly preempting each) is high.

Performance: The transfer time of the accelerators affects the response time, increasing it with higher values, as shown in Figure 12. It is possible to see that the response time increases by one order of magnitude in the cases with the largest transfer time. Previously, *LOEDF* and *LST* had similar performance. Here, for short transfer times, it is actually *LOEDF* with a shorter response time (as opposed to LST in Figure 10a), same as for long transfer times but up to a certain number of accelerators. When more than 64 are present, *LST* has a lower response time, making it a better candidate for this situation. The lateness is affected by the shortest period, as it takes longer for the accelerators to complete their transactions, either when their frequency is short or long (Figure 13a and Figure 13b). Figure 13c and Figure 13d depict the worst-case scenario, when the transfer time is the longest, meaning that it takes significantly more time (one order of magnitude) to finish. Note how LST diverges from the other accelerators after 128 accelerators due to the significant increase of preemptions at this point.

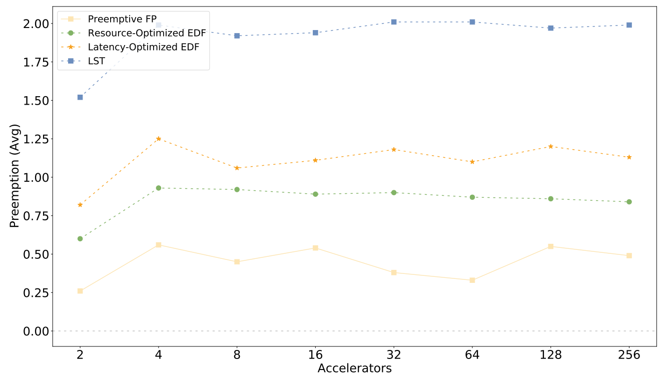
As for LRU, it is the algorithm with the worst performance for these corner cases because its goal is to make sure that all accelerators can finish their transactions at least once.

7.2.5 Combining Schedulers

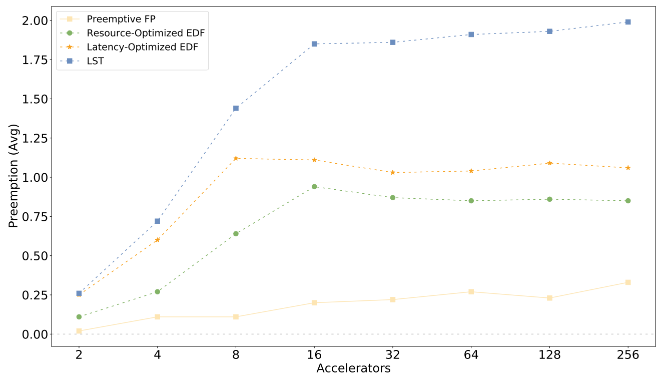
It is possibile to improve the schedulability by smartly combining different schedulers. A dataset of 256 accelerators served as the baseline to evaluate how many get the grant with the two most promising schedulers, namely *LOEDF* and *LST*. Two different cases are studied. The first consists of splitting the accelerators into smaller datasets, in this case, dividing one large scheduler with 256 accelerators into *two of the same one* but with 128 accelerators each. The second one also splits into a smaller number of accelerators per scheduler, but with *two different algorithms*. All these require a third scheduler also to manage the new smaller ones. *LRU* is chosen for this study as it ensures that all requests get a grant. Results are shown in Figure 14. Splitting them does not increase the number of accelerators that got the grant for *LOEDF* but increases 1.16x for *LST*.

Combining schedulers resulted better for *LOEDF* (1.29x) as it was done with *LST*, which showed better schedulability. However, in the *LST* case, combining it with *LOEDF* was, in fact, detrimental. Note that every combination of schedulers is possible. However, it does not guarantee improved schedulability as the decision on which schedulers to pick for the best result should be done following a similar design exploration as shown above.

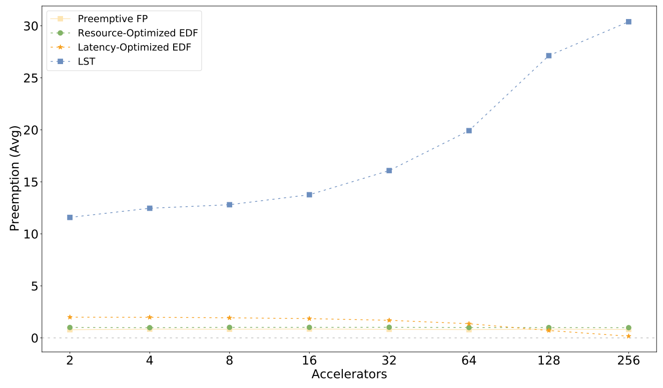
Combining schedulers proved to be beneficla for *LOEDF* (1.29x improvement) enhancing its schedulability. However, in the case of *LST* , combining it with *LOEDF* had a negative impact. Note that while every combination of schedulers is possible, it does not guarantee improved schedulability. The decision of selecting the appropriate schedulers for optimal results should be made through a thorough design exploration, similarly as shown previously.



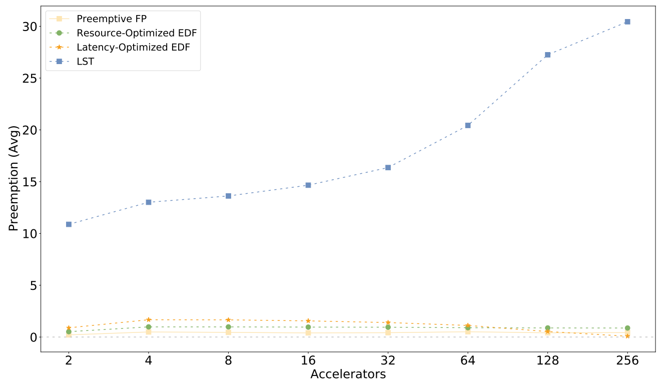
(a) Transfer Time=20 - Frequency=20



(b) Transfer Time=20 - Frequency=170

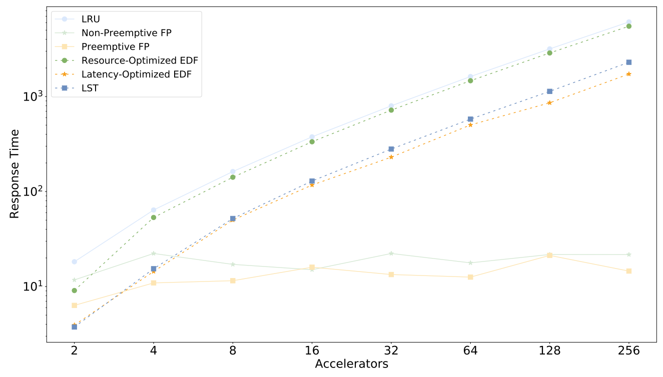


(c) Transfer Time=170 - Frequency=20

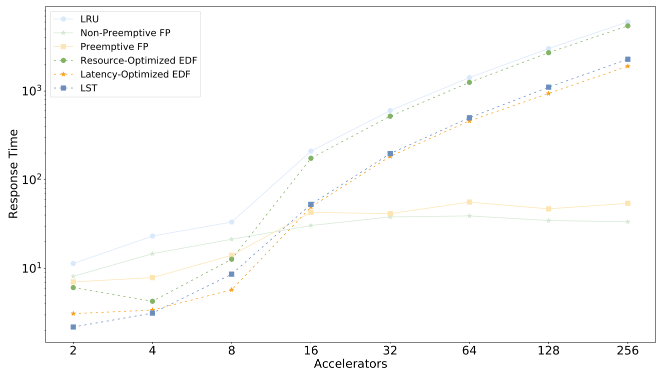


(d) Transfer Time=170 - Frequency=170

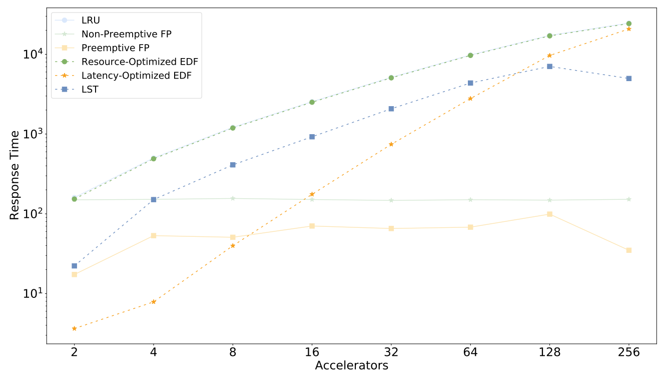
Fig. 11: Schedulers' corner cases: Average Preemption per algorithm



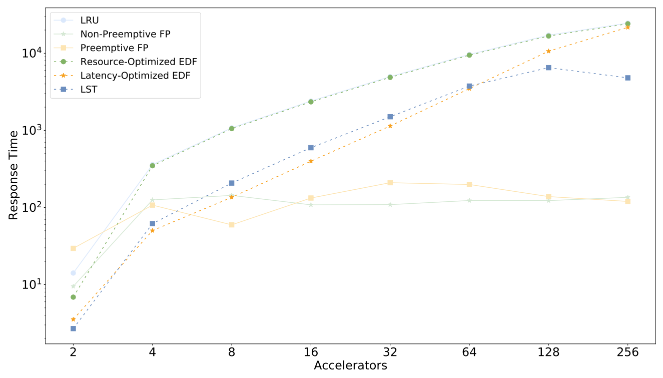
(a) Transfer Time=20 - Frequency=20



(b) Transfer Time=20 - Frequency=170

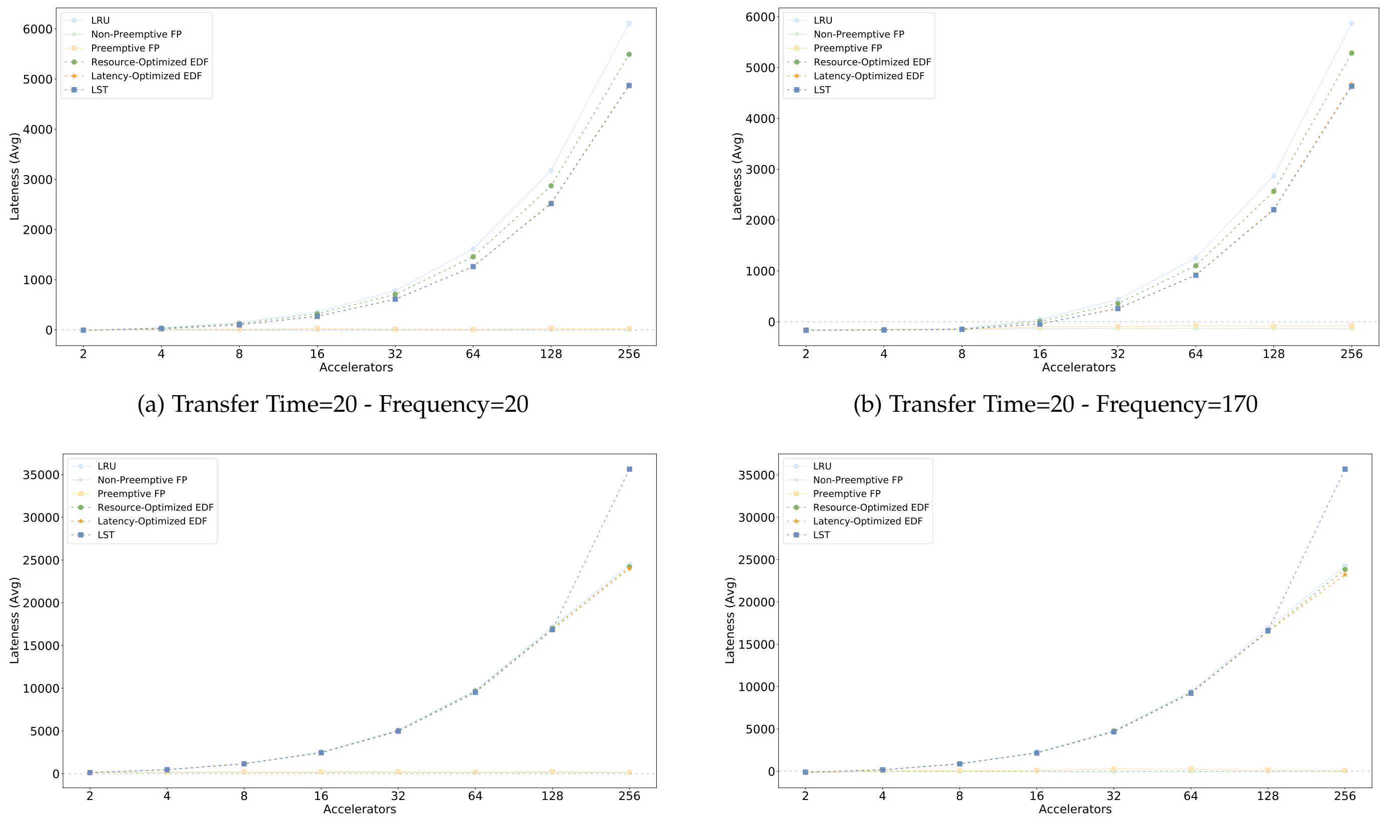


(c) Transfer Time=170 - Frequency=20



(d) Transfer Time=170 - Frequency=170

Fig. 12: Schedulers' corner cases: Response time



(a) Transfer Time=20 - Frequency=20

(b) Transfer Time=20 - Frequency=170

(c) Transfer Time=170 - Frequency=20

(d) Transfer Time=170 - Frequency=170

Fig. 13: Schedulers' corner cases: Lateness

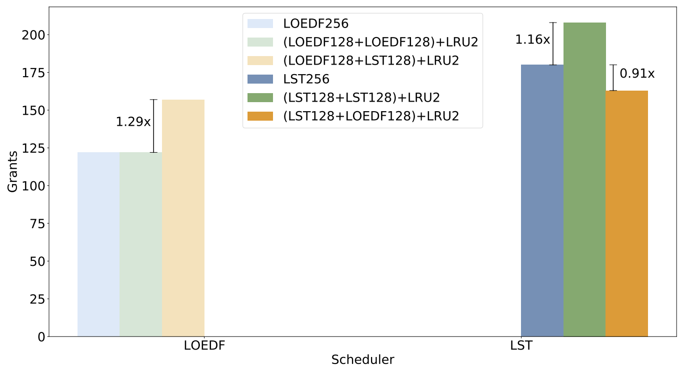


Fig. 14: Combined schedulers

8 Conclusion and Future Work

This work presents a model-based approach to generate FPGA-based architectures for robotics applications. Our systematic approach circumvents the arduous process that involves designing complex architectures and the modifications of HW accelerators needed to incorporate them in existing solutions. For this, a toolchain which takes a concise and expressive *system specification* is introduced. It derives a *holistic model* of the desired system and generates all needed components. The advantages of the systematic procedure are shown with different use cases. Six different scheduling algorithms are proposed to deal with a hybrid HW/SW architecture, and the framework used for evaluating them can be used as part of the toolchain to evaluate which algorithm would fit best each system by performing design space exploration. It is left for future work to dynamically change the scheduling algorithms with Dynamic Partial Reconfiguration (DPR) based on the current workload.

References

[1] A. Kumar, "Methods and Materials for Smart Manufacturing: Additive Manufacturing, Internet of Things, Flexible Sensors and Soft Robotics," *Journar Manufacturing Letters*, vol. 15, pp. 122–125, 2018.

[2] A. C. Simões, A. L. Soares, and A. C. Barros, "Factors influencing the intention of managers to adopt collaborative robots (cobots) in manufacturing organizations," *Journal Engineering and Technology Management*, vol. 57, p. 101574, 2020.

[3] F. Soto and R. Chrostowski, "Frontiers of medical micro/-nanorobotics: In vivo applications and commercialization perspectives toward clinical uses," *Journal Frontiers in bio-engineering and biotechnology*, vol. 6, p. 170, 2018.

[4] U. R. Mogili and B. Deepak, "Review on application of drone systems in precision agriculture," *Journal Procedia Computer Science*, vol. 133, pp. 502–509, 2018.

[5] M. B. Alatise and G. P. Hancke, "A review on challenges of autonomous mobile robot and sensor fusion methods," *Journal IEEE Access*, vol. 8, pp. 39830–39846, 2020.

[6] E. Coste-Maniere and R. Simmons, "Architecture, the backbone of robotic systems," in *Proc. Int. Conf. on Robotics and Automation. Symposia Proceedings*, vol. 1, pp. 67–72, IEEE, 2000.

[7] 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. Int. Conf. on Embedded Software and Systems (ICESS)*, pp. 1–8, IEEE, 2019.

[8] 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. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pp. 1–9, IEEE, 2016.

[9] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *Journal IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 326–342, 2018.

[10] Podlubne et al., "Model-based Generation of Hardware/Software Architectures for Robotics Systems," in *Proc. Int. Conf. on Field-Programmable Logic and Applications (FPL)*, 2022.

[11] K. Yamashina, T. Ohkawa, K. Ootsu, and T. Yokota, "Proposal of ROS-compliant FPGA Component for Low-Power Robotic Systems," *CoRR*, vol. abs/1508.07123, 2015.

[12] K. Yamashina, H. Kimura, T. Ohkawa, K. Ootsu, and T. Yokota, "cReComp: Automated Design Tool for ROS-Compliant FPGA Component," in *Proc. Int. Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pp. 138–145, IEEE, 2016.

[13] Sugata et al., "Acceleration of Publish/Subscribe Messaging in ROS-compliant FPGA Component," in *Proc. Int. Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*, pp. 1–6, 2017.

[14] 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. Int. Workshop on Robotics Software Engineering (RoSE)*, 2019.

[15] X. Shi, L. Cao, D. Wang, L. Liu, G. You, S. Liu, and C. Wang, "Hero: Accelerating autonomous robotic tasks with FPGA," in *Proc. Int. Conf. on Intelligent Robots and Systems (IROS)*, pp. 7766–7772, IEEE, 2018.

[16] C. Lienen and M. Platzner, "Design of distributed reconfigurable robotics systems with reconros," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 3, pp. 1–20, 2021.

[17] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An operating system approach for reconfigurable computing," *Journal IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2013.

[18] C. Lienen and M. Platzner, "ReconROS Executor: Event-Driven Programming of FPGA-accelerated ROS 2 Applications," *arXiv preprint arXiv:2201.07454*, 2022.

[19] A. Podlubne, J. Mey, R. Schöne, U. Aßmann, and D. Göhringer, "Model-based approach for automatic generation of hardware architectures for robotics," *IEEE Access*, vol. 9, pp. 140921–140937, 2021.

[20] A. Al-Zoubi, K. Tatas, and C. Kyriacou, "Towards dynamic multi-task schedulling of opencl programs on emerging cpu-gpu-fpga heterogeneous platforms: A fuzzy logic approach," in *Proc. Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, pp. 247–250, IEEE, 2018.

[21] P. Du, Z. Sun, H. Zhang, and H. Ma, "Feature-aware task scheduling on cpu-fpga heterogeneous platforms," in *Proc. Int. Conf. on High Performance Computing and Communications*, pp. 534–541, IEEE, 2019.

[22] Z. Zhu, J. Zhang, J. Zhao, J. Cao, D. Zhao, G. Jia, and Q. Meng, "A hardware and software task-scheduling framework based on cpu+ fpga heterogeneous architecture in edge computing," *IEEE Access*, vol. 7, pp. 148975–148988, 2019.

[23] A. Vaishnav, K. D. Pham, and D. Koch, "Heterogeneous resource-elastic scheduling for cpu+ fpga architectures," in *Proc. Int. Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*, pp. 1–6, 2019.

[24] S. Soldavini and C. Pilato, "A survey on domain-specific memory architectures," *arXiv preprint arXiv:2108.08672*, 2021.

[25] D. Schmidt, "Model-driven engineering," *Journal Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[26] T. Stahl, M. Völter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2006.

[27] D. E. Knuth, "Semantics of context-free languages," *Journal Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968.

[28] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper, "Higher order attribute grammars," in *ACM SIGPLAN Notices*, vol. 24, pp. 131–145, ACM, 1989.

[29] G. Hedin, "Reference attributed grammars," *Journal Informatica (Slovenia)*, vol. 24, no. 3, pp. 301–317, 2000.

[30] J. Mey, R. Schöne, G. Hedin, E. Söderberg, T. Kühn, N. Fors, J. Öqvist, and U. Aßmann, "Relational reference attribute grammars: Improving continuous model validation," *Journal Computer Languages*, vol. 57, 04 2020.

[31] E. Nilsson-Nyman, G. Hedin, E. Magnusson, and T. Ekman, "Declarative intraprocedural flow analysis of java source code," *Electronic Notes in Theoretical Computer Science*, vol. 238, no. 5, pp. 155–171, 2009. Proceedings of the 8th Workshop on Language Descriptions, Tools and Applications (LDTA).

Ariel Podlubne is a research assistant and PhD student at the Chair of Adaptive Dynamic Systems at Technische Universität Dresden. He worked for two years at LAAS/CNRS in Toulouse France from june 2014 for the european project Two!Ears developing the robotics testbed used for experiment. His research interests are FPGA-based architectures for signal and image processing and robotics.

Johannes Mey is a research assistant and PhD student at the Chair of Software Technology at Technische Universität Dresden. His research focuses on reference attribute grammars an their application in various fields. Besides static program analysis, these include model transformations and adaptive software systems. Furthermore, he investigates the relation between reference attribute grammars and conceptual models.

Andreas Andreou received his B.Sc. in Computer Science from the Technische Universität Dresden. He works as an FPGA Design developer for video data processing at Fraunhofer FEP Insitute. He is interested in robotics and hardware-related programming.

Sergio Pertuz received his PhD and M.Sc. in Mechatronic Systems from the University of Brasilia. Before joining the Chair of Adaptive Dynamic Systems (ADS) as postdoctoral researcher, Sergio was a full time teacher at the University of Brasilia and an appraiser for innovation projects at the FINATEC Foundation. He has experience in digital circuit prototyping using reconfigurable architectures, hardware/software co-design, signal processing, and AI (ANN and swarm intelligence) applied to automation and robotics applications integrating.

Uwe Aßmann holds the Chair of software technology at Technische Universität Dresden. He has obtained a PhD in compiler optimization and a habilitation on *invasive software composition* (ISC), a composition technology for code fragments enabling flexible software reuse. ISC unifies generic, connector-, view-, and aspect-based programming for arbitrary program or modelling languages. Since 2013, he is deputy of the DFG Research Training Group *Role-oriented Software Infrastructures* (RoSI), which develops new techniques for context-adaptive software, from language and application design to run time (rosi-project.org).

Diana Göhringer holds the Chair of Adaptive Dynamic Systems at Technische Universität Dresden since 2017. In 2011, she received her PhD(summa cum laude) in Electrical Engineering and Information Technology from the Karlsruhe Institute of Technology (KIT), Germany. From 2013 to 2017 she was an assistant professor and head of the MCA (Application-Specific Multi-Core Architectures) research group at Ruhr-University Bochum (RUB), Germany. She is author and co-author of over 150 publications in international journals, conferences and workshops. Additionally, she serves as technical program committee member in several international conferences and workshops (e.g. DATE, FPL, FPT, ICCAD). She is reviewer and guest editor of several international journals. Her research interests are reconfigurable computing, multiprocessor systems-on-chip, networks-on-chip, hardware-software-codesign and runtime systems.