Runtime Abstraction for Autonomous Adaptive Systems on Reconfigurable Hardware

Alex R. Bucknall* and Suhaib A. Fahmy†*
*School of Engineering, University of Warwick, Coventry, UK
†King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia
Email: a.r.bucknall@warwick.ac.uk

***Abstract*—Autonomous systems increasingly rely on on-board computation to avoid the latency overheads of offloading to more powerful remote computing. This requires the integration of hardware accelerators to handle the complex computations demanded by data-intensive sensors. FPGAs offer hardware acceleration with ample flexibility and interfacing capabilities when paired with general purpose processors, with the ability to reconfigure at runtime using partial reconfiguration (PR). Managing dynamic hardware is complex and has been left to designers to address in an ad-hoc manner, without first-class integration in autonomous software frameworks. This paper presents an abstracted runtime for managing adaptation of FPGA accelerators, including PR and parametric changes, that presents as a typical interface used in autonomous software systems. We present a demonstration using the Robot Operating System (ROS), showing negligible latency overhead as a result of the abstraction.**

I. Introduction

Autonomous adaptive systems modify their behaviour under unknown scenarios by monitoring and processing external stimuli and applying decision logic to determine their response. Systems such as unmanned aerial vehicles [1], communication systems [2] and self-driving vehicles [3], [4] must rapidly adapt to external events using information gathered from high data rate sensors. Such sensors typically require post-processing for the large volumes of streamed data generated as the external stimuli is monitored, such as with LiDAR [5]. This can be problematic when implemented on traditional general purpose processor based systems that share computing resources between processing sensor data and cognitive decision functions. Many data-intensive sensors demand complex signal processing that is amenable to hardware acceleration through parallelisation, and in some cases such computation may be offloaded to custom application specific integrated circuits (ASICs) for this purpose. However, as hardware becomes more complex due to higher data rates and a wider variety of sensors, and as machine learning applications emerge [6], fixed ASIC accelerators become problematic due to their lack of flexibility.

Field Programmable Gate Arrays (FPGAs) offer application specific hardware acceleration while maintaining computational flexibility through reconfiguration. They have seen widespread use in the acceleration of image processing [7], wireless communications [8], and machine learning [9], with significant performance and energy benefits against CPU and GPU implementations [10].

While accelerating low level processing of sensor data is beneficial, the cognitive decision logic in adaptive systems typically involves complex high level algorithms that interact with scheduled event based operations. This type of computation is more appropriately suited for software implementation on general purpose processors, typically implemented on top of an operating system (OS) to offer wider flexibility and programmability. Hence, it is challenging to manage the low level software to hardware interface in hybrid adaptive systems with an abstraction that can cross the boundary between cognitive algorithms and data processing accelerators. Applications such as flight controllers for autonomous drones require time sensitive communication between a low latency processing system to control actuators but may rely on an operating system such as Linux for course navigation, networking, and decision making [11]. While extensive contributions have been made in autonomous software frameworks for CPSs [12], limited research has considered their coupling with hardware acceleration.

In this paper, we consider hybrid FPGA systems-on-chip (SoCs) that tightly couple programmable logic (PL) with a general purpose processing system (PS), such as the Xilinx Zynq and Zynq Ultrascale+ (ZynqMP) SoC families, as platforms for implementing such hybrid autonomous systems. Our contributions in this paper are as follows:

- An abstracted Linux configuration manager, built on an adaptive systems model to automate reconfigurable hardware management and allow control from a high level programming interface.
- An extension to custom partial reconfiguration design tools to generate PR bitstreams, software drivers and abstract symbols based on configuration specifications.

II. Background and Related Work

CPSs have gained much interest in literature over the years, with various definitions of hierarchy and structure; we begin by defining relevant concepts and terminology.

A. Adaptive System Concepts

A fundamental model of autonomous adaptive systems defines the software operation in a closed loop of 3 tasks; *Observation*, *Decision*, and *Action* [13]. Observation is defined as the (post) processing of sensor data, such as feature extraction from an image processing event and can be considered

978-3-9819263-5-4/DATE21/©2021 EDAA

as the extraction of key data points produced from sensors. The decision task uses these data points to generate a *next* action, which is typically determined by a high level decision algorithm. This task is the main cognitive element of the loop, where data evaluated in the observation task is used, along with historical data, to form learned behaviours and generate informed decisions. Finally, the *action* task propagates the system changes, determined in the decision task, such as adjusting internal memory registers, changing sensor settings or triggering actuators. This could also be instructing a hardware configuration manager of a desired configuration or triggering an actuator/sensor change, as shown in Fig. 1. The model in [14] extends this concept further into the Observe-Normalise-Compare-Learn/Reason-Decide-Act loop, where the additional tasks take a more deliberate approach to constructive feedback, aiming to build a sustained model for the adaptation processes. The process of adapting hardware should be managed independently from the hardware acceleration itself, to limit performance impact on high speed, real-time sensor-actuator processing [15].

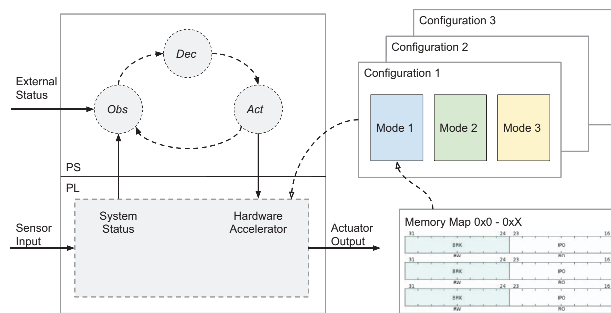

Fig. 1. Visualisation of hardware abstraction and definition.

In [12], the authors survey a number of CPS frameworks proposed in the literature. Many of these refer to collections of heterogeneous objects distributed across a network [16], which provides a useful model for conceptualising how such nodes interact. Frameworks such as ROS [17] build upon these concepts to provide a structured communication layer that supports heterogeneous clusters. We take inspiration from the ROS operating model for our configuration manager.

B. *Configuration Terminology*

We define terminology in the context of a tightly coupled software/hardware CPS. Individual hardware components can exist in a set of possible *states*, each of which might adjust some internal hardware registers (a *parametric* change), or force a hardware reconfiguration with a new circuit (a *structural* change). Combined together, multiple components form a valid *mode* of the system that can be set by the cognitive decision logic. In this way, it is shielded from managing the low-level *states* of individual components. In some cases, the fundamental hardware structure may change through modification of access to specific sensors or actuators (such as a radio switching from sensing to communication modes). We refer to these as distinct hardware *configurations*, potentially requiring a different set of data interfaces between software and hardware. During runtime operation, the decision logic communicates *configuration* changes to the hardware through a *configuration manager* (CM) which abstracts the underlying changes to hardware required for the desired *configuration* and *mode*. The CM is responsible for abstracting the software to hardware interface with an application programming interface (API).

C. *FPGA Acceleration of Adaptive Systems*

Formal adaptive system frameworks are defined in the literature [18] but commonly consider software-only CPSs that lack directly coupled hardware acceleration. Typically, CPSs that utilise hardware offload are tightly integrated between low-level software and hardware control so the design complexity and requirement for extensive FPGA knowledge limits wide spread adoption [15]. This presents significant design challenges for autonomous systems that wish to exploit the capabilities of reconfigurable hardware within a software programmable framework, such as partial reconfiguration in FPGAs. PR is accomplished by defining a static region of functionality, fixed at runtime, and one or more partially reconfigurable regions (PRRs) that can host different hardware modules, interchangeable at runtime. The static region typically contains fixed components such as reconfiguration controllers as well as infrastructure like the processing subsystems and DMA controllers. The PRRs can be loaded at runtime with modules that have been compiled into PR bitstreams to define an instance of hardware logic for that specific PRR. Current research in this area has focused on improving vendor tooling [19] and increasing performance [20], rather than control abstractions. Work such as [21] attempts to virtualise access to PR through the use of shells in the static region that standardise hardware interfaces, providing a more generic means of utilising hardware. Virtualisation helps to reduce the complexity of hardware but it also forces PR designs to conform to fixed configurations, which define software to hardware interfaces and are decided on by the shell provider. This suits general accelerator platforms but not sensor-rich autonomous systems with complex peripheral interfaces.

The current vendor PR design flow is plagued with the need for prerequisite FPGA knowledge and understanding of the convoluted build process. We consider the Xilinx toolchain for reference, however the experience is similar across other toolchains such as Intel Quartus. To begin a PR design, the designer must decide on the number and location PRRs within the PL. This should be done considering a number of factors; a single PRR could be used for simplicity or multiple PRRs to allow for each module to be exchanged independently and thus reducing the number of required reconfiguration events. Using multiple PRRs forces each region to be sized according the requirements of individual hardware modules, while a single PRR can be sized to the largest union of supported modules. Reconfiguration latency is dependent on PR bitstream size, which itself is dependent on PRR area. Grouping multiple

modules into a PRR means it must be reconfigured multiple times per module. Vendor tools restrict PR module generation to a specific static region, as netlists and physical interfaces must align, meaning PRRs should be generated with the original static regions, including new bitstreams generated after the initial build of a platform, limiting flexibility.

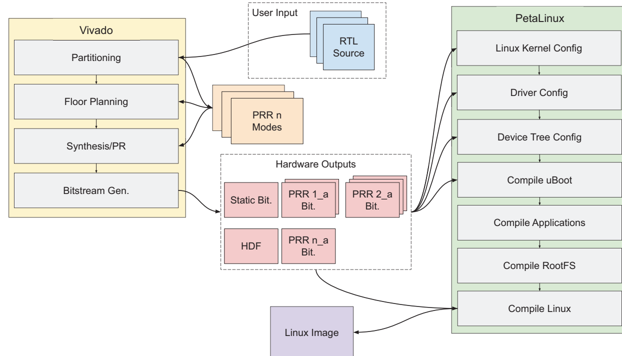

Fig. 2. An outline of the Xilinx PR build flow, from generating hardware within Vivado, to exporting hardware data into PetaLinux.

This development process requires complex sequential steps in order to generate hardware, before even exporting into the Xilinx's Linux build process, PetaLinux. The hardware definition files (HDF) generated from the Vivado toolflow, outlined in Fig. 2, are required for PetaLinux to compile a Linux image. PetaLinux is used to build the device tree (DT) and kernel drivers required to communicate with the PL from Linux. The setup process for a standard flow involves importing the HDF and bitstreams then manually configuring kernel parameters, enabling device drivers and potentially injecting custom drivers and/or applications into the build. While PetaLinux offers a build system for some automation, this does not generate from PR logic and is unable to determine drivers for PR.

Xilinx attempts to abstract the PS to PL interfaces with their Linux distribution, PYNQ [22]. PYNQ is a Python abstraction for controlling the PL from the PS and supports PR as well as isolating the kernel recompilation requirements for loading new bitstreams. While PYNQ does abstract hardware interaction, it relies on the standard vendor build flow and binds the user to a Python framework, adding further layers of software to their application, making it unsuitable for low-latency systems. Additionally, there is no concept of configurations or modes and the user is left to manage this manually.

Abstraction frameworks such as [23] use loadable kernel modules wrapped within a high level threaded API to load/unload hardware accelerators at runtime. While this abstraction is suitable for wrapping low level interfacing, it still requires the user to have extensive knowledge of the hardware and how to control/manage it. Furthermore, this framework does not implement any build time abstractions for hardware; it is bespoke and is implemented directly on an FPGA using a soft-processor as opposed to a hard processor, as used on the Xilinx Zynq. ReconOS [24] is a build and runtime framework that takes custom hardware and software libraries and generates a reconfigurable OS centred around a POSIX API for thread-based acceleration control. Their framework demonstrates a tight interface between hardware and software but provides limited abstraction for state or configuration control of the system. Given that PR modules must be compiled with their libraries and drivers, this creates a distinct architecture and set of interfaces that users must adhere to. CoPR [25] provides a number of the missing elements to reduce the complexity, such as generating PRRs based upon input configurations specified by the designer. However it does not support configurations under a full operating system and lacks support for Linux and the Zynq Ultrascale+ architecture. More recently, FOS [19] decouples the build stages so that they can be generated independently. This reduces design complexity, however, it does not automatically generate runtime configurations from the underlying hardware or abstract control for data streaming into/out of hardware.

III. Architecture

We have designed a Linux configuration management runtime that offers a software abstraction for managing programmable logic configurations, by masking the complexities of structural reconfiguration (such as PR), parametric changes and hardware addressing. This runtime provides a generic API, over a network capable protocol, for software frameworks to interact with hardware acceleration available to the processing system.

A. Adaptive Hardware Design Tooling

We use a collection of existing build tools [26], which extend the vendor PR build flow with the inclusion of user defined build schemas, representing the user's desired PR modules and arrangements. The build schema is a config file that defines which hardware description language (HDL) source files, such as Verilog or Xilinx IP cores, should be used to generate PR modules as well as any custom logic that defines the static region. This file constrains the combinations of valid modules to modes that can be loaded, for the implemented combinations. Based upon this build schema, the tool is able to extract known interfaces (typically AXI(s) as well as additional user defined protocols) from the PRRs to generate infrastructure for communication between partial and static regions. It then produces the bitstreams for the PRRs, which are synthesised and implemented using the vendor tools. These bitstreams are then exported to a Linux build flow, where the design framework uses associated HDFs and meta-data, to create the kernel configs, PR specific DT overlays and inject any custom drivers required to support PR hardware into the kernel. We use outputs produced by the build tools to generate higher level abstraction schemas that are used to load the state of hardware and software according to the configurations applied.

B. Runtime Configuration Schemas

In order for the CM to infer the structure of hardware in the PR at a given configuration, it uses a runtime schema generated upon completion of the Vivado and PetaLinux builds. We generate a template config schema for each possible configuration of PRRs (constrained prior to building), prompt the user for changes and make this read/writeable to the Linux userspace. It includes the modules provisioned in the PRRs, register addresses (and default values) to be set under Linux's memory mapped I/O (MMIO) interface and any device tree overlays (DTO), which also contain drivers for described hardware. Full configuration change of hardware may also be specified through the schema, allowing for new static regions and thus more PR modes, although compatible DTOs will need to be generated and managed out-of-tree.

Listing 1 shows an example output file from the build tool with two template modes for the camera, default and edge detection. We chose the YAML format as the user is expected to modify these files to define their desired modes, thus requiring the files to be human readable as well as consumable by the CM. For example, upon loading the *edge* mode, the CM understands that this means loading the partial bitstream *edge.bit*, adjusting the memory map to cover the *edge* register and leaving the DT overlay unchanged from the *default* mode.

```yaml
1  config:
2    - name: camera
3    - modules: [default,edge,colorise]
4    - type: partial
5    - modes:
6        edge_detection:
7          - modules: [edge]
8          - mmio:
9            - base_address: 0x2000
10           - registers:
11             - focus:
12                 - offset: 0x1010
13                 - default: 0x01
14         - overlay:
15           - camera
16       default:
17         - modules: [default]
18         - overlay:
19           - camera
```

Listing 1. Configuration Specified in YAML, produced by the build tool.

Additional schemas can be added to the tool post-build, given they are compatible with PRR interfaces and generated from the matching static region.

C. Configuration Manager

The CM is designed to be available within the Linux userspace, regardless of the programming interface/framework attempting to consume/control it. We use ZeroMQ [27], a lightweight asynchronous messaging library, used for both internal and external networking. Alternative communication libraries such as OpenDDS [28] could be substituted for ZeroMQ, which we selected for its lighter-weight implementation, support for multiple transport methods and programming languages as well as lower latency.

We use shared memory via Linux's inter-process communication (IPC) transport for low latency, however the CM can also use external IP transport methods, such as TCP. ZeroMQ is a publisher/subscriber (pubsub) protocol in which processes that publish/subscribe to data are known as *nodes* and share data over subscribable *topics*. The protocol is important as it decouples data producing nodes from nodes that process data, enabling multiple nodes within the system to subscribe to CM changes and thus monitor the state of the hardware. This allows other processes running inside of the PS, such as system health checks or networking interfaces to also be aware of hardware changes. ZeroMQ is message protocol agnostic, allowing us to use protobufs [29] as our messaging protocol and as raw binary for directly streaming data into and out of hardware. Protobufs are a serializing method for structured statically typed data, used to define the API interface.

We use userspace-to-kernel drivers to provide application access to hardware memory buffers. The runtime is hosted in the userspace to allow for greater flexibility of libraries and reduce security risks of asking user code to interface directly with hardware. To communicate using MMIO, we use the Userspace IO (UIO) kernel module, which allows for the mapping of configuration memory addresses and interrupts within the PL up to the PS. For streaming data between hardware, we use a zero-copy kernel driver and userspace wrapper for the Xilinx AXI DMA interface [30] (AXIDMA). This driver supports both DMA and VDMA as well as allocation of contiguous buffers through the Linux kernel's contiguous memory allocator (CMA), which allows for streaming data between userspace and hardware. While the build tool does allow for custom drivers to be used, our CM API wraps UIO and AXIDMA modules for controlling hardware as they sufficiently provide read/write to AXI and AXI Stream interfaces from the ZeroMQ interface.

To manage structural reconfiguration we use a custom PR manager [26] that allows for provisioning via the high speed internal configuration access port (ICAP) reconfiguration interface using DMA, on both Zynq and Zynq Ultrascale+ devices. This manager supports asynchronous provisioning, meaning the PS is not blocked from processing during PR.

D. Configuration API

We expose an API on a selection of topics, available over ZeroMQ. A sample list of topics used in the framework is shown below, along with a pseudo code example in Listing 2.

- `cm_status` – publishes the state of the PL as a struct containing values such as `string:config_name`, `bool:fpga_busy` and `enum:pr_mode`.
- `cm_(read/write)_config` – publish/subscribe to the config that is currently loaded. This topic can also receive a config file and provision it to hardware.
- `cm_(read/write)_reg` – allows reading/writing to MMIO as defined by names in the config file for PR.
- `cm_data_config` – allows configuration of the AXI, AXI Lite, and AXI stream channels of the PS-PL interface.
- `cm_(read/write)_data` – allows reading/writing to DMA buffer for reading/writing to hardware over DMA.

```
1 def act(dec_result):
2     if dec_result == "default":
3         # Do nothing
4         pass
5     elif dec_result == "edge_detect":
6         # Swap to edge detection mode
7         zeromq_publish("cm_config", "edge")
8     elif dec_result == "default_zoom":
9         # Request a register write to zoom camera
10        zeromq_publish("cm_write_reg", "{'focus':0
    x2}")
11 # Continuously read from cm_data_read topic
12 obs_buffer = zeromq_subscribe("cm_data_read")
13 while True:
14     # Request from observation thread
15     dec_buffer = obs(obs_buffer)
16     # Request from decision thread
17     dec_result = dec(dec_buffer)
18     # Request to action thread
19     act(dec_result)
```

Listing 2. Pseudo example of congitive engine controlling the CM.

IV. Demonstration

We provide a demonstration of our CM by implementing it alongside the ROS2 framework to highlight the minimal effect on latency and performance introduced with our abstraction.

A. Robot Operating System

ROS, designed by Willow Garage and maintained by the Open Source Robotics Foundation, is a communication framework designed to streamline networking across robotics platforms using standardized schemas and messaging interfaces. This later became ROS2 [31], which introduced the Data Distribution Service (DDS) and added features such as quality of service, security improvements, flexibility, and robustness. In ROS2, networking is built on top the pubsub model akin to ZeroMQ, as used in our CM.

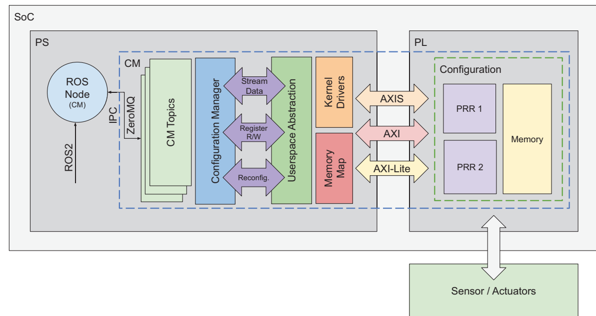

Fig. 3. Architecture of the ROS2 wrapper using our CM and ZeroMQ.

A ROS2 message is composed of predefined data structures and is used to standardise the production/consumption of data. For example, a publishing node in an autonomous vehicle, such as a controller for LiDAR, could publish an image payload to a `camera_data` topic. This would then alert n subscribers listening to the topic that a payload was available for consumption. In a CPS example, subscriber nodes could be additional image processing tasks, data observation events, network events to push the images back to a data centre, etc.

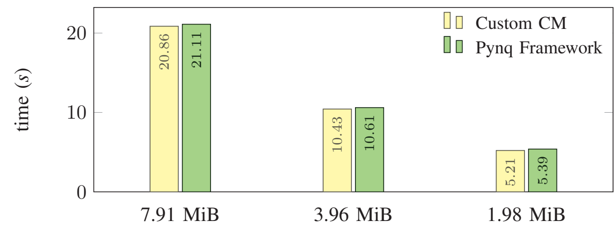

Fig. 4. Round trip time to transfer 1000 images between PS and PL.

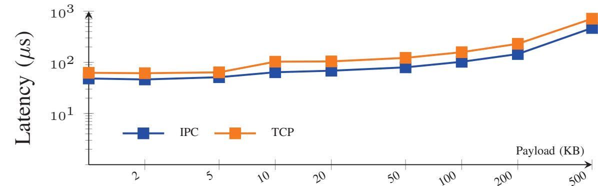

Fig. 5. ZeroMQ latency measured with varying payloads.

B. Experiment

We use a ZeroMQ-ROS2 wrapper which maps our CM's ZeroMQ endpoints to a ROS2 node, seen in Fig 3. This turns the ROS2 node into a wrapper for the CM's ZeroMQ publishing/subscribing topics. We use the userspace abstractions provided by our CM to expose the PL to the ROS2 node.

We perform an experiment to quantify the overhead of moving data between an ROS2 node and a hardware accelerator. This demonstrates streaming image/video data to/from hardware to a simple software cognitive engine in a CPS. We focus on two main overheads; at the ZeroMQ interface and moving data between the PS and the PL. We benchmark a PS-PL-PS DMA transaction to quantify the transfer performance between the CM and PL as well as effects of ZeroMQ on latency and throughput for general communication. Data is transferred round trip from the PS to a PL DMA controller clocked at 100 MHz, running in Ubuntu 18.04, on an Ultra96v2 ZynqMP development board. This is isolated from the ZeroMQ interface and the time taken to perform the DMA transactions over 1000 intervals was measured using 3 different size images (7.91, 3.96 and 1.98 MiB). Considering the 3.96 MiB image, we measured a max. approximate throughput of 380.04 MiB/s (per direction), which was a total of 10.43s for 1000 transfers and 10.43ms per image; 95% of the theoretical max. throughput of this PS-PL interface [32]. We compared this to the PYNQ framework, where we saw the same image transferred in an average of 10.61ms, highlighting how our abstraction performed better by an acceptable margin against the overhead seen in a comparable abstraction framework. CM PR time can be drawn from benchmarks in [26], where the maximum configuration throughput is 398.6 MB/s with an average trigger latency of 7μs.

We then tested the ZeroMQ layer under the same experiment using IPC, to simulate data moving between the CM memory buffers and the ROS node. We measured the latency and throughput across 1000 trips with the 3.96 MiB image. We saw an average of 5.36 ms latency and approximate throughput of 1939.5 MiB/s, indicating it does not provide a bottleneck. We also compared differing payloads using ZeroMQ IPC

and TCP mode to represent local API calls, as seen in Fig. 5. We evaluate these results against a comprehensive benchmark of ROS2 overheads, performed on a desktop class machine [33]. They demonstrate the transmission of a 2 MB ROS2 payload, using the OpenSplice DDS, which shows approximately 2 ms of latency; within 1.2 ms of our results for an equivalent payload. We conclude that compared to the latency experienced under ROS2 networking, the CM overheads for both data transmission and reconfiguration are insignificant, considering the added abstraction benefits. Given the factors that can influence latency such as Linux scheduler, networking, payload size, etc., it is likely that we could see further reductions, such as with the use of component latency reduction techniques presented in [34].

V. Conclusion

We have demonstrated an abstracted configuration manager for managing hardware acceleration within an autonomous adaptive system implemented on an FPGA SoC. We show that high level software automation frameworks can offload complex hardware processing of sensor data and actuator logic, without requiring custom low level integrations with the kernel. We showed how our CM can be used with frameworks such as ROS2, where the hardware can be abstracted into a networked node. Additionally we expanded existing build flows to take user configurations, extend them with implementation details from the build process and pass them to the CM, for seamless integration under Linux. As future work, we intend to release this work as part of an open source framework for adaptive systems and evaluate networking the CM across multiple platforms.

Acknowledgement

This work was supported by the UK Engineering and Physical Sciences Research Council, grant EP/N509796/1.

References

[1] M. Bouhali, F. Shamani, Z. E. Dahmane, A. Belaidi, and J. Nurmi, "FPGA applications in unmanned aerial vehicles-a review," in *Int. Sym. on Applied Reconfigurable Computing*, 2017, pp. 217–228.

[2] S. Haykin, "Cognitive radio: brain-empowered wireless communications," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 2, pp. 201–220, 2005.

[3] C. Hao *et al.*, "A hybrid GPU+ FPGA system design for autonomous driving cars," in *IEEE Int. Workshop on Signal Processing Systems (SiPS)*, 2019, pp. 121–126.

[4] J. Kok, L. F. Gonzalez, and N. Kelson, "FPGA implementation of an evolutionary algorithm for autonomous unmanned aerial vehicle onboard path planning," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 2, pp. 272–281, 2012.

[5] Y. Lyu, L. Bai, and X. Huang, "Real-time road segmentation using LiDAR data processing on an FPGA," in *IEEE Int. Sym. on Circuits and Systems (ISCAS)*, 2018, pp. 1–5.

[6] P. Jamshidi, J. Cámara, B. Schmerl, C. Käestner, and D. Garlan, "Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots," in *IEEE/ACM Int. Sym. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2019, pp. 39–50.

[7] C. Brugger, L. Dal'Aqua, J. A. Varela, C. De 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 *IEEE Sym. on Computer Applications & Industrial Electronics (ISCAIE)*, 2015, pp. 201–206.

[8] S. Shreejith, B. Banarjee, K. Vipin, and S. A. Fahmy, "Dynamic cognitive radios on the Xilinx Zynq hybrid FPGA," in *Int. Conf. on Cognitive Radio Oriented Wireless Networks*, 2015, pp. 427–437.

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

[10] 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 *IEEE Int. Conf. on Embedded Software and Systems*, 2019.

[11] B. Fuller, J. Kok, N. A. Kelson, and L. F. Gonzalez, "Hardware design and implementation of a MAVLink interface for an FPGA-based autonomous UAV flight control system," in *Australasian Conf. on Robotics and Automation*, 2014.

[12] C. Savaglio and G. Fortino, "Autonomic and cognitive architectures for the Internet of Things," in *Int. Conf. on Internet and Distributed Computing Systems*, 2015, pp. 39–47.

[13] H. Hoffmann, "SEEC: A framework for self-aware management of goals and constraints in computing systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2013.

[14] J. Strassner, N. Agoulmine, and E. Lehtihet, "FOCALE: A novel autonomic networking architecture," *ITSSA Journal*, vol. 3, no. 1, pp. 64–79, 2006.

[15] S. A. Fahmy, "Design abstraction for autonomous adaptive hardware systems on FPGAs," in *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, 2018, pp. 142–147.

[16] S. Clayman and A. Galis, "INOX: a managed service platform for interconnected smart objects," in *Workshop on Internet of Things and Service Platforms*, 2011.

[17] M. Quigley *et al.*, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, vol. 3, no. 3.2, 2009, p. 5.

[18] R. De Lemos *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*, 2013, pp. 1–32.

[19] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "FOS: A modular FPGA operating system for dynamic workloads," *ACM Tran. Reconfigurable Technol. Syst.*, vol. 13, no. 4, Sep. 2020.

[20] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–39, 2018.

[21] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *IEEE Int. Sym. on Field-Programmable Custom Computing Machines*, 2014, pp. 109–116.

[22] "Python productivity for Zynq." [Online]. Available: http://pynq.io/

[23] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators," in *IEEE Int. Sym. on Field-Programmable Custom Computing Machines*, 2011, pp. 170–177.

[24] 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, 2013.

[25] K. Vipin and S. A. Fahmy, "Mapping adaptive hardware systems with partial reconfiguration using CoPR for Zynq," in *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, 2015.

[26] A. R. Bucknall, S. Shreejith, and S. A. Fahmy, "Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq Ultrascale+," in *Int. Conf. on Field-Programmable Technology*, 2020.

[27] ZeroMQ. [Online]. Available: https://zeromq.org/

[28] OpenDDS. [Online]. Available: https://opendds.org/

[29] "Protocol buffers." [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview

[30] B. Perez and J. Choi, "Xilinx AXI DMA linux driver," https://github.com/bperez77/xilinx_axidma, 2015.

[31] V. DiLuoffo, W. R. Michalson, and B. Sunar, "Robot operating system 2: The need for a holistic security approach to robotic architectures," *Int. Journal of Advanced Robotic Systems*, vol. 15, no. 3, 2018.

[32] *PG021: AXI DMA v7.1*, Xilinx Inc., Jun. 2019, v7.1.

[33] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *Int. Conf. on Embedded Software*, 2016.

[34] Y. Sugata, T. Ohkawa, K. Ootsu, and T. Yokota, "Acceleration of publish/subscribe messaging in ROS-compliant FPGA component," in *Int. Sym. on Highly Efficient Accelerators and Reconfigurable Technologies*, 2017.