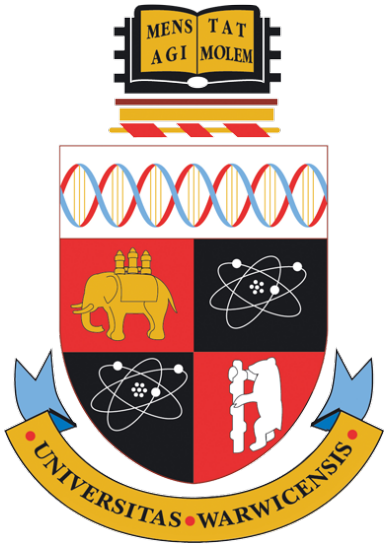
WARWICK
THE UNIVERSITY OF WARWICK



Build Framework and Runtime Abstraction for Partial Reconfiguration on FPGA SoCs

by

Alex R. Bucknall

Thesis

Submitted to the University of Warwick

in fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

School of Engineering

January 2022

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my supervisor Suhaib Fahmy for his guidance and support throughout my research. His advice has been invaluable and helped me to shape the way I approach my work.

I would like to thank my colleagues, Ryan and Lenos, who undertook their PhDs during the same period as myself. Their advice and company has been much appreciated and assisted me with many of the technical and challenging issues I faced during my research.

Lastly I would also like to thank my friends, Catriona, Joe, Phil and Lizzie who have supported me through this difficult period, both with challenges I faced with my work as well as externally with the stresses of the pandemic.

Declarations

Parts of this thesis have been previously published in the following:

[16] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Network enabled partial reconfiguration for distributed FPGA edge acceleration. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 259–262, 2019. doi: 10.1109/ICFPT47387.2019.00042

[14] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq UltraScale+. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 215–220, 2020. doi: 10.1109/ICFPT51103.2020.00037

[17] Alex R. Bucknall and Suhaib A. Fahmy. Runtime abstraction for autonomous adaptive systems on reconfigurable hardware. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 1616–1621, 2021. doi: 10.23919/DATE51398.2021.9474199

Parts of this thesis are pending publication in the following:

[18] Alex R. Bucknall and Suhaib A. Fahmy. ZyCAP2: End-to-end build tool and runtime manager for partial reconfiguration of FPGA SoCs at the edge. In *submitted to: TRETS*, 2021

Abstract

Growth in edge computing has increased the requirement for edge systems to process larger volumes of real-time data, such as with image processing and machine learning; which are increasingly demanding of computing resources. Offloading tasks to the cloud provides some relief but is network dependant, high latency and expensive. Alternative architectures such as GPUs provide higher performance acceleration for this type of data processing but trade processing performance for an increase in power consumption. Another option is the Field Programmable Gate Array; a flexible matrix of logic that can be configured by a designer to provide a highly optimised computation path for incoming data. There are drawbacks; the FPGA design process is complex, the domain is dissimilar to software and the tools require bespoke expertise. A designer must manage the hardware to software paradigm introduced when tightly-coupled with general purpose processor. Advanced features, such as the ability to partially reconfigure (PR) specific regions of the FPGA, further increase this complexity. This thesis presents theory and demonstration of custom frameworks and tools for increasing abstraction and simplifying control over PR applications. We present mechanisms for networked PR; a mechanism for bypassing the traditional software networking stack to trigger PR with reduced latency and increased determinism. We developed a build framework for automating the end-to-end PR design process for Linux based systems as well as an abstracted runtime for managing the resulting applications. Finally, we take expand on this work and present a high level abstraction for PR on cyber physical systems, with a demonstration using the Robot Operating System. This work is released as open source contributions, designed to enable future PR research.

Sponsorships and Grants

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

Acronyms

AAS Autonomous Adaptive System.

ADAS Adaptive Driver Assistance Systems.

AEX Advanced Encryption Standard.

ALM Adaptive Logic Module.

API Application Programming Interface.

APU Application Processing Unit.

ATF ARM Trusted Firmware.

BRAM Block Random Access Memory.

BSP Board Support Package.

CGRA Course Grain Reconfigurable Array.

CLB Configurable Logic Block.

CM Configuration Manager.

CMA Contiguous Memory Allocator.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

DCP Design Checkpoint.

DFX Dynamic Function Exchange.

DMA Direct Memory Access.

DMAC Direct Memory Access Controller.

DPR Dynamic Partial Reconfiguration.

DPU Deep Learning Processing Unit.

DRAM Dynamic Random Access Memory.

DT Device Tree.

DTO Device Tree Overlay.

E2E End to End.

EMAC Ethernet Media Access Controller.

FIFO First-In-First-Out.

FPGA Field Programmable Gate Array.

FPU Floating Point Unit.

GP General Purpose.

GPIO General Purpose Input Output.

GRU Gated Recurrent Network.

GUI Graphical User Interface.

HDF Hardware Description File.

HDL Hardware Descripton Language.

HLS High Level Synthesis.

HP High Performance.

ICAP Internal Configuration Access Port.

IOB I/O Block.

IPC Inter-Process Communication.

ISA Instruction Set Architecture.

KNN K-nearest neighbour.

LKM Loadable Kernel Module.

LUT Look Up Table.

MCAP Media Configuration Access Port.

MMIO Memory Mapped I/O.

NIC Network Interface Card.

OS Operating System.

PCAP Processor Configuration Access Port.

PCI Peripheral Component Interconnect.

PCIe Peripheral Component Interconnect Express.

PE Processing Element.

PHY Physical Layer Device.

PL Programmable Logic.

PMU Platform Management Unit.

PR Partial Reconfiguration.

PRM Partially Reconfigurable Module.

PRR Partially Reconfigurable Region.

PS Processing System.

ROS Robot Operating System.

RP Reconfigurable Partitions.

RPU Real-time Processing Unit.

RTL Register Transfer Level.

RTOS Real Time Operating System.

SDK Software Development Kit.

SERDES Serializer/Deserializer.

SEU Single Event Upset.

SoC System on Chip.

SRL Shift Register Logic.

SVM support vector machine.

TCAM ternary content-addressable memory.

TDP Thermal Design Power.

TPM Trusted Platform Module.

UIO Userspace I/O.

Chapter 1

Introduction

Heterogeneous computing is construction of differing compute architectures within close proximity, where exchanging of data is performed bi-directionally using the most efficient computing architecture for a specific operation. Often this will be mapped to a host and follower hierarchy where a system such as a general purpose processor (GPP) may offload tasks to a co-processing system. This approach often lends itself to higher performance, lower energy consumption as well as reduced latency for performing computation, when an alternative computation platform is optimised for that workload.

Throughout history, co-processors have changed and advanced as the demands of computing have increased in complexity. Early co-processing cores were utilised by GPPs for simple tasks such as floating point operations performed on a discrete floating point unit (FPU), which was optimised specifically for floating point arithmetic. These operations were demanding enough to require custom compute and thus freed the GPP from performing these operations on non-optimised architecture, reducing latency and core utilisation. Due to the rise of screens and displays as human-computer interfaces, graphical processing units (GPUs) became more commonplace for compute offload for GPPs, where powerful GPUs could be accessed over high performance external bus interfaces such as the peripheral component interface (PCI). Recently offloading tasks have become increasingly discrete with cryptographic, digital signal processing and networking tasks passed to co-processors that are optimised for these kinds of operations.

The evolving nature of computing tasks has lead to an interest in flexible hardware accelerators for task offloading where devices, such as Field Programmable Gate Arrays (FPGA), can customise their internal architecture to best serve the computation, are gaining popularity. This type of co-processing can be described as custom hardware acceleration, where computation paths made from logic on the FPGA, allow it to be optimised for the specific task at hand. Compared against traditional GPP software operations, where the processor

must follow the fetch, decode and execute cycle to run code, an FPGA can perform compute operations on incoming data in a pipelined manner through a custom datapath for increased throughput. FPGAs are reconfigurable at runtime, allowing them to dynamically change their behaviour on demand or in response to a specific type of task. This unique feature has seen popular use cases for in-network processing such as with switches and radio base stations, ideal for inline packet processing operations. Reconfiguration at runtime makes FPGAs ideal for constrained edge of network type devices, allowing resource limited systems to adjust their compute while deployed, enabling hardware to be updated remotely. This flexibility is a major differentiator compared against other types of hardware acceleration, such as ASICs and TPUs, better enabling remote edge systems to adapt as their environment demands. In Chapter 2 we provide a comprehensive background and literature review discussing the specifics of architectures and implementations of various types of hardware accelerators.

1.1 Motivations

Many modern computing systems combine varying types of compute architectures with the intention to offload tasks that may benefit from a specific architecture, such as GPUs for video processing and network interfaces for network processing. GPU-based architectures have significant support across operating systems with popular software development kits (SDKs) as well as drivers with low level acceleration. This specific combination has also seen a rise in edge-of-network acceleration, such as in the internet of things, as the parallelism of GPUs is beneficial for vision processing as well as machine learning tasks. These systems need to be highly optimised for efficient computing, where events are abstractly passed between the most suited hardware for the target task. Typically IoT applications are producers of sensor data, with historical use-cases simply forwarding the generated data up to the cloud for aggregation and processing at a later date. While this is sufficient for applications that may tolerate significant latency, when sending data to and from the cloud, real-time applications, such as RF baseband processing or image processing systems operating in millisecond or even nanosecond response times, cannot afford such delays. As edge devices become more powerful and provide heterogeneous computing, the ability to dynamically offload tasks to localised compute becomes increasingly interesting for developers.

This has lead to the development of heterogeneous Application Processor Unit (APU) that are tightly coupled with FPGAs, where the APU can support an operating system such as Linux and the FPGA can be configured at runtime for dedicated acceleration operations. Xilinx's Zynq and Zynq Ultrascale+

platforms are examples of such powerful embedded ARM APUs, extended with the programmable logic and resources of an FPGA. FPGA acceleration provides a number benefits to edge computing including increased performance of pipelined tasks such as image processing, higher energy efficiency than the equivalent compute on a GPP. While GPUs run software instructions and can thus be more flexible, FPGAs are best placed to serve applications that do not map well to GPUs and where the energy cost of powerful GPUs is too great for the given application.

Additionally, FPGAs do not require a GPP host to move data into and out of acceleration, they are capable of hosting their own processors and even network controllers for data transfer and bitstream provisioning. These FPGA System on Chip (SoC) devices locate the ARM APU and FPGA on the same silicon die with multiple high performance interfaces for bidirectional communication between them. There are numerous layers of complexity for managing control of the FPGA from the APU, where the APU typically directs control of the FPGA as a co-processor, while running a fully featured operating system.

While FPGAs excel at optimised compute, when designed specifically for an acceleration operation, they are less desirable for general purpose compute as supporting an operating system or managing complex networking stacks. While significant research has been undertaken to optimise and explore the co-processing power of FPGA SoCs, the abstractions for software interfaces and data exchange between the APU and FPGA are limited. Many implementations of the APU to FPGA interfaces require low level drivers, complicated memory management between virtual and physical memory as well as limited abstraction for provisioning the FPGA on demand.

A greater investigation of the APU to FPGA interface and abstraction is needed to increase the accessibility and adoption of FPGAs as co-processing offload for higher level software libraries and SDKs that are popular in modern edge computing applications. This thesis aims to understand, explain and offer tools to reduce these barriers for FPGA accelerated applications.

1.2 Objectives

The main objectives of this research are as follows:

1. To develop an architecture for rapidly re-provisioning FPGA SoCs over the network, bypassing the software networking stack typically used by their coupled APU to decode and initiate reconfiguration.

2. To design build tools capable of absorbing/reducing the complexity associated with designing and building FPGA accelerators for non-experts,

including:

- The ability to abstract the build pipeline for hardware description language files and PR configurations into FPGA bitstreams
- The capacity to gather the exported bitstreams and metadata to construct a Linux operating system image with full support and abstraction for underlying PR hardware

3. To develop a high performance runtime manager that is capable of abstracting the complexity of software hooks for low level memory management of FPGA logic, provisioning of full and partial bitstreams as well as the ability to configure the Linux kernel to accommodate the corresponding FPGA logic.

4. To design an abstraction for managing the complexities of configurations and modes of operation of cyber physical systems while being managed by the APU. This should allow of high levels of abstraction and usage within popular IoT frameworks and tooling.

1.3 Contributions

The main contributions of this work are a collection of build tools, runtime managers, and case studies that are intended to assist with the complexities of designing and developing partially reconfigurable acceleration applications on heterogeneous computing platforms. Throughout the thesis, these tools are described as the ZyCAP2 framework; where we draw on the work of authors in [11] to extend abstraction, features and utility of the tools, specifically supporting the Linux operating system. Figure 1.1 shows how the contributions in this thesis can be categorised. A number of these contributions are grouped together as the ZyCAP2 build framework and runtime manager, where other contributions may be combined at a later date as plugins or extensions to the base build framework.

These can be summarised as follows:

1. An arbitration mechanism for parsing networked reconfiguration frame headers directly in the FPGA, before forwarding reconfiguration commands on to the processing system.

2. An extensible end-to-end FPGA PR and Linux build tool, for partially reconfigurable designs that automates the generation of infrastructure to support user logic and manages device tree overlays, drivers, and memory mapped IO, with particular focus on edge computing platforms. This includes:

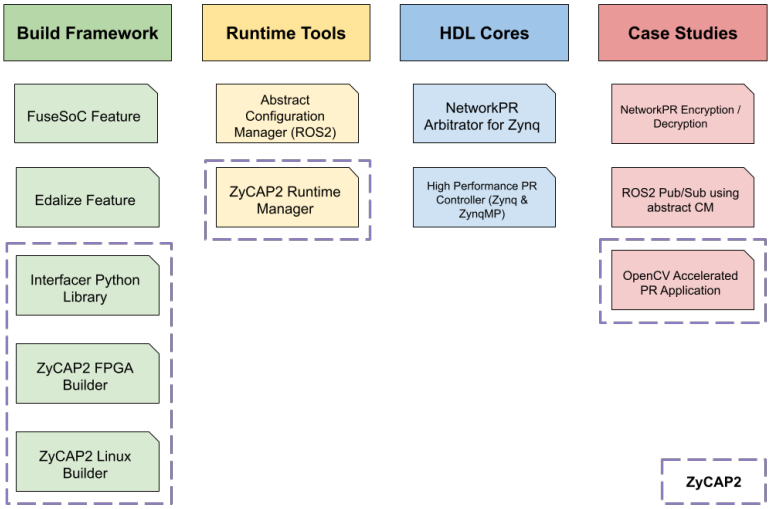


Figure 1.1: Thesis Contributions

(a) A standalone Python library for interface extraction and wrapping PR module interfaces to enable easily building soft SoC infrastructure.

(b) Extension of the FuseSoC/Edalize libraries to support Partial Reconfiguration; an open source tool for managing cross-platform building of hardware description language IP cores and libraries.

3. A PR runtime manager and configuration API for PS-PL management that enables simple software abstraction of memory mapped IO, DMA streaming as well as loading/unloading partial and complete bitstreams as part of our described mode and configuration abstractions. This includes:

   (a) Improved high performance asynchronous PR controller for loading the ZynqMP ICAP interface at near theoretical throughput (approximately 757 MiB/s).

   (b) A case study using Vitis HLS generated OpenCV edge accelerators in a PR application that demonstrates our software abstraction and benchmarks the performance impact and logical resource consumption.

4. An abstracted Linux configuration manager, built on an adaptive systems model to automate reconfigurable hardware management and allow control from a PubSub (ZeroMQ) architecture.

5. A case study for the Robot Operating System 2 designed to utilise configuration abstraction to allow for simplified software control of hardware on cyber physical systems.

1.4 Thesis Roadmap

The thesis is organised as follows:

Chapter 2 provides a comprehensive research background and overview of the objectives of the work in this thesis.

Chapter 3 presents a detailed literature review and comparison of design tools, partial reconfiguration workflows, abstractions for adaptive systems as well as academic applications of partial reconfiguration.

Chapter 4 presents our work on improving network responsiveness for the loading and provisioning partial reconfigurable systems through bypassing the networking stack in software with arbitration on the FPGA.

Chapter 5 discusses our FPGA and Linux build workflows for generating partially reconfigurable bitstreams with accompanying design abstractions and generating accompanying drivers, APIs and configurations under Linux.

Chapter 6 examines our PR runtime manager and API as well as discussing and benchmarking the tool with an image processing case study, highlighting features enabled by the build and runtime components.

Chapter 7 examines an abstraction for controlling autonomous cyber physical systems from a heterogeneous reconfigurable systems approach, with demonstration of the configuration manager built on the Robot Operating System 2.

Chapter 8 summarises the work presented in this thesis and provides suggestions for potential future research opportunities.

1.5 Publications

The work presented in this thesis has featured in the following publications:

1. Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Network enabled partial reconfiguration for distributed FPGA edge acceleration. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 259–262, 2019. doi: 10.1109/ICFPT47387.2019.00042 [16] (Conference)

2. Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq UltraScale+. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 215–220, 2020. doi: 10.1109/ICFPT51103.2020.00037 [14] (Conference)

3. Alex R. Bucknall and Suhaib A. Fahmy. Runtime abstraction for autonomous adaptive systems on reconfigurable hardware. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 1616–1621, 2021. doi: 10.23919/DATE51398.2021.9474199 [17] (Conference)

The work pending publication:

1. Alex R. Bucknall and Suhaib A. Fahmy. ZyCAP2: End-to-end build tool and runtime manager for partial reconfiguration of FPGA SoCs at the edge. In *submitted to: TRETS*, 2021 [18] (Journal)

1.6 Open Source

The following work, presented in this thesis, are provided as open source:

1. **ZyCAP2 Build Framework** — End-to-End Partial Reconfiguration Build and Runtime Framework for Zynq and Zynq Ultrascale+ Devices. `https://github.com/warclab/zycap2`.
2. **ZyCAP2 Runtime Manager** — C++ Runtime Manager API for High Throughput DMA PR Provisioning on the Zynq and ZynqMP. `https://github.com/warclab/zycap2`.
3. **Interfacer Library** — Python Library for Verilog Interface Extraction. `https://github.com/warclab/interfacer`.

The work pending contribution (to be accepted):

1. **Edalize/FuseSoC PR Extension** — An extension to Edalize and FuseSoC libraries that provides support for PR module and static region generation from within the Vivado tools. To be contributed to `https://github.com/olofk/edalize` & `https://github.com/olofk/fusesoc`.

Chapter 2

Background

Adaptive systems are systems that can provide capacity for compute in uncertain or changing operating conditions. Such systems can be described with layers of abstraction, defining their behaviour and functionality. [19] describes a number of models for autonomous systems, outlining an advanced model for how such systems could be designed.

At runtime, the system may be required to adjust its operating parameters, either via alternative software routines or utilising co-processing hardware to handle demanding data rates. General purpose processing enables autonomous adaptive systems (AAS) to easily and rapidly respond to stimuli however such approaches are limited to the processing capabilities, which are typically low power and energy efficient for edge systems. Such systems that must interface with the physical environment, via sensors and peripherals such as cameras and networking interfaces, may be required to process large streams of data as well as meet strict real time requirements. Often the limited compute capability of such systems is insufficient for high performance embedded applications, due to the power and energy restrictions. This type of scenario lends itself to application acceleration via co-processing hardware, such as GPUs and/or FPGAs.

This Chapter explores processing architectures available to edge computing platforms, comparing and contrasting their advantages and disadvantages. Then it moves into exploring specific characteristics of FPGAs, highlighting Partial Reconfiguration as a key feature for acceleration. The next Chapter examines the concept of partial reconfiguration in adaptive systems and their applications in the real world. Finally it examines autonomous adaptive systems and the abstractions required to enable and extend such systems to span the hardware-software paradigm.

2.1 Computing Platforms

In the context of this thesis, a computing platform can be defined as any hardware architecture capable of performing compute. This includes a range of varying platforms from traditional general purpose processing on a CPU through to task specific dedicated compute on application specific integrated circuits. As the demands for compute have changed alongside technological advancements in fields such as image processing and machine learning, the requirements for compute have adapted to keep pace.

2.2 General Purpose Processors

General Purpose Processors also described as generalised compute are backbone of many traditional computing systems. Numerous processing architectures exist, where compromises are made between best service performance, energy efficiency and thermal properties. Operations and compute tasks are defined by a software program, often a high level abstraction on the GPP's instruction set architecture (ISA) or fundamental compute operations. Software programs are stored in memory and are executed sequentially, defined by their composition in the program. There are a number of different types of GPP; in this thesis Central Processing Units (CPU) are referenced as desktop or server class compute and Application Processing Units (APU) as referred to the hybrid architectures often found in mobile computing or System on Chip platforms.

2.2.1 x86

Traditionally personal computers and datacenter compute such as CPUs have been based upon the x86 architecture. This architecture is largely power inefficient and while it succeeds in terms of performance delivery, is less useful to edge platforms where energy efficiency is key. The two major vendors and manufacturers of x86 based architectures are Intel and AMD, who both design and fabricate processors of this architecture. Intel briefly explored low power internet of things style x86 processors such as their Quark [20] portfolio but these failed to gain commercial adoption and have since been discontinued.

2.2.2 ARM

ARM introduced an alternative offering with their own ISA, which optimised for power efficiency through the use of a heterogeneous computing architecture which offered to share workloads across high and low performance GPP cores to best optimise for a low thermal design power (TDP), as required by mobile applications. While x86 processors tend to use fewer high performance cores,

ARM processors optimise workloads across many small processing cores, often task specific cores. This is described as ARM's big.LITTLE architecture, where higher power and less energy efficient cores are coupled with smaller more power efficient cores to spread the compute load across a heterogeneous multi-processing system. [21] offers a comparison of performance and energy of ARM's big.LITTLE against Intel's Sandy Bridge architecture, where the Intel processors were shown to have an average 12.6× to 152.4× higher power demand than the ARM processors. For this reason, ARM processors have become the major architecture in mobile computing such as with smart phones, tablets and internet of things devices. ARM-based processing cores are interesting for acceleration due to their advantages for low power edge applications as well as their custom heterogeneous compute cores. Unlike Intel or AMD, ARM licenses their processor architecture to commercial customers such as Xilinx who can integrate their IP into heterogeneous systems.

Recent ARM architectures such as ARMv8-A and ARMv9-A support trusted firmware which is used at boot time to ensure that the bootloader/kernel are signed and allowed to execute at specified levels of hardware privilege. This is used to limit or restrict application access from hardware such as attached FPGAs or secure keys.

2.3 Graphical Processing Units

Graphical Processing Units are another compute platform traditionally used for image and video processing. Like the processor, the GPU interprets compute tasks through a software program and are typically coupled to a GPP for acceleration. Where the GPU differs from the processor is that it is composed of a significantly larger numbers of simple compute cores that operate in parallel. Each compute core is computationally similar to that of a GPP but optimised for floating point arithmetic operations. Unlike a GPP which may be required to handle and manage system IO, the GPU is considerably more dense, able to compact a high number of compute cores into its silicon die. GPU cores may also be grouped and used to perform specific operations, reducing the need to fetch sequential operations from memory.

Traditional GPUs are typically attached to a GPP via high throughput data buses such as PCIe, where the throughput can be multiple gigabits per seconds. For embedded devices, GPUs may be tightly coupled to the processor, existing on the same System on Chip (SoC) using hardened bus interfaces.

Given that image manipulation typically occurs with large matrices of pixel data, GPUs are optimised for operations that can be distributed across parallel workloads, allowing for the time taken to perform complex processing to be drastically reduced. More recently GPUs have found applications in

accelerating machine learning such as in [22], [23] and [24]. While GPUs have a significant performance advantage over general purpose processors for massive parallel operations, they still are susceptible to limitations regarding memory access, also described by Von Neumann's Bottleneck [25]. Operations are executed sequentially such that memory must be accessed to retrieve the next operation, resulting in limitations on throughput and increasing energy consumed. Additionally, applications that are not easily expressed through parallel matrix operations are not well suited for GPU acceleration.

2.4 Application Specific Integrated Circuits

Application Specific Integrated Circuits (ASIC) are task specific computation devices, exclusively fabricated to carry out a specific set of tasks. The architecture of an ASIC is fixed; unlike a processor or GPU which are able to interpret a instructions within software, an ASIC operates like a pipeline of fixed logical operations, able to perform a specific task extremely efficiently but are specific to their designed application. ASICs are capable of performing at significantly higher clock rates than GPPs or FPGAs as at design time, they are not restricted to a specific architecture or clocking interconnect. The drawback of designing applications for ASICs is the initial cost required to develop custom silicon devices. Unlike an FPGA, which has custom architecture that can be defined in-field, once an ASIC is manufactured, it will only ever perform the task it was originally designed for. Any change in functionality or application will require a new ASIC to be manufactured; typically reserved for large production runs due to the cost of fabricating silicon devices.

A common example of ASICs for compute offload are cryptographic coprocessors, such as Trusted Platform Modules (TPM), which enable an application designer to generate, store and limit the use/access to cryptographic keys. For security applications, the advantage of an ASIC in this specific application is that master keys are physically burnt to the device at manufacture and are never exposed to any other component or software. Both GPPs and FPGAs can be susceptible to key extraction attacks where private keys used for encryption/decryption can be extracted thought bypassing software restrictions or dumping the data stored within firmware/bitstreams.

Another popular use case of ASICs is machine learning, where platforms such as Google's Tensor Processing Unit (TPU) offer commerical ASICs for acceleration TPUs are designed to accelerate the computation of tensor models, used in the neural networks required for machine learning. For tensor model acceleration a specialised matrix processor is able to handle massive multiplication and addition operations rapidly, while consuming significantly less power than an equivalent GPP or GPU. Google's Coral TPU [26] platform offers

significant performance improvements over the traditional GPU acceleration; [27] shows how Google Cloud TPU can accelerate training of CNNs at a rate of 2-3× faster than an Nvidia V100 GPU. [28] offers a comparison for CNN interference on FPGAs against TPUs, comparing Xilinx's Deep Processing Unit [29] (DPU) in differing configurations against the Coral TPU. The DPU is shown to outperform the TPU in inference time, in specific configurations of the DPU. The DPU implementation has some flexibility advantages where it can exchange logical resource consumption on the FPGA for performance, although the DPU architecture itself is proprietary.

2.5 Field Programmable Gate Arrays

Field Programmable Gate Arrays are configurable silicon devices that consist of a matrix of logical elements. FPGAs are unique compared to software centric processing platforms (processor, GPU, etc.) as user logic is described using to logical elements and hardened macros such as digital signal processing and memory elements embedded within the device. Figure 2.1 provide a overview of the generic architecture of a Xilinx FPGA, showing generally how elements are arranged. There are many other vendors offering FPGAs for market niches, such a Lattice for low powered applications, but the two major vendors are Xilinx and Intel.

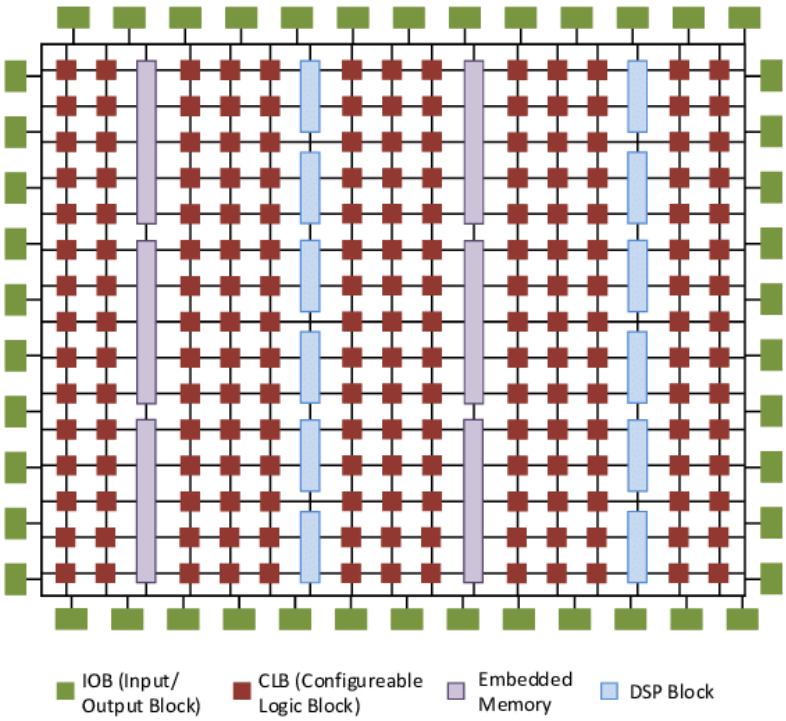


Figure 2.1: Generic Xilinx FPGA Architecture [1]

FPGAs can be used as adaptive compute platforms where reconfiguration of

the internal logic of the device may be performed either before or during runtime and adapted to logic designed by the user. While ASICs may offer lower latency and a highly optimised data path, FPGAs provide respectable performance for operations that lend themselves to pipelining and customer architectures while being able to be reconfigured in the field. FPGAs enable efficient acceleration of operations such as extraction of key information or features, controlling the data-flow and analysing data [30], for a range of applications including image recognition [31], in-network security [32], predictive maintenance [33], machine learning [34], amongst many others. Academic works and real world applications are later explored in Chapter 3.

2.5.1 Architecture

While fundamentally all FPGAs are composed of loop-up tables (LUTs) and a routing matrix, vendors have their own custom architectures that devices are composed from. The FPGA fabric is often described as the programmable logic (PL). Often vendors will provide a wide variety of processing elements (PEs) to best serve the logic that is intended to be synthesised for them. For example, Xilinx's latest current generation Ultrascale device architecture is composed of Configurable Logic Blocks (CLBs) that contain:

- Real 6-input LUTs
- Dual 5-input LUTs
- Distributed Memory and Shift Register logic (SRL)
- Dedicated high-speed carry logic for arithmetic operations
- Wide multiplexers
- Dedicated storage elements that can be configure into flip-flops or latches

Figure 2.2 shows the two types of LUTs available to Ultrascale devices and Figure 2.3 highlights one of six elements, contains eight 6-input LUTs and sixteen storage elements, found in the single slice of a CLB. There are two types of slices; SLICEL (Logic), which contains just LUTs and SLICEM (Memory), which also contains memory elements such as shift registers and distributed RAMs.

These are then further organised into columns on the devices, interleaved with hardened blocks such as SRL, DSP and IOB blocks. Figure 2.4 shows a shift register logical element that SLICEM blocks are composed of. DSP blocks are used to reduce the number of LUTs required to implement complex arithmetic functions when synthesising logic, in particular for digital signal processing operations. In addition to DSP blocks, Block RAM (BRAM) blocks

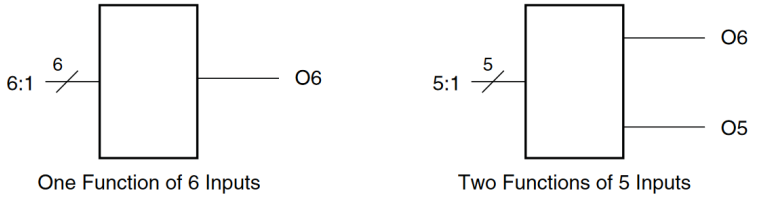


Figure 2.2: Ultrascale CLB LUT6 and dual LUT5 blocks [2]

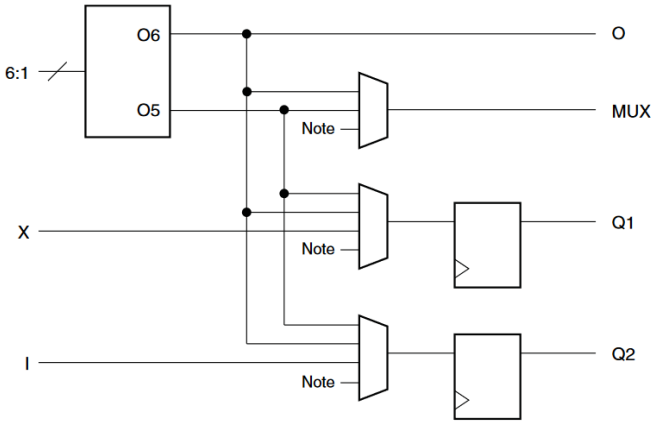


Figure 2.3: Ultrascale CLB LUT and storage elements (1 of 6 in a Slice) [2]

are typically implemented to provide local, in chip storage on the FPGA. This reduces the need for high latency reads and writes to external memory units. IO-blocks (IOBs) are placed around the edge of the architecture, enabling translation to the appropriate signal properties, such as voltage required to communicate with external devices. IOBs are organised into banks of IO, where IOBs may be specialised for high-speed transfers such as serializer-deserializer (SERDES) IO, which is typically used to communicate with protocols requiring high-speed clocks and differential data channels, like Ethernet and PCIe. These hardened resources are implemented to reduce the number of CLBs required to implement the equivalent logic or to provide signal conditioning that would not be possible, internal to the FPGA.

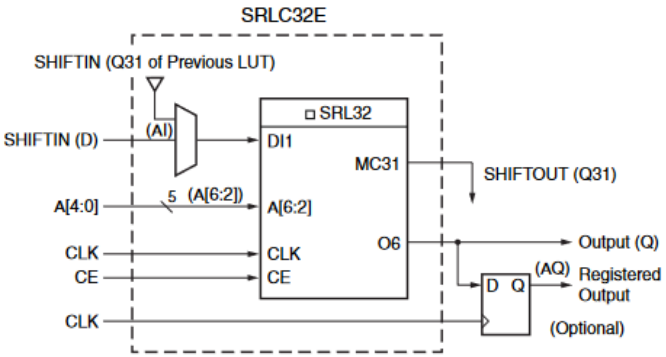


Figure 2.4: Ultrascale CLB Shift Register Logic [2]

IOB, CLB, BRAM and DSP columns are organised into clocking regions and distributed across the device as shown in Figure 2.5.

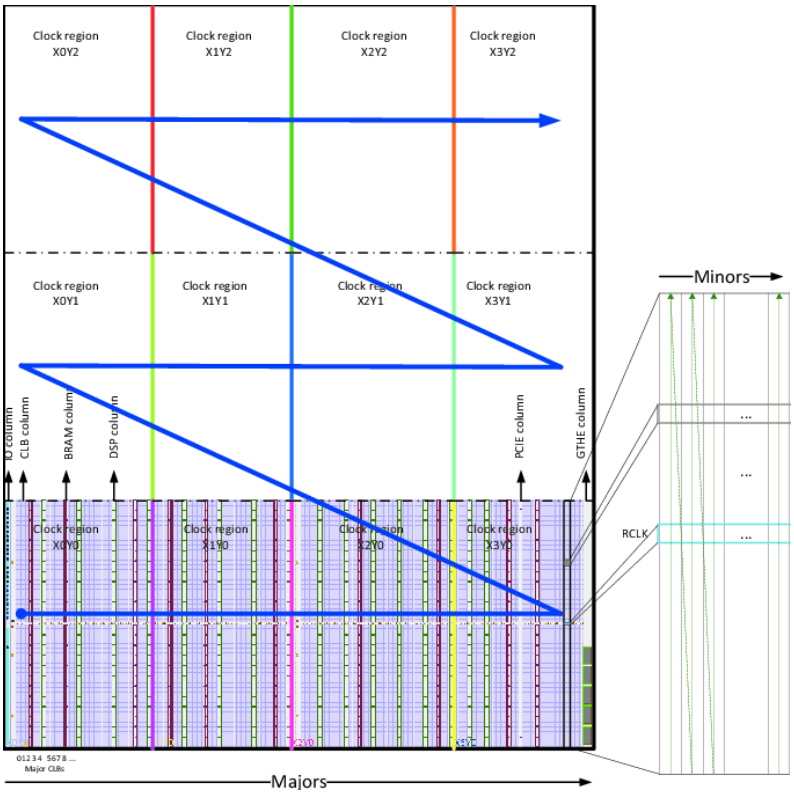


Figure 2.5: Ultrascale Kintex Floor plan [3]

CLBs are connected to each other, using a dedicated routing matrix. The routing matrix may be connected to multiple clocks, specified at design time. During the build process, implementation algorithms determine optimal placements for synthesised logic and optimise the routing matrix to minimize clock skew between LUTs. Other vendors organise their FPGAs in similar manners but use varying terminology to describe components. For example, Intel organises their LUTs into Adaptive Logic Modules (ALM) with 8-input LUTs.

A bitstream used to program and reprogram the FPGA, contains the information require to define the of logic within the LUTs and the how the LUTs are connected via the routing/switching matrix. This field re-programmability means that the cost to deploy custom FPGA based compute has a lower initial cost than developing an ASIC while still maintaining high levels of performance and/or throughput. Additionally, it is possible for FPGAs to function standalone since a soft-processor can be implemented, providing support for bare-metal (directly executing instructions with no operating system) and even full operating system support. Xilinx offers the MicroBlaze and Intel provides the NIOS II/V processors; support for soft ARM and RISCV-based processors also exists.

Table 2.1: Comparison of FPGA Hard & Soft Processors

Vendor	Processor	Type	Architecture
Xilinx	MicroBlaze	Soft	RISC32 or RISC64
Intel	NIOS II	Soft	RISC32
Intel	NIOS V	Soft	RISC-V
ARM	Cortex-M	Hard	ARM32
ARM	Cortex-A	Hard	ARM32 or ARM64

2.5.2 Reconfiguration

The major advantage of FPGAs over ASICs is their ability of reconfigure in the field. This allows for changes to be made to the acceleration hardware during and after deployment. Reconfiguration offers the ability to completely change the function of the FPGA and provide dynamic acceleration. There are a number of types of reconfiguration including static, parametric, partial and dynamic. Each of these methods has various advantages and disadvantages with regards to build and runtime implementations. For the purpose of relevance and maturity of partial reconfiguration, this Section will exclusively explore reconfiguration on Xilinx FPGA SoCs.

2.5.2.1 Static Reconfiguration

FPGAs are flashed through a process known as static reconfiguration. Static reconfiguration involves a fixed or static bitstream, containing information about how to connect LUTs, populate memory registers and how to route the interconnect matrix. Hardened circuitry internal to the FPGA reads a bitstream file and extracts the information to flash the FPGA accordingly. This configuration circuitry can occur over a number of protocols, including JTAG, serial flash, parallel flash as well as high speed protocols like PCIe. The procedure can be controlled by a host PC using the vendor's software as well as an accompanying microcontroller or SoC. For the ZynqMP devices, this can be managed by the PMU, which passes a bitstream to the FPGA from the processing system.

Figure 2.6: Xilinx Static Bitstream Layout (7 Series FPGAs) [4]

For complex applications, data that is transferred between the processing systems and PL may need to take multiple pathways for logical propagation.

Under static reconfiguration, all of these pathways must exist at any instance of time as the bitstream that has been flashed onto the FPGA is immutable at runtime. To satisfy complex designs, demanding more hardware resources, this may require a larger FPGA device. This leads to design complexity as well as an increase in cost and power consumption. Larger designs are more difficult to achieve higher operating frequencies as the clock path may need to be extended to match the datapath. Additionally if the design in the FPGA is static, this means that new functions or module logic will require a full redesign of the logic in the PL.

Additionally the FPGA itself can be completely reprogrammed with a new bitstream at runtime however this is typically slow, in the region of multiple seconds for small bitstreams (over JTAG or Flash mechanisms), proportional to the size of the bitstream being loaded.

2.5.2.2 Parametric Reconfiguration

One option for managing changing datapaths is a design methodology known as parametric reconfiguration. This is performed by using accelerators that utilise internal registers to specify their operating mode, such as an image processing block changing the resolution of images that it can process. Parametric reconfiguration can potentially reduce the resource consumption of a design and limit the effects of an increasing datapath on clock frequency by condensing the operating modes into a single function/module. While this means that a number of configurations can be supported, this increases design time complexity as the FPGA must now support communication from an internal/external logic to specify the desired parametric changes and modules must be designed to support all valid configurations internally.

2.5.2.3 Partial Reconfiguration

Partial reconfiguration provides an alternative to the drawbacks of static and parametric reconfiguration. PR provides the ability to time-multiplex hardware by segmenting the FPGA into partially reconfigurable regions (PRRs) that can be isolated from the static FPGA bitstream.

Figure 2.7 shows how an FPGA can be divided into PRRs where the subsequently labelled bitstreams contain the logic that is mappable to that specific region. The rest of the FPGA can be described as statically reconfigurable, unlike partial reconfiguration, requiring the FPGA to be completely re-flashed in order to reconfigure. A partial bitstream is smaller than a complete static bitstream as the startup and configuration segments are not required for PR, shown in Figure 2.8. PR also provides a mechanism for rapidly updating the behaviour of a system without the need to re-provision the entire static region.

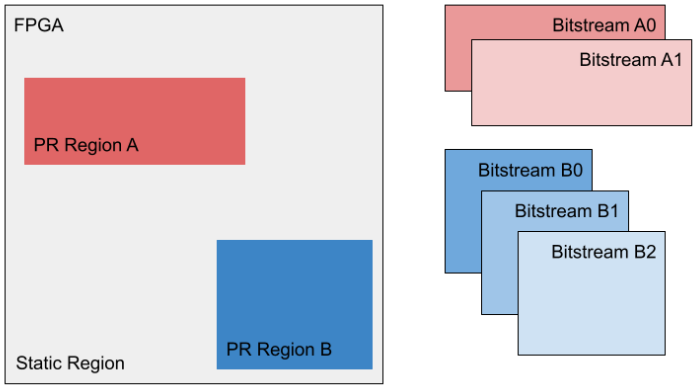


Figure 2.7: Example of partially reconfigurable regions and PR bitstreams

Rather than re-programming the entire device with large bitstreams using slower flashing mechanisms, PR allows for only the PRRs being exchanged to be replaced and can utilise higher throughput interfaces to do so.

Figure 2.8: Xilinx Partial Bitstream Layout (7 Series FPGAs) [4]

As PR enables logic to be time-multiplexed, this allows the datapath required by the accelerator to be drastically shortened as data may be sent through different acceleration logic at different points in time, using PR to replace acceleration hardware as needed. Reducing the required resources of programmable logic to implement behaviour allows for smaller, cheaper FPGAs to be used that provide the same effective acceleration.

Additional accelerator functions (as PR bitstreams) can be stored in non-volatile memory, such as on the attached filesystem of ARM processor, for loading as required, where the fixed datapaths ensure that the PR hardware is mapped to the same memory addresses or interconnects as previous PR logic. A common method of encapsulating PR logic is through the use of static shells, that expect the same interfaces for any PRMs that are loaded within them. This provides an easier design path for building additional accelerator functions but restricts the availability of the interfaces and buses that are connected to the specific function.

Another feature of PR is the ability to enhance fault-tolerance or the ability for a system to recover from faults or errors. A popular use case for FPGAs is in space applications, often used in satellite communication systems, where a high exposure to radiation can lead to store bits being flipped and changing the behaviour of the logic. PR can potentially provide resilience for these systems; if errors are detected in a PR region, they can be rapidly reloaded using PR

and could even be loaded with a more fault tolerant accelerator module. Using methods like dynamic PR would also prevent the need for the dataflow to be halted during operation, an important feature to critical systems such as those required in satellite control applications.

Additionally, it has been demonstrated that PR can have beneficial power savings over classic methods of parametric control [35] as well as reducing PL resource consumption.

The advantages of a system that incorporates PR can be described as:

- Time-multiplexed Logic — The logical resources required for an application drastically decrease as acceleration functions can be swapped in and out on demand.
- Performance for reconfiguration of the PL is improved — Reducing the requirement to reprogram the entire device, reduces the overall time required to configure a device. This also allows for higher performance reconfiguration ports built into the device itself, rather than external ports.
- Power efficiency is improved — a design may be partitioned in a way to allow for regions to be unloaded with PR to reduce power consumption.

It is also important to note that Xilinx's FPGAs can not startup directly with PR and must be initialized with a static bitstream using external flashing mechanisms such as JTAG on first boot.

2.5.2.4 Dynamic Partial Reconfiguration

Despite being used interchangeably in literature, Partial Reconfiguration and Dynamic Partial Reconfiguration (DPR) have slightly different definitions. PR refers to the modification of part of an FPGA, typically described as a region, while the remainder of the device is left unaltered. This does not specify the state of operation of the FPGA; dataflow could be paused and the device held in a reset state during the process of reconfiguration. DPR specifically refers to the state of operation of an FPGA undergoing PR as functional, data can flow through the static and non-reconfiguring PR regions of the FPGA whilst a PR operation is ongoing. This is critical to applications where the system must remain operational. Functional modules that are not critical to the operation of the system can be placed into PR regions and the surrounding static region can be kept operational. For autonomous applications such as UAVs or drones [36], this type of behaviour is critical. Considering that dataflow continues under DPR, designers must consider and account for the reconfiguration time as part of task latency to ensure that real-time critical event deadlines can be

met. Throughout this thesis, PR is assumed to be synonymous with DPR and that it can be assumed that the FPGA is in an operational state during PR.

2.5.2.5 PCAP

The Processor Configuration Access Port (PCAP) is an interface available to the processing system to enable reconfiguration of the FPGA. It both supports static reconfiguration as well as partial reconfiguration and can be triggered from both bare metal and a fully fledge operating system. Under the Zynq-7000 architecture, the processing system (PS) has a dedicated DMA controller that can be used to transfer bitstreams from external memory into the PCAP for reconfiguration. Within the ZynqMP architecture, the PCAP is accessed via ATF which makes requests to the the Platform Management Unit (PMU) and transfers bitstreams from external memory into the PCAP using the CSU DMA driver [15]. The use of the ATF interface ensures that access to programming the FPGA with known bitstreams is restricted to authenticated users at boot or within Linux. The process is complex due to the available security aspects but Xilinx provides an abstraction kernel driver within Linux known as FPGA Manager. This interface is restricted to a maximum throughput of 128 MBytes/s but does not require any resource utilisation on the FPGA itself and thus can be used for the initial FPGA reconfiguration.

2.5.2.6 ICAP

The Internal Configuration Access Port (ICAP) is an IP block provided by Xilinx that abstracts a hardened primitives integrated into the FPGA fabric that allows for reading and extracting bitstream configuration information, known as ICAPE2 for the Zynq and ICAPE3 on the Zynq UltraScale. Figure 2.9 shows the ICAPE2 macro for Xilinx's 7 series FPGAs. The ICAP allows for commands and data concerning reconfiguration to be written to/read from the configuration logic of the FPGA. Unlike PCAP, the ICAP can only be used once the FPGA has been programmed as well as only be used for partial bitstreams as it requires the internal ICAPE2/3 macro, accessible only from within the FPGA itself.

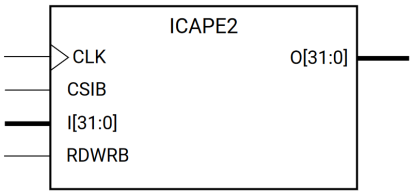


Figure 2.9: Xilinx ICAPE2 Primitive (7 Series FPGAs) [5]

The ICAP, however, is an internal resource on the Xilinx devices and cannot be utilised until the FPGA has been first programmed via an external mechanism such as PCAP.

2.5.2.7 MCAP

The Media Configuration Access Port (MCAP) is an interface on UltraScale FPGAs that enables a similar provisioning mechanism as the ICAP integrated block but for use over PCIe. An initial complete bitstream can be used to establish the PCIe interface and then additional configuration can be performed using the PCIe interface, described as Tandem PCIe. Similarly to ICAP, the MCAP can only be used after initial configuration has been applied. Unlike the ICAP, it does not support a bitstream read-back mechanism.

2.5.3 Development Process

In order to develop acceleration applications for an FPGA, there is an extended workflow that the designer must undertake to realise their designs in hardware. Due to the nature of hardware applications being synthesised into physical logic, as opposed to a sequence of instructions such as with a software program on a processor or GPU, the build tools must take a description of logic and map it onto the FPGA. Given the complexity of designing for individual LUT and wire logic, designers typically use an abstraction known as the register transfer level to describe their logic. Within a register transfer level (RTL) description a designer can model their applications as combinational transformations of the logic in the datapath. Using typical hardware description languages (HDL) such as Verilog and VHDL, the designer can describe registers for storage as well as wires and buses to move data around as well as arithmetic operations that should be applied to the data. There is no fetch, decode and execute path, as per a software paradigm, so the logic described in the HDL must be able to be synthesised into a low-level netlist of logical elements. Next the FPGA must be constrained, the inputs and outputs pins and ports of the FPGA must be defined along with any requirements for timing, i.e. the minimum frequency of the clock that the tools must attempt to establish during implementation. After the constraints have been established and synthesis has been performed, the vendor tools can perform the implementation, where the logic netlist is mapped to the FPGA device itself. This process is known as place-and-route, often a time consuming process where the tools attempt a number of optimisations, in order to most efficiently place and connect the logic on the FPGA such that it will represent the logic as described by the designer. Upon placing logic and meeting the timing requirements of the logic circuit, the tools can then generate a device bitstream, containing all of the

LUT configuration and routing data. This is used to provision the FPGA with the appropriate logic, prior to runtime.

FPGAs are capable of operating as standalone devices, however they are better supported as accelerators for a general purpose processing system. Edge applications that require acceleration are often required to be networked as well as provide resilience for deployment, such as security and update mechanisms, which are better optimised for software centric compute. This means that typically a designer will then need to write software to manage moving data to and from the FPGA and either a physical or soft processor. A designer must consider if they intend to run their application on bare metal or under an operating system as well as consider the interfaces that the processor might have access to the FPGA with. Linux is a popular choice of lightweight accelerators, due to its small footprint as well as flexibility for customisation, especially when managing compute offloading. Common interfaces for managing the functionality of available hardware are the AXI and AMBA bus protocols to move data between memory mapped regions. These interfaces allow for high performance data transfer but generally require complex software drivers to control the datapath.

After understanding the design requirements of the FPGA, processor, operating system and designing the processor to FPGA interface the user can then start to develop their high level applications. These applications may wish to control reconfiguration of the FPGA, demanding knowledge of the subsequent bitstream locations and interfaces, etc.

All of these steps lead to a complex, specialised workflow that forces a high knowledge barrier to entry for designers choosing to leverage FPGAs for acceleration.

2.5.4 PR Design Challenges

PR presents a number of challenges for designers both at build time as well as at run time and deployment. The PR build flow can generate a number of bitstreams that must be appropriately tracked and managed, for the users application at runtime.

2.5.4.1 Abstraction

PR provides a complex challenge to abstract time-multiplexed hardware, in particular when using a software centric control flow, such from a processing system, either hardened like the Zynq or a soft-CPU on the FPGA such as Xilinx's MicroBlaze. PL hardware can be interfaced from the PS through a number of buses, typically through high speed buses such as AXI. Providing an abstraction for managing different methods of transferring data to/from the

PL/PS as well as the factor of time-multiplexed hardware in the PL requires control/decision software to be running at all times. This software is typically described as a runtime manager and should provide a managed method of accessing hardware on the PL as well as reconfiguring it. Typically a runtime will be accompanied by a build workflow that will generate associated metadata for the hardware static and partial bitstreams.

2.5.4.2 Build Workflow

Designing FPGA applications is a low-level process, where the challenges of translating high level behaviour to low level RTL, is already a challenge itself before introducing PR to the process. To design for PR, a designer must follow the standard FPGA workflow of designing an HDL netlist, synthesis, implementation, place and route as well as additional steps for the partitioning and floorplanning of the device. Figure 2.10 shows the steps required for each combination of PRMs to generate bitstreams that will load the respective modules.

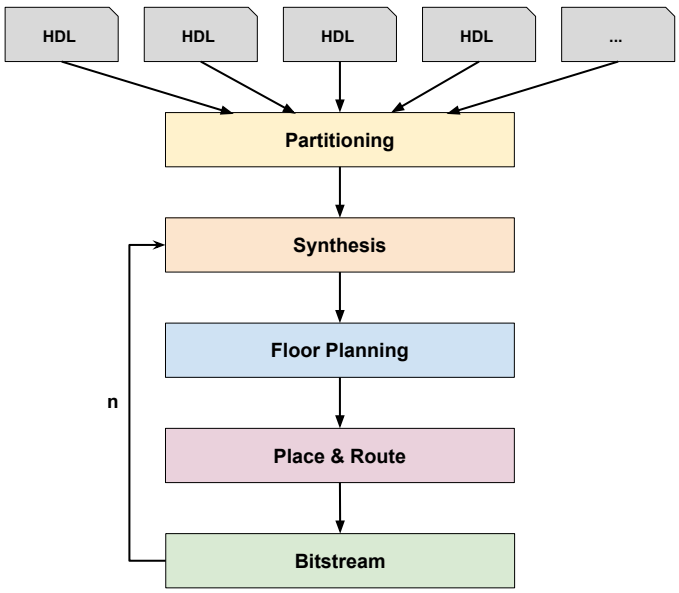


Figure 2.10: PR Build Process where n is the number of configurations required to be generated

To design a PR application there are two sections of the design process, the static regions and the reconfigurable regions (one or more). The static region is the fixed element to the design, that will stay operational during reconfiguration. This region generally manages dataflow as well as control interfaces between the PS. Controllers such as DMA, memory interfaces and bus interconnects will typically be placed into the static regions. The partially reconfigurable regions (PRRs) are the regions that can be exchanged at runtime. Each region

can allow for n partially reconfigurable modules to be implemented. A designer must decide at build time how many regions their design needs as a single region can, in theory, support many modules as long as the resources defined within that region will support it. To start the design process, the FPGA must first be partitioned into the number of regions required by the design. This process is known as floorplanning, where rectangular regions are selected to encompass the various logical elements needed for acceleration modules. There are 3 main types of floorplanning: island, where regions are isolated from one another and should be sized according to the largest implementable module, slot and grid where regions are continuous and can be connected to each other with the difference being the location of regions. The general advantage of slot/grid floorplanning is that reconfigurable modules can be spread across multiple contiguous regions depending on the resources of the module. If slots or grids are small enough, this can decrease the required resources between differing sized modules. Regions are generally selected base upon tiled regions but can be allocated unaligned to tiles as well. Modules as either an HDL or a HLS source are then synthesised. The modules must be synthesised prior to placement to ensure the partial regions have enough resources to support the required logic. The initial placement of the synthesised modules must then be made in these regions (these can be placed as blackboxes or blank functionality if required). The tools can then be instructed to place and route the logic in the FPGA as a static placement, in order to implement additional place and routing runs against, to ensure commonality between PR configurations. The static run is then locked and the process of placing and routing can be repeated for all valid combinations of modules. Upon completion of all of the runs, the tools can perform a design rules check (DRC) to verify that all of the PR runs are valid. Both a partial and a compete static bitstream can then be generated for valid combination of modules.

2.5.4.3 Runtime

An FPGA accelerator could contain a number of hardware features accessible from a processing system that may require drivers or knowledge of such features from software. This problem extend when there are multiple PRRs and can quickly become complex for a user to track and manage hardware interfaces as well as the bitstreams that the PRMs belong to. Each of the PRMs may also have their own memory maps as well as parametric modes and controls; as the system loads new states, the underlying configurations must be tracked somehow.

Given that the PR workflow can generate a number of bitstreams, it is unfeasible for these to all persist in memory simultaneously so a method

for caching and providing fast memory access to bitstreams is important. This might be considered in the form of intelligently caching bitstreams and dynamically managing which bitstreams are ready to be loaded at any point in time.

Since kernel v4.4, Linux has supported the FPGA Manager driver, a runtime which aims to abstract FPGA reconfiguration across a variety of vendors under a unified API. Xilinx has built a lightweight utility that provides control over the FPGA Manager API from the userspace and offers a simple but unmanaged means to flash bitstreams to attached FPGAs. Other vendors such as Intel have their own implementations of the FPGA manager driver that support their FPGA SoC platforms. While this runtime manages the loading of FPGA bitstreams from a processor to an FPGA, it is limited in features and requires the user to have full awareness of the PR bitstream locations and associated drivers.

2.6 FPGAs compared to GPPs, GPUs & ASICs

FPGAs, GPUs and ASIC have found usage as acceleration devices for GPPs; the devices have been compared to each other across numerous academic works. GPUs are excellent at handling image pipelines such as a manipulation of a video feed but cannot be directly connected to a camera interface such as CSI, set up by the processing system. The same task implemented on an FPGA would be highly resource intensive however the FPGA could be connected directly to the CSI via IOB blocks at the edge of the device, reducing the latency and potentially improving performance by bypassing the requirement for a GPP. ASICs may be more performant than the FPGA implementation however are highly rigid and require significant design and time resources to manufacture. Figure 2.11 provides a general overview of where the various devices compare to each other in relation to flexibility, ease of use, performance, efficiency and cost.

[37] demonstrates a stereo vision processing algorithm comparing FPGA performance against an equivalent implementation on a GPU. They conclude that despite a slower internal clock, the FPGA demonstrates superior performance due to extensive pipelining and no longer being susceptible to the I/O transfer time cost that the GPU experienced in order to move images from memory into the GPU. The FPGA did demonstrate limitations in logical resource consumption, struggling to implement hardware that was able to support a disparity range of 256 managed by the GPU. An important consideration is that FPGAs can only support a finite arrangement of logic and unlike a GPU, which will suffer from performance impacts when dealing with complex computation, an FPGA may simply not be large enough to implement

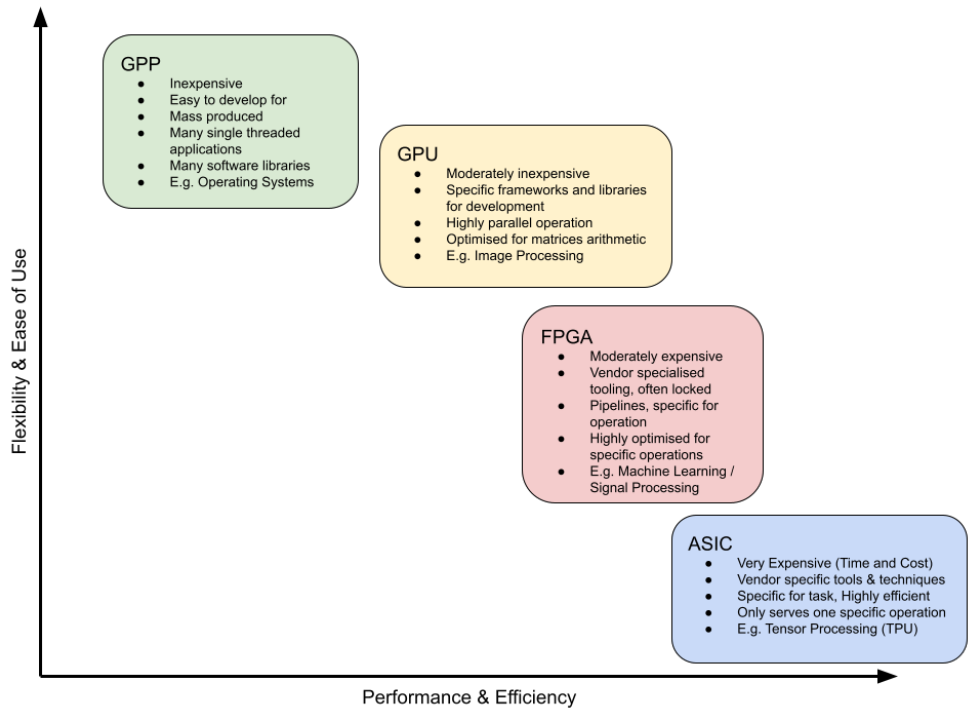


Figure 2.11: General comparison of GPP, GPU, FPGA and ASIC

the required logic.

[38] provides a comparison of FPGA, CPU, GPU and ASIC accelerations of an RNN algorithm, Gated Recurrent Network (GRU). Their results demonstrate that generally the FPGA was more efficient with better peak performance, at 10× performance/Watt. GPU/CPU performance was improved by batching data but not significantly enough to compete with the FPGA, which was more performant and did not suffer from the increased latency and complexity of batching. As expected the ASIC outperformed the FPGA, where the FPGA was 7× less efficient than the ASIC. Cost should also be considered when evaluating a target compute platform as fabrication of an ASIC is significantly more expensive both in cost and time than the use of a GPP, which are plentiful and cheap to develop for.

Developing applications for GPUs is dissimilar to FPGA development as typically developers can write applications in higher level software languages such as C, C++, Fortran, Python and MATLAB, that leverage GPU specific frameworks such as OpenCL [39] and Nvidia's CUDA [40] libraries. These tools abstract the complexities of operation parallelism to best serve the GPU's architecture and allow developers to access acceleration via APIs. OpenCL enables developers to rapidly build applications that can leverage the parallelism of the GPU. FPGAs require a lower level understanding of digital logic as well as knowledge of the target device's features. This domain specific knowledge is

typically a contested point for building FPGA accelerated applications where designers require highly specialised knowledge. Recent developments in HDL design have seen the introduction of High Level Synthesis Languages (HLS) for developing hardware applications. HLSs enable designers to work at higher levels of abstraction and can use programming language paradigm to describe the algorithms they intend to implement. While these languages, such as Xilinx's Vitis HLS [41], have similar constructs to the C++ language and can be co-simulated against C++ testbenches, still require the designer to be aware of the context of their designs. Vitis HLS supports macros such C++ functions to instantiate RLT module interfaces as well as simple mechanisms to parallelising tasks via unrolling loops. Open source HLS languages also exist such as the Scala-based Chisel [42] and nmigen [43], written in Python. Developers must still design with awareness that their code will be synthesised to physical hardware and traditional software paradigms will not directly translate to an FPGA implementation.

2.7 Heterogeneous SoCs

Heterogeneous System on Chips or FPGA SoCs are devices that integrate both a hardened processor and FPGA device into a single silicon device. Rather than standalone processors and loosely-coupled FPGAs, FPGA SoCs offer lower power, higher bandwidth between devices as well as better integration. These devices often include high-speed transceivers, a wide range of hardened peripherals as well as on-chip memory. Integrators of the ARM architecture, such as Xilinx and Intel's Altera, have coupled ARM processors with high capacity FPGA devices, to provide flexible platforms for building and designing customer hardware for acceleration. A number of vendors offer FPGA SoC devices such as Microchip's PolarFire SoC, which couples a RISC-V architecture processors with an FPGA.

A major advantage of a hardened ARM core is that it frees the FPGA from having to support complex logic required for a soft CPU of equivalent performance. Additionally, given that the FPGA is considered as a subsystem of the CPU, the CPU and FPGA can be clocked independently, allowing the CPU to run at traditional clock speeds of 1 GHz and higher. The two major vendors, Intel and Xilinx offer a number of architectures integrating their FPGA devices with processing systems.

2.7.1 Intel

Intel offers the Agilex-F, Agilex-I, Stratix 10, Arria 10, Arria V and Cyclone V families of FPGA SoCs, as shown in Table 2.2. Majority of these device families

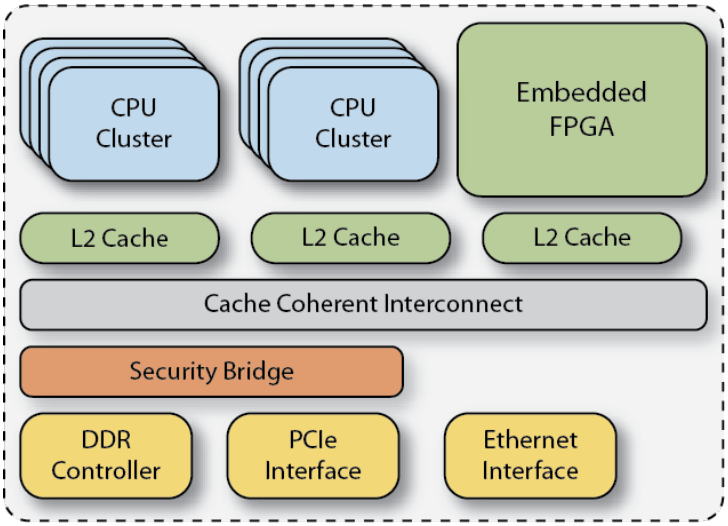


Figure 2.12: Generic FPGA SoC Architecture [6]

Table 2.2: Intel FPGA SoC Families

Family	Processor	FPGA	Arch.	Cores	Year
Agilex-I	Intel Xeon & ARM A53	10nm	x86 + arm64	n + 4	2019
Agilex-F	ARM A53	10nm	arm64	4	2019
Stratix 10	ARM A53	14nm	arm64	4	2013
Arria 10	ARM A9	20nm	armv7	2	2013
Arria V	ARM A9	28nm	armv7	2	2011
Cyclone V	ARM A9	28nm	armv7	1	2011

support ARMv7 or ARM64 processors with the exception of the Agilex-I, which provides a cache and memory coherent interconnect to Intel's Xeon platform of server-grade processors. Despite Intel's main target architecture being x86 based (for personal computing and enterprise servers), many of its FPGA SoC families are targeted for embedded deployment and/or acceleration, thus more commonly use ARM-based processing systems. The Agilex-I series is the exception provide acceleration support for Intel Xeon (x86), however it is more common in datacenter applications to use PCIe based FPGA acceleration cards instead of tightly coupled FPGA SoCs.

2.7.2 Xilinx

Xilinx offers the Versal, Zynq Ultrascale+ (MPSoC & RFSoC) and the Zynq-7000 families of FPGA SoCs, as shown in Table 2.3. Similar to Intel's offering, these FPGA SoCs are based on ARM architectures, targetted at embedded

Table 2.3: Xilinx FPGA SoC Families

Family	Processor	FPGA	Arch.	Cores	Year
Versal ACAP	ARM A72	7nm	arm64	2	2019
Zynq Ultrascale+ RFSoC	ARM A53	16nm	arm64	4	2017
Zynq Ultrascale+ MPSoC	ARM A53	16nm	arm64	2 or 4	2015
Zynq-7000	ARM A9	28nm	armv7	1 or 2	2011

devices for accelerating the ARM processor, in field applications. At this time the Versal ACAP family is a newly release product but has limited academic or commercial research has been performed against these devices and is not targetted at embedded applications.

2.7.2.1 Zynq-7000

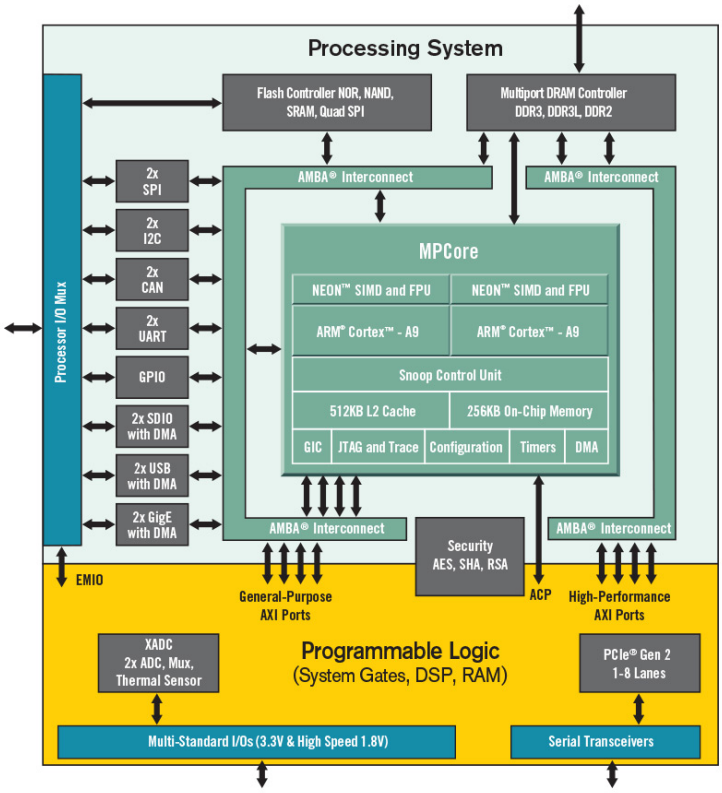


Figure 2.13: Zynq-7000 Architecture [7]

The Zynq-7000 or Zynq series are Xilinx's initial FPGA SoC offering, their first product to offer a tightly coupled in-die FPGA and ARM processor SoC. This FPGA SoC leverages the general purpose flexibility of the processing system with the dynamic reconfiguration and performance of the programmable

logic to provide capable acceleration.

Figure 2.13 shows the Zynq's architecture, describing the interfaces available between the PS and PL. The architecture of the Zynq devices supports a number of high performance interfaces between the hardened ARM cores and the programmable logic. The most basic communication method is a general purpose PS master interface where the CPU can move data into two of the master AXI general purpose ports on the PL via an addressable memory map. The PL interface receiving this data can be a single AXI slave to receive CPU requests; this method is relatively slow as 25 MB/s as the CPU must spend compute cycles to perform these operations rather than on other more complex tasks. An alternative higher performance approach is to use Direct Memory Access from the PS DMAC or with a PL DMA Controller. The PS DMAC is quoted at being able to perform at 600 MB/s and a dedicated PL DMA Controller at 1200 MB/s when using a high performance interface on the PL. The disadvantage of the PL DMA Controller is that it required potentially limited resources from the FPGA fabric, where as the PS DMAC is a hardened controller driven directly by the PS. Both the Zynq and ZynqMP architectures support these three methods of PS-PL data transfer.

2.7.2.2 Zynq Ultrascale+

Xilinx's Zynq UltraScale+ MPSoC (ZynqMP) is positioned to compete directly with Intel's Stratix 10. It is available with 1.5 GHz dual or quad ARM A53 APU cores, a Mali-400 MP2 GPU and FPGA. Additionally it supports a dual-core realtime processing unit (RPU), that is designed to be highly deterministic and low latency. It advances the platform introduced by the Zynq-7000 with a new 64-bit architecture, modern FPGA structure built on a 16 nm fabrication process. The ZynqMP architecture contains a platform management unit (PMU), that is responsible for number of features including secure access to peripherals and memory as well as power and programming of the FPGA. The programmable logic is based on Xilinx's Ultrascale+ design, with a range of interconnect options, DSP blocks and hardened marcos.

Xilinx also offers a variant of the ZynqMP, the RFSoC, specifically designed for radio applications, such as software defined radio and 5G wireless.

2.8 Operating Systems (Linux)

An operating system (OS) is a collection of software layers that manages hardware such as IO, system resources as well as provide interfaces for applications to interact with. A common feature of an OS is the ability to efficiently schedule tasks for allocation of processor memory and time, system resources as well as

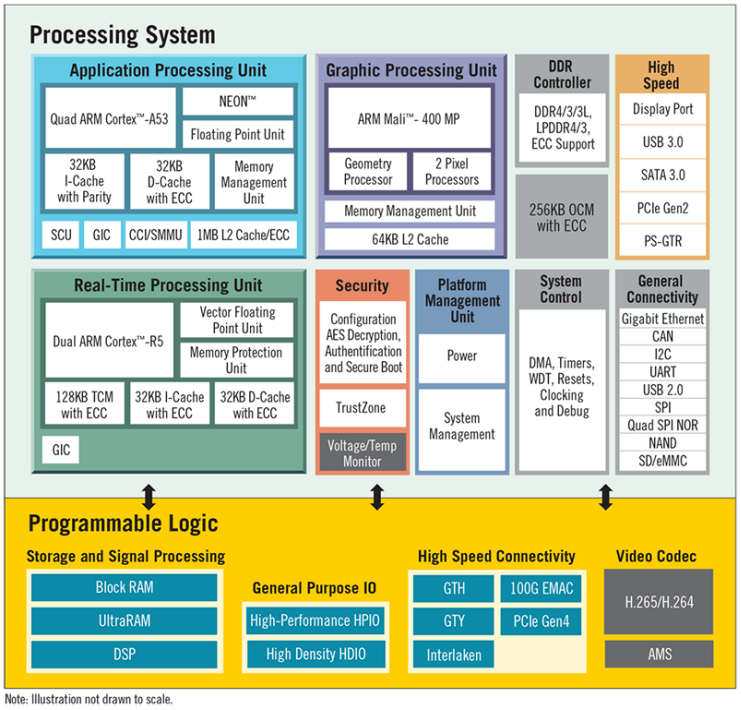


Figure 2.14: Zynq Ultrascale+ Architecture [44]

hardware access.

Linux is a popular OS designed to be modular and capable of supporting a large number of processing architectures, including x86 and ARM64. While other operating systems such as Real-Time Operating Systems (RTOS) are popular for adaptive systems, Linux has risen in popularity due to open source nature as well as wide support for programming paradigms and languages. Linux is designed to abstract low level and potentially security risky operations to its direct interface with hardware, the Kernel. The Linux kernel has direct access to physical hardware that is attached to the processing system, in the case of an FPGA SoC, this includes the FPGA fabric. Typical user applications will run in the userspace, an abstraction within Linux to isolate applications from having direct access to hardware and physical memory addresses. This abstraction exists to provide isolation that serves to protect memory and hardware from malicious behaviours. The Linux userspace uses virtual memory mapping to allocate processes their own sandboxed memory space that restricts access to the memory of other processes. User processes and applications will communicate with hardware accelerators through the use of kernel drivers, that abstract low level functionality with high level system interfaces such as *sysfs*, a pseudo-filesystem that provides an interface to kernel data structures.

2.9 Summary

Acceleration functions are important to a range of applications, where general purpose processors are inadequate to handle the computing demands, whether that is performance, latency or parallelism. Many alternative processing platforms have both been proposed and many accepted, such as GPUs for video processing, however FPGAs have shown to be a promising high performance, yet flexible custom solution to acceleration. Partial reconfiguration further increases the flexibility, power efficiency and cost savings of FPGAs as compute offload as it enables adaptive systems that support rapidly configurable accelerators. However, while PR has significantly matured since its introduction, the tools remain difficult to use for a designer without bespoke knowledge. The current workflow requires a number of designer inputs across the design process, requiring low-level knowledge of both FPGA and CPU architectures to communicate between devices. This is further complicated at runtime where the designer is expected to manage the abstraction of PR specifically for their own applications, managing this at an FPGA, Kernel and Userspace level. It is increasingly important that these tools are made more accessible to software designers to provide automation and abstraction to non-experts in order to encourage wider adoption of PR. We believe the research and tools presented in this thesis begin to democratise the PR workflow and enable a new audience of autonomous systems designers to build FPGA accelerated edge applications.

Chapter 3

Literature Review

In this Chapter, we review the state of dynamic partial reconfiguration, design tools for PR as well as PR management runtimes. We investigate different approaches to streamlining the development process of PR accelerated applications for FPGA SoC deployment, examining both vendor and academic build and runtime tooling.

3.1 Design Methodologies

To understand how PR applications are currently developed as well as the advantages and disadvantages of these techniques, both design methodologies from device vendors as well as current academic literature are described.

3.1.1 Vendor PR Tools

This thesis focuses on Xilinx's FPGA SoC devices and the associated toolflows; while Intel's FPGA SoCs support PR, the availability and documentation of the PR process and tooling is significantly less mature than Xilinx's offering, hence our focus on Xilinx. The design process for both major vendors is broadly similar, where the differences are related to the architecture specifics of the FPGAs.

3.1.1.1 Vivado Design Studio

Xilinx offers the Vivado Design Studio for FPGA application design, testing and deployment. Vivado is a visual design tool (with GUI) that is built with a scripting backend, written in the TCL programming language. It offers three options for workflows: GUI-based where the Vivado IDE is used to control the design process, TCL-based where the user can interact with a command-line to pass commands into/out of Vivado and a mix of GUI/TCL automations. The Vivado suite provides tools for verification, the functional and behavioural testing of logic, Synthesis, the translation from HDL/HLS to RLT as well as

Implementation, for placing, routing and optimizing logic on an FPGA. Within Vivado a designer can leverage Xilinx's workflow methodologies for optimising their implementation for power efficiency, optimal clock frequency as well as other recommendations. Recently Xilinx added support for a board store, which allows designers to access a repository of boards and example designs that are setup with constraints for the designer to immediately start using.

3.1.1.2 Dynamic Function eXchange

Dynamic Function eXchange (DFX) is Xilinx's PR workflow and tooling, introduced with Vivado Design Studio 2019.2. DFX is a collection of tools provided by Xilinx to reduce some of the complexities associated with designing partially reconfigurable applications. It specifically targets platform-based design flows, such as for the Alveo FPGAs (PCIe-based), however this workflow is now the default for Xilinx FPGA SoCs, the Zynq and ZynqMP. DFX allows for hardware designs that use the TCL-based non-project flow or RTL/IP-based project flows. DFX provides a number of IP cores that manage aspects of PR like isolating the PRR during reconfiguration, managing AXI interfaces to prevent data loss during PR as well as IP cores to monitor and debug PR bitstreams.

Previously, DFX designs demanded a rigid static structure across the whole device in order to match the PRMs with existing implementation run. This requires the complete locked static region to be used for context when placing and routing for PRMs. This is both RAM heavy and potentially a vulnerability for proprietary designs as the static region must be loaded into memory for the implementation of PR regions. Recently Xilinx has introduced two advanced concept known as Nested DFX and Abstract Shell [8] to their DFX workflow.

Nested DFX enables the placement of one or more RPs within an already existing RP, to permit further granular reconfiguration. This means that smaller acceleration functions can be placed more conservatively when a larger function might also exist. At this time it is a feature only supported by the Ultrascale+ devices, including the ZynqMP.

DFX Abstract Shell is a workflow available for UltraScale+ FPGAs, in Vivado Design Studio 2020.2, that allows a trimmed down static design to be used for RPs. This is a static layer required for the minimum context needed to implement an PRM and generate a PR bitstream for that RP. The logical shell contains the boundary interfaces for the PRMs and the Pblock of the partition is captured in the design constraints including any clocking or boundary timing requirements.

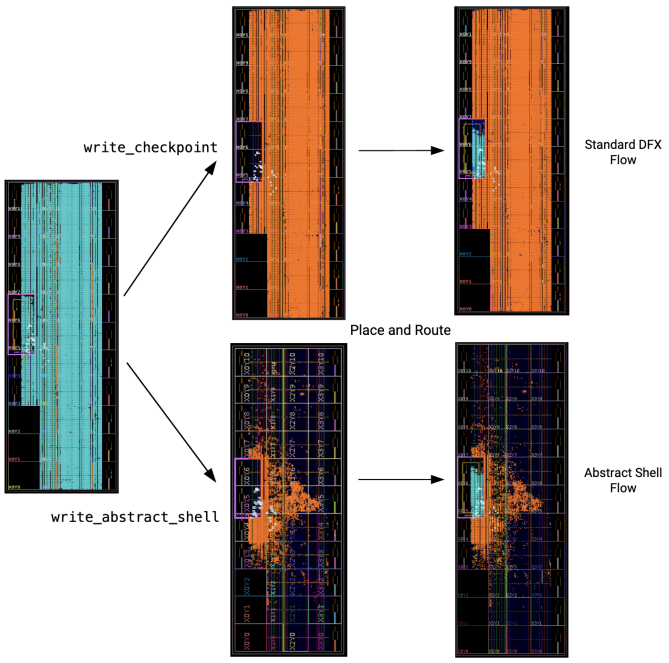


Figure 3.1: Standard DFX vs Abstract Shell implementation logic [8]

The DFX Abstract Shell process allows the implementation requirement of the static region to be separated from the PRM using a trimmed down version of the static design based upon a given RP. The abstract shell is effectively a minimal design image required for implementing a new PRM and generating a partial bitstream for those modules. Figure 3.1 shows the reduction in static logic required to place and route PRMs from the standard DFX and the Abstract Shell workflow. Abstract Shell also provides advantages for multi-user designs; as a minimal static region is required, this can be provided to acceleration function designers enabling proprietary or custom static logic to be restricted.

3.1.1.3 Vitis HLS

Vitis HLS allows designers to easily create complex FPGA-based functions using C/C++ and OpenCL code. The Vitis high level abstraction provides a number of utilities to simply the design process for software designers looking to leverage HLS to generate RTL. These accelerated functions can either be deployed as part of the Vivado workflow or as RTL Kernels for the Vitis IDE workflow. Vitis HLS enables the generation of complex interfaces through the use of `#pragma` keyword within C/C++ functions. This can be used to rapidly build powerful hardware functions from C/C++ code where traditional software paradigms such as `for` loops can be unrolled and pipelined for synthesis. There are a number of libraries for Vitis HLS that enable the rapid deployment of acceleration functions such as the Vision, AI and Security libraries. Figure 3.2

```
void example(int A[50], int B[50]) {
    #pragma HLS INTERFACE axis port=A
    #pragma HLS INTERFACE axis port=B

    int i;

    for(i = 0; i < 50; i++){
        B[i] = A[i] + 5;
    }
}
```

Figure 3.2: Example of Vitis HLS Pragma for AXI Stream Slave/Masters.

shows the minimal code required to generate an IP core with an AXI Stream Master and Slave interface that takes the incoming stream and appends an integer value of 5 to each element of the stream. While this example is compilable as C++, designers still must be aware of the constructs that HLS will generate upon compiling and synthesising this code into logic. The abstraction is increased but the fundamental implementation still requires knowledge of the tools and their ability to interpret the designer's code.

Co-simulation is also possible where the behaviour of an HLS design can be tested using a software based C++ test bench. This makes it easier for designers to pass complex data objects such as waveforms or video frames into a design for testing.

Xilinx's Model Composer [45] is a DSP design toolbox for MATLAB's Simulink tool that offers similar abstraction features to Vitis HLS. In Model Composer, users can utilise a library of HDL, HLS, and AI Engine blocks for the design and implementation of acceleration functions on Xilinx devices. The Model Composer workflow is closer to visual block design tools than Vitis HLS but provides an alternative workflow that offers high level abstractions. Xilinx's System Generator [45] for DSP is a lower level add-on for Simulink that generates prepared IP cores, exclusively for Xilinx FPGAs, that can be imported directly into Vivado. Similar to System Generator, MathWorks provides MATLAB HDL Coder [46] which allows the designer to generate VHDL and Verilog code from MATLAB functions or Simulink models. This enables designers familiar with MATLAB to rapidly design HDL code without needing to be concerned with low level complexity. Additionally it provides a simple toolchain for simulating HDL logic directly in Simulink and outputs HDL so can be used with non-Xilinx based devices. Both System Generator and HDL Coder are lower level abstractions than Vitis HLS or Model Composer and are intended for RTL level design rather that automatically generating bus interfaces like AXI and AXI Stream. These tools specifically focus on generating specific accelerator hardware logic, rather than the integration into

a complete application.

While Vitis HLS helps to ease the transition from software to hardware paradigms, it still requires in-depth understanding of how the HLS compiler will generate hardware based upon its description in C++. Underlying mechanisms, such as the depth and width of bus interfaces, must be specified by the designer or otherwise will be inferred by the compiler. This can lead to undesirable consequences for the optimisations and implementation provided by the tooling.

3.1.1.4 Vitis & PetaLinux

PetaLinux is a Linux build tool that enables designers to build and deploy embedded Linux for Xilinx based processing systems, including the Zynq and ZynqMP devices. It is built on top of the Yocto toolchain and extends the standard Yocto flow with Xilinx specific application, driver and library generators. It provides a number of tools for testing and debugging Linux applications such as debug agents as well as GCC and GDB tooling. The PetaLinux tools allow a designer to automatically generate a custom Linux Board Support Package including Xilinx device drivers for embedded IP cores, kernel and bootloader configurations. The tools are able to import Vivado-exported hardware along with specific data files that allow for device drivers from Xilinx embedded IP cores to be automatically built and deployed according to specified addresses of that device. Upon generating a Linux image (BSP, device drivers, core applications, etc.), the tools allow developers to package libraries and software components to use in the Vitis IDE for building applications.

The Vitis IDE supports a number of development flows for edge, server and cloud deployments. For edge development flows Vitis IDE allows for C/C++ applications to be developed and accelerated with programmable logic functions. Vitis IDE can import and use exported Linux kernel headers from PetaLinux for building applications that are intended to run on the built Linux image. The IDE also has supplied tools that allow for remote application development of C/C++ applications directly on Xilinx devices.

The PetaLinux workflow helps to abstract some of the complexity from the underlying Yocto toolchain but generally only works well when the designer follows the exact design patterns intended by the tools. For example, PetaLinux does not include a workflow for PR based designs as the tools are both unable to import metadata from the FPGA build process to assist with Linux kernel design as well as abstract, manage or import any of the partial bitstreams generated by Vivado.

3.1.1.5 Limitations

Vendor tools offer low level control for designing and developing FPGA applications, that typically focus fixed or static applications with limited support for interchangeable hardware such as with PR. For example, using Xilinx's current workflow, any change in IP core or user hardware (in particular, within a PRM) will result in breaking effects throughout the rest of the build process, when targetting a Linux based application deployment. PetaLinux is unable to handle PR-based designs and the interfaces and drivers requires to communicate with PRMs is not propagated from the FPGA build process. A designer must manually configure each build run to support their intended PR design and must replicate this for as many accelerators as they wish to deploy. Recent tooling, such as Abstract Shell, has made the FPGA process more streamline but still cannot handle exchanging hardware information, such as PR-linked memory maps or interfaces, from the FPGA build workflow to the Linux OS image creation. Additionally, these tools require a high level of knowledge and expertise across varying layers of abstractions, from low level RTL design for the hardware accelerators, to writing custom Linux drivers to communicate with the FPGA as well as designing high level applications to leverage and accelerate software using PR.

3.2 Academic PR Tools

The complexity in the FPGA workflow has generated numerous works to automate and better manage the tools surrounding PR application design, development and deployment. Research has tackled aspects of the design from designing shell-based accelerators to scheduling the tasks that are accelerated in the PL.

3.2.1 Build Workflows & Floorplanning

Numerous works have been presented for optimising FPGA floorplanning but less so for the challenge of floorplanning for PR applications. This Section explores build tools, including the automation of floorplanning and Linux building, designed to ease/simplify the FPGA PR application design process. A further tabulated comparison of these tools are discussed in Sections 5.3.2 and 6.3, where they are evaluated against our own tools.

RAMPSoC [47] presents an early approach to combining multi-processors and reconfigurable computing with run-time reconfiguration. Their tool provides a system of exchangeable processing elements and communication interfaces within the programmable logic to adapt to the demands of algorithms running within software. Their system supports run-time integration of special-

ised processors including RISC, CISC and ASIP architectures and hardware accelerators to achieve a balanced workload fo computing elements. They suggest this approach targetted the gaps presented by homogeneous hardware and allowed to adaptation of parallelized modular application blocks and/or tasks by providing a processor or computation element with the required bit-width or acceleration features for the target task.

ReCoBus [48] is an early PR design tool that provided a number of advantages for development including a hybrid communication bus that connects hardware accelerators to a main processor as processing units. It uses dedicated connections to enable stream-based communication between contiguous hardware tasks. ReCoBus uses templates for accelerators which enforces placement and routing constraints to simplify how hardware changes are generated within the FPGA. Users are allowed to use fixed slots to implement their acceleration logic using more or more as required to implement their application. Acceleration modules can communicate with adjacent modules across as stream-based interface however a PLB-ReCoBus (Processor Local Bus) bridge is required for access from the main system bus as well as to provide access to main memory.

GoAhead [49] is a PR floorplanning tool designed for Xilinx's toolchains. It advances on the work presented in ReCoBus to provide further flexibility for PR design, such as with its scripting interface. Their design flow starts with a designer synthesising their design to generate resource utilisation reports for their static and partial modules. The reports are then used by GoAhead which attempts to floor plan the device while optimising for size and location. GoAhead masks the PR regions with blocker macros that occupy all wires inside the PR regions while implementing the static region. This prevents static nets from crossing into PR regions, which is an invalid state for PR reconfiguration. The PRMs are similarly implemented; blocker macros are used to prevent wires crossing into the static region. GoAhead can also extract and generate custom interfaces between PR modules.

CoPR [50] is a toolflow that focuses on the abstraction layer for PR applications on the Xilinx Zynq. The designer uses *configurations* and *adaptations* to specify system states. They define configurations as valid system configurations and the corresponding library modules declared in an XML file. Adaptions specify the software code required to change configuration at runtime. CoPR does not support Vivado or the ZynqMP architecture.

IMPRESS [51] is a TCL based tool for automating the generation of relocatable PR bitstreams within Vivado. Their tool specifically focuses on the implementation for PR systems that include Vivado HLS blocks and that used standardized buses such as AXI. Their tool has a number of advantages over vendor (Xilinx) workflows, including its ability to allow for relocation of RMs, stacking of RPs with a single clock region, hierarchical reconfiguration

and decoupling the implementation of static and reconfiguration regions. At the time of writing, their tool only supports the Zynq-7000 and 7 series FPGA devices.

BITMAN [3] is an open source tool and API for generating and manipulating bitstreams. It allows for the placement and relocation of FPGA modules for a variety of applications including PR. PRMs may cross a number of CLBs and BRAM columns and BITMAN enables the reconfiguration without effect on other surrounding modules or the static regions. This is performed by using routing constraints on the static routing using a tool such as GoAhead. BITMAN also supports the ability to update LUTs and BRAMs within the FPGA at runtime; this is particular valuable for parametric reconfigurable applications, such as changing filter coefficients or updating cryptographic keys. They also provide a demonstration of BITMAN being used to stitch together Coarse Grained Reconfigurable Arrays (CGRAs) with a processing element (PE) library as explained in [52], extending the original work by stitching the PEs at a bitstream level rather than at the netlist, drastically decreasing the time required to generate a complete bitstream.

RapidWright [53] is an open source pre-implementation tool designed to help achieve higher performance and\or productivity when designing FPGA applications. Their approach increases the modularity at design time by exporting design checkpoints throughout the Vivado design run enabling design checkpoints to be manipulated for better reusability and performance. RapidWright allows for strategic injection of pre-implemented modules with programmable fabric structures allowing the design process to be deconstructed and use out-of-run implemented logic to speed up design time. This tool has enabled similar strategies to BITMAN for injecting and stitching application specific overlays [54] at build time. The work in [54] showed that they were able to demonstrate a productivity improvement up to 20× compared to the state of the art FPGA overlays, while achieving over 1.33× higher maximum clock frequencies than a direct FPGA implementation along with the possibility of lower resource and power consumption compared to bare metal implementations on a processor. While RapidWright is capable of producing pre-implemented modules as partial IPs, it is not able to generate partial bitstreams independent of the static design [55].

ReConOS [56, 57] is a framework that is designed to extend the multithreading programming model from software to heterogeneous computing with reconfigurable hardware. They use threading and common synchronisation to abstract hardware, allowing for portable and flexible multithreaded FPGA SoC applications. Their automated toolchain provides a scriptable set of Python and TCL scripts that take an input specification for threads in one of the supported languages and generate configuration bitstreams for the target FPGA.

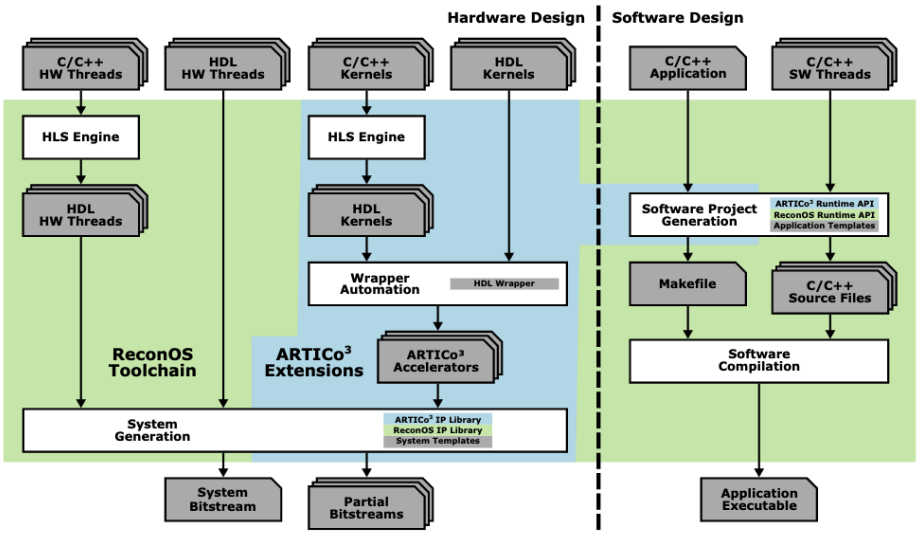


Figure 3.3: ReConOS toolchain with ARTICO3 extensions [9]

This automated process is controlled by a configuration file that uses a specific hardware template to define the constraints of the target system, including FPGA device, operating systems and any required hardware IPs. The same configuration file is used to generate a software project for ReConOS that is capable of leveraging the multi-threaded paradigms.

FUSE [58] is a similar approach to ReConOS that utilises a slot-based reconfiguration system with embedded soft MicroBlaze processor to manage POSIX threads. Shared memory elements are used to exchange data between software and hardware tasks to attempt to reduce transfer overhead. FUSE uses a loadable kernel module (LKM) to abstract hardware as memory mapped I/O peripherals to simplify managing attached accelerators. Under FUSE, updating a hardware accelerator only requires changes to be made to the hardware interface and corresponding LKM rather than the user's application or the userspace aspects of the FUSE software.

ARTICO3 [9] is a toolchain for the Zynq and ZynqMP that offers a reconfigurable processing architecture based upon custom kernels that are assembled as part of their workflow. Their toolchain includes both the build tools [51] for automating hardware acceleration as well as a runtime for managing reconfiguration and hardware acceleration offloading. They describe their architecture as a DPR-enabled processing architecture for task and data level acceleration that can trade off performance, energy efficiency and fault tolerance. Their toolchain takes HDL or HLS input sources and wraps them within a custom kernel that provides local memory, configurable register banks and the required interconnection to the main ARTICO3 infrastructure as shown in Figure 3.5. Software applications can be written to leverage these acceleration kernels by using a standardized API as accelerator logic is abstracted behind the kernel

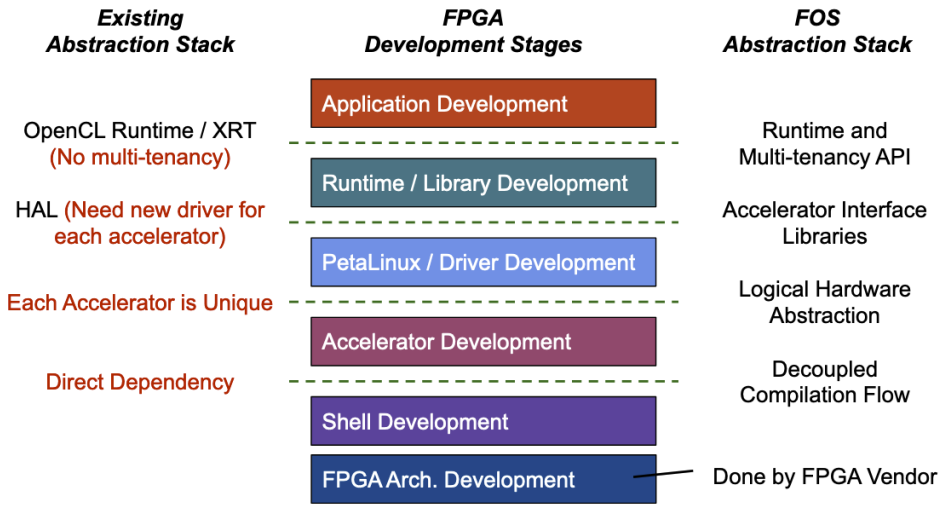


Figure 3.4: FOS compared to traditional development abstraction [10]

interfaces. ARTICO3 shares the base toolchain with ReConOS, where the ARTICO3 extensions are shown in Figure 3.3.

ZUCL [59] & ZUCL 2.0 [60] are a collection of abstraction services for hardware applications on ARM-based FPGA SoCs. The original ZUCL framework was designed to enable the development and deployment of OpenCL applications onto ZynqMP devices. Their framework exploits PR to provide a plug-and-play approach for loading/unloading hardware modules, where an automatic compilation process implements the OpenCL modules directly into relocatable PR modules. This enables a software-centric approach to using OpenCL for acceleration. ZUCL 2.0 extends the original work by providing a number of advancements with support for: 1) decoupled implementation, where the static shells can be implemented separately to the PRMs 2) a number of HDLs, HLSs as well as directly from netlist 3) variable bus interfaces (e.g. 32/64/128 Bit AXI Stream/Master/Slave) 4) various floorplanning strategies 5) hardware context switching to enable resource allocation 6) memory isolation for multi-tenancy applications. They evaluate their framework with context to performance limitations of the PCAP controller for loading PR bitstreams; both tools use the PCAP interface for control of reconfiguration.

FOS [10] offers an alternative approach to developing FPGA accelerated applications by modularising the workflow and splitting it into system components that can be developed by independent domain experts without context of other aspects of the workflow. Their build process attempts to provide a decoupled and agnostic approach to handling variation in EDA versions as well as related software and hardware. Their workflow, provided it can be built using the FOS tools, should work across tooling versions; For example,

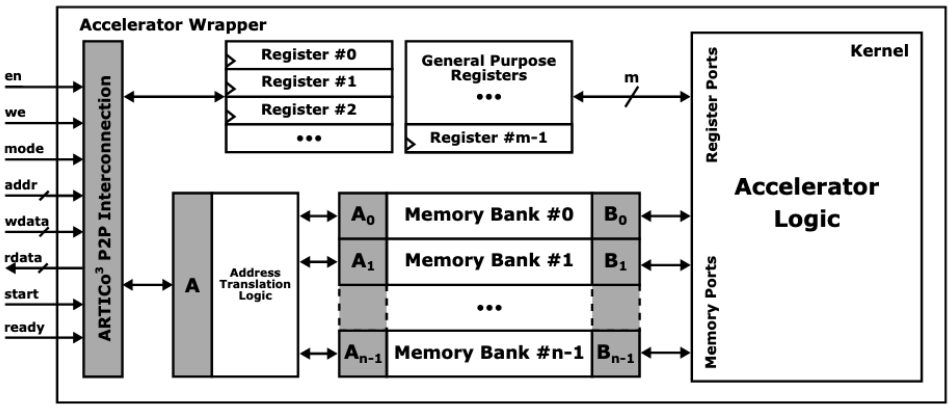


Figure 3.5: ARTICO3 kernel wrapper [9]

importing an IP core that was synthesised in Vivado 2018.2 into Vivado 2019.2 will require re-synthesis as the changes in the synthesis mechanism across tooling cannot be guaranteed however FOS uses the BITMAN [3] tool to extract and relocate FPGA modules and thus reduce this issue. This process also significantly reduces development time as the end-to-end process does not need to be repeated every time a new accelerator is introduced. Figure 3.4 shows the various stages of the development process between the traditional and the authors abstracted approach. This highlights the issues of non-reusable hardware and software dependencies found in the traditional abstraction stack and where FOS addresses them.

While many of these tools attempt to address the issues with prerequisite knowledge and domain expertise, such as FOS with its modular workflow, they often expect and enforce a shell based workflow. The problem with this type of workflow is that a designer must learn and understand how RTL modules function and communicate, requiring bespoke hardware knowledge. In addition to requiring the designer to modify their IP cores to fit the shells, typically these frameworks also require their own APIs and control flow mechanisms for interacting with accelerators. An alternative for generalised acceleration could be to completely abstract the complexity of the accelerator from the user, such that they can accelerate their software without needing to understand how to directly control the hardware via wrapping the user's hardware and exposing hardware to software via generalised userspace memory maps and/or DMA transactions.

3.2.2 PR Runtime Management

ReconOS [61] provides a unified hardware/software multithreading programming model for processing acceleration across a central bus that is connected to the processor, for a Linux based target. The hardware components of the

ReconOS runtime consists of interfaces, communication channels as well as memory access and address translation to the hardware functions. ReconOS offers the ability to perform multithreading across the hardware/software paradigm where hardware threads (e.g. acceleration functions) can request functions to be executed from the OS via their OS interface (OSIF) state machine. Access to the OSIF is performed via a VHDL library that abstracts the OS calls and is provided to the designer of the application. From the OS perspective only software threads exist and hardware threads are abstracted from their delegate threads. Alternatively from the application designer's perspective, delegate threads are abstracted with only hardware and software threads existing. Threads can communicate with a number of established OS techniques such as message queues or mailboxes, barriers or semaphores or with mutually exclusive locks. This enables high levels of transparency of execution for functions running on either hardware or software, adding some additional overhead for the benefit of reduced complexity. To perform reconfiguration of hardware, ReconOS uses Xilinx's FPGA Manager for the Zynq and ZynqMP.

CAP-OS [62] is an early runtime scheduler for task mapping and resource management on reconfigurable architectures that uses the RAMPSoC [47] architecture. Their scheduler was implemented on top of a Xilkernel RTOS on a IBM PowerPC 405 with support for the Virtex-4FX FPGA. While now based upon older, less relevant hardware, this OS provided a foundation for a number of works to address managing the abstraction of resource allocation.

Xilinx's FPGA Manager provides an driver level API for reconfiguration via the Zynq and ZynqMP's PCAP. This flashing process can take several milliseconds and scales linearly with the size of the partial bitstreams. This is non-negligible in comparison the performance of the processing system, requiring the system to pause during reconfiguration to await completion before communication with the FPGA can be resumed.

LinROS [63] is a runtime layer that utilises a novel Linux driver to automatically manage the software and hardware of reconfigurable MPSoCs. It has an accompanying IP core to facilitate the integration of accelerator functions using HLS based tools. The IP core manages the data exchange between the accelerator functions and the processing system. Using an image processing algorithm on the Zynq SoC, they demonstrate how negligible overhead is introduced under their runtime during scheduling. The programming of the accelerators functions is abstracted by their device driver and uses the PCAP interface for provisioning. LinROS is also able to schedule software threads across multiple MicroBlaze processors.

The ZUCL [60, 60] runtime uses a combination of kernel and userspace drivers. Their focus is on sandboxing of hardware application and enabling memory isolation by utilising the System Memory Management Unit (SMMU)

in the ZynqMP's PS with a custom driver. They provide userspace memory access via a userspace library that creates handles that associate the user's page table and accelerators with the SMMU. It is shown that the SMMU adds non-negligible overhead to initial DMA transfers.

FOS [10] makes use of a resource-elastic scheduler to manage FPGA resources both with spatial and time constrained acceleration. They provide a demonstration of this mechanism for both single and multi-tennant environments, where an API is provided in Python and C++. They use a cooperative scheduler approach to run acceleration tasks for the user by placing them into request queues. Requests are handled as independent events which allows them to executed in parallel, where time and spatial constraints allow. The instantiation of hardware accelerators is handled by FOS to encourage re-use of modules and reduce the need to reconfigure.

The ARTICo3 [9] framework uses a runtime library that leverages custom DMA and reconfiguration drivers (Linux) to abstract hardware management for the user, with a reduced API. The framework is built around contained kernels that possess their own allocated memory for moving data into and out of the accelerator. Communication between software application and hardware kernels is performed via shared memory buffers. Similar to other frameworks such as [10], they use a physical to virtual memory map driver to make hardware memory available to the user's application and thus allow for DMA requests to and from hardware. They argue that using uncached physical memory in the userspace penalizes execution performance and thus secondary memory buffers are also allocated using malloc, where the runtime copies data between them and the physical memory using standard `memcpy` calls.

R3TOS [64] is an RTOS that allows for on-chip resources to be used indistinguishably for computation or communication tasks using DPR. R3TOS does not rely on any static infrastructure besides its core circuitry; unconstrained parts of the device are kept free of obstacles, such as static routing, allowing the spare resources to be used as required. At runtime, the hardware tasks can be scheduled and allocated for improving computation density and circumventing damaged resources on the FPGA. Users are provided with a high-level API using mechanisms and services similar to a full OS, such as task relocation and data exchange, which abstract low-level operations and simplify the development of PR applications. A demonstration is provided for a Xilinx Virtex-4 FPGA, with the RTOS running on a soft MicroBlaze processor; a latter publication [65] implements R3TOS on a Zynq-7000 SoC.

CoPR [50] runtime operates with a two layered architecture, with a control plane implemented in software that applies hardware configurations based upon labels generated upon the system's specification. A configuration manager controls how and when physical reconfigurations are applied and can be called

from user software through an API. PR operations are abstracted and carried out automatically by the configuration manager. The API attempts to hide the low level complexity required to initiate PR. CoPR does not support Linux or the ZynqMP architecture.

Many of the mentioned runtimes propose PR management that enable multi-tenancy, accelerator scheduling as well as redundancy/sandboxing. Typically these runtimes expect the designer to have a good understanding of the underlying accelerator and how to apply the state of the hardware, either via API calls to shell or kernel interfaces. While some tools such as FOS do provide generic software abstractions with userspace drivers, this still requires the assumption that the designer is aware of how to move data into and out of the shell, rather than simply requesting a state for the hardware to be set to.

3.2.2.1 PR Controllers

In order to reconfigure Xilinx FPGAs, there are typically two main physical interfaces for writing a bitstream to the device, the ICAP and PCAP interfaces. Academic research has been conducted to extend the functionality of these interfaces either with abstraction or performance.

ZyCAP [11] was an early custom hardware controller and software API for managing the loading of bitstreams via the Zynq-7000's ICAP interface (the ZynqMP is unsupported). Their work showed that on the Zynq-7000 device, using a DMA controller in the PL to the ICAP to flash PR bitstreams, their PR controller had a 2.96× improvement on performance versus the PCAP, getting close to achieving the theoretical throughput of the bus at 382 MB/s (theoretical max. of 400 MB/s). Their implementation was slightly less resource efficient compared to the other vendor mechanisms and only supported a bare-metal software implementation rather than full operating system support. Figure. 3.6 shows how the ZyCAP controller is connected to the Zynq's PS. Work has been done to attempt to overclock the ICAP primitive [66], however this frequency depends on manufacturing variation and specific placement and routing.

RT-ICAP [67] is a lightweight PR controller aimed at real-time systems that require bounded and predictable reconfiguration times. Their controller supports two operating modes; scratchpad memory (SPM) stream and CPU stream. In their architecture, SPM is used to store the PR bitstreams and load them into the ICAP and additionally acts as a local general purpose memory for the processor. This is important as access time for an SPM is guaranteed to be a single clock cycle and thus deterministic in terms of bitstream transfer time. Bitstream compression is used to overcome limitations of the SPM, where the CPU stream mode is used for when the bitstream is too large for the SPM.

MiCAP [68] implements a PR controller that is designed for dynamic

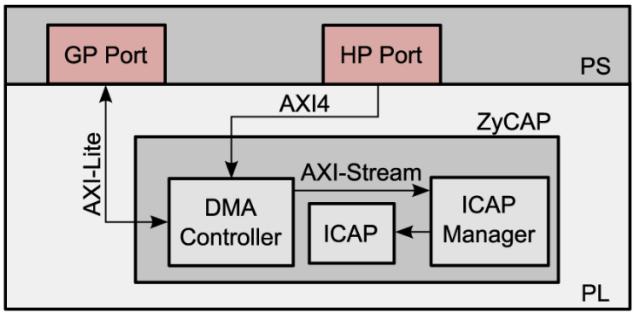


Figure 3.6: ZyCAP highlighting interfaces between PS and PL [11]

circuit specialisation (DCS), an FPGA implementation technique that serves parametric designs. A design is defined as parametrized when some of its inputs are infrequently changed compared to the rest, specified as constants. Under DCS, for each change in the parametrized values, a new circuit is generated and the FPGA is reconfigured at runtime using partial reconfiguration. The MiCAP controller allows for the current configuration to be read back from the PL, unlike controllers such as ZyCAP, where the ability to read-back is an important feature for enabling DCS. MiCAP was extended in [69] to supports an AXI-DMA engine, improving performance by a factor of 3 but increasing resource consumption by 4 times when compared against Xilinx's HWICAP with AXI-DMA engine.

DyRACT [70] manages to achieve a near theoretical throughput for the ICAP using a high-speed PCIe interface along with external memory, targetting FPGAs coupled as co-compute rather than tightly coupled. This is able to achieve 399.80 MB/s on a Virtex-7 FPGA, where the theoretical throughput of the ICAPE2 is 400 MB/s. Their implementation assumes that the PCIe interface is dedicated to reconfiguration and could begin to impact data transfer if other systems utilise the same interface.

The PR controllers mentioned were introduced as demonstrations rather than implementations for real world applications; where the control interfaces are limited and bespoke. Many current academic works reuse Xilinx's FPGA Manager driver and PCAP controller, utilising the readily available driver and documentation. As mentioned the FPGA Manager driver is both limited in performance (latency and throughput) as well as abstraction as well as forcing the configuration process to pause while the driver loads a new PR bitstream.

3.3 Applications of Partial Reconfiguration

Across both academic and commerical systems, PR has found applications in domains ranging from communication systems to autonomous vehicles. The advantages of time-multiplexed hardware are prevalent across power, efficiency

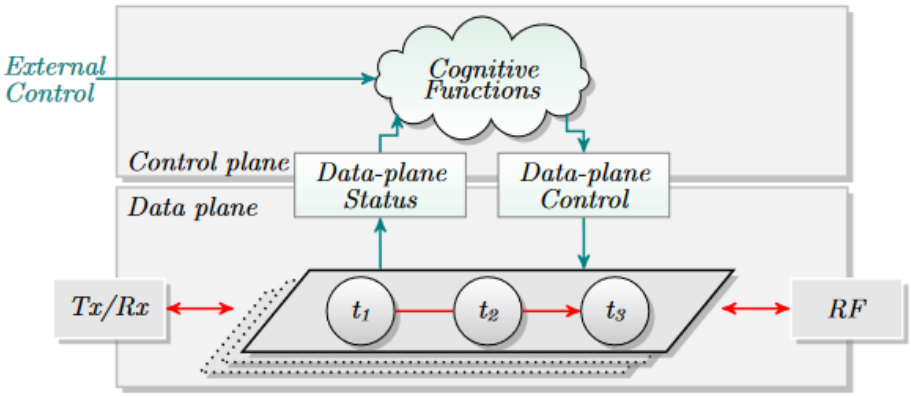


Figure 3.7: Concept of a cognitive radio with control and data planes split across a CPU and FPGA [12]

and cost factors of such applications.

3.3.1 Communications Systems

Wireless communication standards rapidly evolve adding new features and supporting new technologies. To adapt and support changing standards developments in software defined radio (SDR) have allowed for increased hardware flexibility. Given that SDR are typically required to handle complex signal processing applications, FPGAs are ideal candidates for compute. Combining this with a general purpose processing system as well as time-multiplexed hardware using DPR, creates a powerful platform for handling demands of modern RF systems. Demonstrated in [12] is a platform for enabling radio designers to build cognitive radios, SDR radios with the capacity to adapt channel conditions to best utilise the RF spectrum, using DPR on the Xilinx Zynq. Their cognitive functions exist within the software domain with the baseband sensing in the FPGA, as shown in Fig. 3.7.

A baseband processing module for 3G, LTE and WiFi standards is compared for area, power, memory and time overhead in [71], with and without DPR. They were able to show that time-multiplexing the baseband processing modules rather than supporting them all simultaneously resulted in an average power consumption of 57.11 mW compared to 171.47 mW. They showed that the time overhead for loading the PR bitstreams over the ICAP interface, only introduced a 1.46 ms delay for a 1.4 MB bitstream.

3.3.2 Networking

Given that FPGAs posses a large resource of logical elements that can be used to implement complex data paths, utilising them for in-network acceleration is a key use case. [72] demonstrates a PR-based architecture for ternary content-

addressable memory (TCAM) emulation. TCAMs are used in networking for complex matching patterns and lookups and are typically implemented as soft-modules within the FPGA. They show how PR enables efficient and effective application of TCAMs in an FPGA's resources to support the addition and subtraction of rules engines. Another example is [73], where the authors show how network function virtualization can be implemented for software network functions (NF) using PR on an Xilinx FPGA within an Intel server. Their implementation, Dynamic Hardware Library (DHL) implements the main architecture of the NF in software running on the CPU and offloads the computational intensive functions to the FPGA using PR, which is more cost efficient and improves flexibility. They provide a decoupled software-level API and hardware-level acceleration to abstract complexity.

3.3.3 Image Processing

FPGAs are well suited to image and video processing due to high performance IO and the capacity to effectively pipeline incoming images frames for operations. DPR is well suited to these types of applications as often the implementation of these algorithms can be resource expensive and consume large areas of smaller FPGAs. [74] demonstrate a number of edge detection algorithms implemented under DPR. They provide experiments demonstrating filtering scenarios where size, complexity and intensity of computation are varied and resource utilization and timing are evaluated. [75] provides an example of how a two-dimensional (2-D) Haar wavelet transform IP core can be implemented using DPR and they compare performance, area, power and maximum clock frequency achievable for both static and partial implementations. [76] highlights an interesting demonstration where the FPGA is connected directly to the VGA interface of a camera and uses PR to filter the image before outputting it to a monitor. This direct data stream from a camera peripheral is particularly important to acceleration applications as time critical systems may not be able to suffer the latency introduced when streaming data first through a processing system before it reaches acceleration in the hardware.

3.3.4 Machine Learning

Due to the benefits of parallelism in machine learning, flexible hardware and specifically the ability to time-multiplex hardware has made PR a viable implementation strategy. Machine learning is a branch of artificial intelligence where the use of statistical methods and algorithms to make predictions or classifications are used to imitate the way that humans learn, gradually improving their precision and accuracy as they are provided with data. In [77] they develop a support vector machine (SVM) and K-nearest neighbour (KNN)

multi-classifier architecture for FPGAs using DPR. Their implementation allows specific regions of the FPGA to work either as an SVM or KNN classifier for processing bioinformatic data, where applying different classification algorithms to the same dataset is desirable for decision making. They show that by using DPR in their design, they can achieve 8× reduction in reconfiguration time versus the same implementation using traditional reconfiguration as well as reducing the consumed hardware resources and physical space on the FPGA device, when compared against deploying both classifiers simultaneously. Similarly, [78] also uses DPR to build an SVM classification system for seizure detection, that exploits the advantages of DPR to show how energy spent in this implementation has a reduction of 64% compare to static reconfiguration. [79] provides a accelerator for convolution neural networks, which are widely used for image classification, that utilises DPR for increased efficiency, while citing that they were able to compose the architecture such that there was no loss in performance or classification accuracy. Their architecture uses a basic processing element, similar to a shell, that possesses three main interfaces; a data interface (AXI Stream), a memory interface (AXI) and a GPIO interface.

3.3.5 Automotive

The rapidly adapting automotive sector has also benefited from FPGA applications due to the increase in computational nodes as well as a need for more performant networking within vehicles. Modern vehicles must be able to control a range of non-essential systems such as climate control and electric mirrors as well as critical systems like drive-by-wire and adaptive braking systems. [80] provides an insight into how PR can be used within automotive systems such as electronic control units (ECUs) to improve computational capacity as well as reduce power consumption. In [81] the authors propose a novel ECU architecture specifically designed to enable CPS that improve security and dependability in the design while citing negligible impacts on performance, energy and resource overheads.

They implement their architecture onto a Xilinx Automotive Spartan-6 FPGA and suggest that they can achieve an increase in speed of 47.93× with 2.4× less energy than a competitive processor on a quad-core iMX6Q SABRE automotive control board. They also showed that their architecture was 2.13× more tolerant to faults than the baseline SABRE control board. They evaluate their ECU architecture for common in-vehicle networks such as CAN, CAN FD and FlexRay. In [82], the authors explore how redundancy can be implemented into safety-critical in-car systems using PR along with a custom bus controller to provide rapid recovery from faults. Their work provides a custom extension to an on-chip FlexRay controller that can leverage a high performance PR

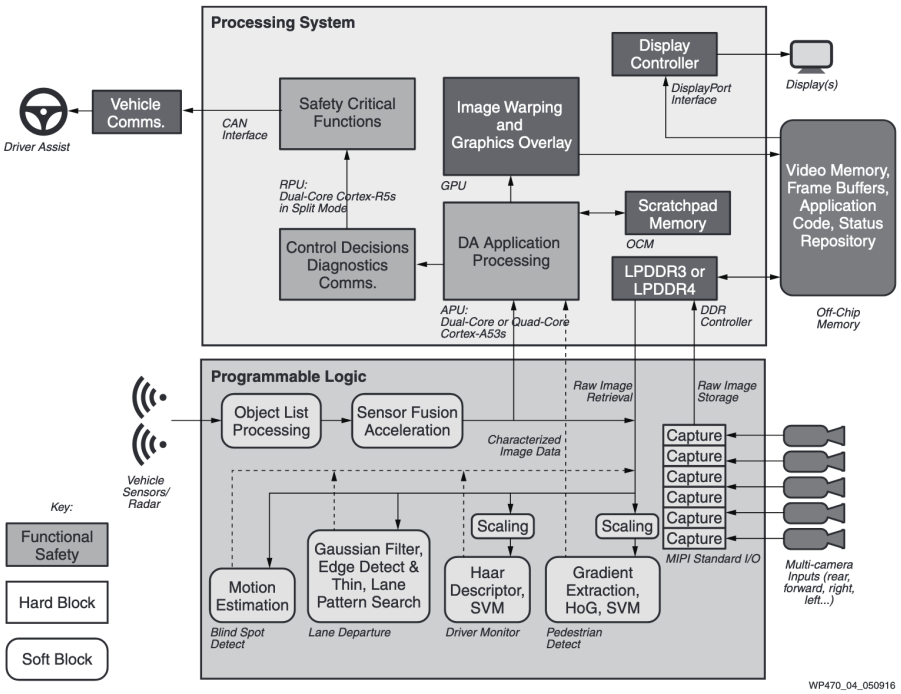


Figure 3.8: Xilinx ZynqMP Example ADAS Application [13]

controller for rapid reconfiguration.

Xilinx provides an example of an automotive adaptive driver assistance systems (ADAS) in their ZynqMP white paper [13], where extensive usage of camera systems in vehicles are used to provide additional safety features. Figure 3.8 provides an overview of how such an ADAS system might look, where hardware acceleration for events such as lane departure and blind spot detect can be performed on the incoming video stream from cameras and radars attached directly to the PL. The high performance bus interfaces between the PS and PL enables the processed characteristics data extracted from acceleration to be handed to the PS for appropriate decision algorithms to be performed on the data under software centric processing.

3.3.6 Space

PR has have proven popular for space applications where detection and recovery from errors are critical. This is due to an effect known as Single Event Upset (SEU) [83]. Volatile FPGAs based upon SRAM technologies such as Xilinx and Altera's device are particularly susceptible to the effects of SEU which can result in bits in the FPGAs configuration being incorrectly set and thus introducing unexpected behaviours and potentially even system failures. In [84], the authors developed a design flow, IPRDF, that allows the designer to build fully isolated and reconfigurable systems. One of their case studies

demonstrates how PR can be used to mitigate from SEUs as well as provide resilience against ageing and device imperfections. They also argue that the module insulation introduced by their framework increases the security aspects of systems designed by this tool, citing that no IP details, code or netlist are required to be available to the tool. [85] introduced a System-on-Chip Wire (SoCWire) architecture on a Virtex-4 FPGA. SoCWire is a network-on-chip protocol used by the space industry for flow control, detecting link errors, providing error recovery as well as other functions. For their design, the SoCWire routing structure was implemented into the static region of the design and the acceleration functions were implemented as PRMs, allowing the functions to be loaded/unloaded based upon the processing requirements at any time. [86] uses the R3TOS [64] runtime, a scheduler for allocating real-time PR hardware tasks onto FPGAs to circumvent the effects of SEUs and Total Ionizing Doses (TID). The effects of TID can result in permanent damage to a device and results in oxide breakdown and leakage current. R3TOS introduces fault-aware heuristics that are designed to reduce the number missed task deadlines by determining where to best place a hardware task. These heuristics include Empty Area/Volume Compaction heuristics (EAC, EVC and EVC2P) which address the state of the whole sandbox to assign a score for quality, in order to evaluate each potential placement. Their experiments demonstrate, that with the use of these heuristics, missed deadlines during hardware tasks are reduced and that the compute provided by the FPGA can be more effectively utilised. [65] expands on the original work of R3TOS to the Zynq platform utilising the SoC's configuration memory for exchanging data between the hardware partitions. As opposed to fixed partitions, R3TOS keeps the FPGA's reconfigurable area free of any partition boundaries and static routes. They demonstrate their tool by simulating different chip damage scenarios and show that their R3TOS-based prototypic avionics system can tolerate an average increase of around 13% more on-chip damaged resources than a traditional solution.

3.3.7 Autonomous Adaptive Systems

Crucially there are a number of components required to realise autonomous adaptive systems: high performance hardware to accelerate realtime and latency sensitive data acquisition, high level abstract software to interface with higher level operating system functionality as well network capable transports for communicating with other systems as well as a host platform.

The authors of [87] present a DPR system for autonomous driving systems (ADS) that is able to provide real-time vehicle and pedestrian detection. Their approach use a deep learning methodology for detection in poor/dark lighting

conditions. They utilise a custom PR controller that runs on the Zynq for managing the vehicle detection block logic, reducing the resource requirements via time-multiplexing to enable the existence of other ADS functionalities. This custom PR controller uses a similar approach to [11] but instead utilises a dedicated DDR module in the PL to store PR bitstreams. They demonstrate their system's performance capabilities by detecting pedestrians and vehicles in varying lighting conditions with a framerate of 50 frames per second at a resolution of 1080x1920 (where the system is clocked at 125 MHz).

[88] is a commercial documentation and demonstration of a real world autonomous adaptive system that utilises partial reconfiguration on the ZynqMP SoC for hardware acceleration. The authors explain that the high speed camera interface (MIPI) is directly attached to the FPGA to reduce CPU data transfer bottlenecks and to reduce energy consumption. They explain that due to latency dictating user-experience and safety, their design offloads localization tasks to the FPGA while performing perception tasks on the GPU. They cite that offloading the localization task to the FPGA reduces the scene understanding event from 120 ms to 77 ms compared to when performed on a Nvidia GTX 1060 GPU.

3.4 Summary

PR has seen many developments in recent years, with new device architectures from major vendors introducing new challenges. The process for designing PR applications is still complex, requiring bespoke knowledge across many domains. Many existing tools, both academic and commerical, require highly specialised knowledge of the tools themselves before applications can be built, tested or deployed. Vendors tools are historically proprietary and provide limited support for academic work looking to extend functionality and thus limit adoption for tools that must continue to evolve to support unannounced changes from the vendors. Many tools only support specific versions of vendor software and FPGA architecture, often rendering academic/open tools useless upon new releases of vendor software.

PR has been a key design aspect in a wide range of applications, as discussed in this Chapter, but is often implemented in a bespoke and custom manner to the target application. A more systematic methodology, with a focus on minimizing overhead and increasing design abstraction, would allow for a wider range of applications and their developers to leverage the benefits of PR. A low design impact tool that does not require learning of a new and custom framework or enforcing the user to design around custom hardware shells would have the benefit of making PR applications both more portable as well as more future proof.

While some academic tools aim to segment the workload of designing and developing PR applications, we believe that this is still a significant limitation for independent designers wishing to accelerate their software applications with PR. We demonstrate a justification for open source tools that are built around vendor software that can support version changes while providing a high level of automation to the design process such that non-expert and independent designers may develop applications with them. We have developed a selection of tools and abstractions that enable the design and development of PR applications, access to high performance reconfiguration both locally and remotely (over the network) as well as extend the abstraction of software to support modern cyber physical systems and enable the design of autonomous adaptive systems.

These tools are organised to form an end-to-end workflow for designers to implement and integrate PR into their applications. This includes tools to abstract constructions of PR based projects, automatic generation of PR modules and regions, integration of these designs into a Linux image builder, management of PR modes and configurations with a high performance runtime as well as abstraction of the operation of this runtime into a autonomous systems centric workflow. Additionally, a number of utilities to improve the design process were also added; including extensions to open source IP management libraries and project builders. These works will be further discussed in the following Chapters.

Chapter 4

Over the Network FPGA Reconfiguration

4.1 Introduction

Cloud computing provides a streamlined architecture for handling and accelerating large volumes of data from distributed sources. The centralised compute model in dedicated data centres allows for increased performance, scalability and provides access to a wider pool of computational resources including accelerators such as FPGAs and GPUs.

Although cloud-based approaches provide benefits when information (and computation) can be aggregated at a central location, it also introduces notable overheads in latency, bandwidth and resource requirements with respect to the volume of data being processed [89]. Edge computing through decentralised or distributed accelerators, that push compute from the data center up into the network and beyond, aim to address such challenges [90].

Distributed accelerators, owing to their proximity relative to the data source, are typically spread across the edge of the network's data sources and sinks. In addition to processor-based accelerators, non-traditional architectures such as reconfigurable heterogeneous SoC platforms like Xilinx Zynq and Altera Arria can be deployed to provide high throughput and parallel processing compute to edge applications. This interest in reconfigurable SoCs stems from the tight architectural coupling of a capable ARM-based PS for managing the tasks and flexible PL that can support custom accelerator kernels. This allows deployment of operating systems such as Linux to manage networking and peripherals, while processing can be offloaded to the programmable logic for acceleration when required. Applications mapped to such SoC architecture typically utilise the peripherals of the PS such as network and control interfaces and rely on software to manage the interaction between them.

Software managing the networking interface can lead to issues with control

prioritisation, due to the nature of task driven processing seen within typical operating system schedulers. This procedural handling of tasks can lead to non-deterministic latency in the time taken to process incoming network packets, such as those containing state control information for the accelerator. Higher priority tasks may override any control flow events such as decoding networking packets and potentially lead the accelerator to delay handling/miss critical tasks such reconfiguration of the compute hardware. An example of task prioritisation in edge acceleration can be seen in research into automotive ECU systems where system failures can be mitigated using FPGA based backup circuits [91].

This effect of latency on the network stack can be mitigated by first directing control packets into the accelerator, which analyses and performs decision logic as the packets as arrive at the interface. This technique, employed in many smart network interface cards (Smart NICs), can be extended to edge devices based on heterogeneous SoCs to address such challenges. However, building a low-latency accelerator system around such an architecture with native offload capability of regular communication as well as reconfiguration requires considerable FPGA expertise.

In this Chapter we introduce a smart NIC approach to initiating PR over the Ethernet PHY on the Xilinx Zynq-7000 SoC platform. We compare performance between traditional methods of initiating PR to a method of bypassing the PS to load the control packets for reconfiguration directly. Thereby bypassing the Zynq's PS Ethernet driver and scheduler, reducing latency non-determinism. We explore using alternative methods to decoding incoming Ethernet frames, moving the packets directly into the PL to sniff the frame header for reconfiguration commands.

The work presented in this Chapter has also been discussed in:

- Alex R Bucknall, Shanker Shreejith, and Suhaib A Fahmy. Network enabled partial reconfiguration for distributed FPGA edge acceleration. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 259–262. IEEE, 2019 [92]

4.2 Contributions

The key contributions of this Chapter can be summarised as:

- A flexible packet sniffing Ethernet framework for dynamically routing & processing packets in PL
- Custom architecture for provisioning PR bitstreams without PS intervention over the network interface with arbitration

- Demonstrated performance improvements and latency reduction compared to vendor tools for locally-cached bitstreams

4.3 Related Work

Traditionally, FPGA-based accelerators are coupled with powerful centralised compute, such as in a cloud data centre. They can provide massive parallel process offloading and/or deep pipelining for compute bound software tasks, as required in applications such as artificial intelligence, data analysis and other traditional high performance applications. An advantage that FPGA-based acceleration provides is the ability to dynamically reconfigure the hardware circuitry for the task at hand, unlike similarly style ASIC-based acceleration units.

PR has several advantages over standard reconfiguration, including use of the static regions whilst reconfiguring and the ability to retain logical state whilst under reconfiguration. PR can also provide faster reconfiguration time intervals than standard reconfiguration as partial bitstreams are often significantly smaller than complete static bitstreams.

Historically, FPGAs would require an external controller to manage reconfiguration, which was typically via serial protocols such as JTAG or SWD; as can be seen Xilinx's earliest Virtex Flagship FPGA, circa. 1999 [93]. This approach was slow and cumbersome; requiring each attached device to be individually addressed (over serial) via the processing system and was restricted by the throughput limitations of the low speed programming protocols. As FPGA offloading within data centres increased in popularity, PCIe became the standard interface for FPGAs as it provided both high throughput and low latency to address individual platforms. Additionally this interface could be both used for reconfiguration as well as a high performance streaming between the client and the host. The Microsoft SIRC platform presented a software-hardware API for managing communication and synchronisation for reconfigurable compute attached over PCIe [94]. Although this makes it easier to control attached FPGAs, it still requires a host/master entity (such as a server in a data centre) to manage and address individual FPGAs. This architecture, forces any accelerator applications to first pass through the central entity, often a server in a data centre providing software independence, which then forwarded the tasks onto the attached FPGA/FPGAs. Approaches like DyRACT [70] extended this approach by utilising custom PR controllers and software stack to deliver PR bitstreams at high-speed over PCIe. However, for time critical applications the software stack adds additional layers of processing before the task reaches the accelerator on top of the existing delay from network propagation. Furthermore, PCIe protocol is composed of signalling pairs known

as lanes, which are highly sensitive to interference and do not support long range communication, restricting the protocol to tightly clustered systems. As these systems became more independent, such with Xilinx's Hybrid SoC Zynq platform, the demand to shift the task of reconfiguration onto the FPGA itself increased, requiring the FPGA to manage it's own reconfiguration flow.

Efforts to improve the vendor tools; ZyCAP, DyRACT (for PCIe), there are a few other high-speed reconfiguration controllers, but all of them are standalone and integrating them with a network stack would require the design of appropriate pipelines.

Vendors have developed internal reconfiguration managers to support FPGA-driven reconfiguration. Xilinx initially allowed for internal reconfiguration using its ICAP interface and later provided a software centric mechanism, the PCAP. While PCAP uses a software interface and synchronous control flow to handle reconfiguration, preventing the processor from executing other compute tasks while reconfiguration, a standalone ICAP also requires users to design infrastructure around it to enable reconfiguration. Custom controllers such as the RT_ICAP [67] provide the advantage of a real time PR manager but do not provide software support for hybrid FPGA platforms, restricting implementation of networking layers within the PL and requiring the user to have a deep understanding of the network controller implementation. Alternatively, buffering bitstream onto internal BRAMs offers a high performance interface for streaming into the ICAP, while also being processor independent, though the approach is limited to small PR bitstreams [95]. DyRACT [70] is PCIe based platform that uses a custom PR controller and software stack designed to integrate with high speed channels for streaming partial bitstreams. However, designing a network trigger that can utilise DyRACT provides challenges as integration with the system's networking stack would require a complex understanding of data pipelines. Despite the developments in custom PR tooling, deterministically triggering PR from within network infrastructure is still non-trivial, especially on reconfigurable SoC systems that rely on PS and software for network/packet management.

The problem of deterministically triggering reconfiguration from an external network interface persists. Moving reconfiguration control from the PS to the PL, enables the PS to both continue to operate while simultaneously allowing for a control packet to be processed during the receive transaction. This is flow is faster than traditionally waiting for a complete frame to arrive before beginning to process it in software. This allow for lower latency in arriving control packets and for higher throughput with respect to total time for partial reconfiguration.

Reconfigurable SoC platforms are highly suited for IoT streaming/edge accelerators that improve on centralised accelerators by offering improvements

in energy efficiency, throughput latency and resource cost, as showcased in [90] for the case of accelerating convolutional neural networks. PR in IoT-type accelerator applications has a range of potential benefits to data streaming; placing the acceleration logic in-network and closer to the data source results in a decrease in latency and a tighter feedback loop for control data. It also provides advantages for scaling and computation allocation; if a data source requires additional acceleration units, PR allows nearby acceleration units to reconfigure for the demand. [96] highlights the advantages of virtualization in FPGA based acceleration platforms and demonstrates how such allocation of compute can outperform traditional use of virtual machines. The work explores full reconfiguration and we believe PR can extend this advantage further, especially on reconfigurable SoC platforms.

The approach presented in this Chapter enables network triggered partial reconfiguration, utilising a strategy for remapping the network processing path from the PS into the PL. This strategy enables inline decoding of control ethernet frames in order to extract reconfiguration commands with minimal latency and with deterministic timings compared to traditional networking approaches.

4.4 System Architecture

This Section presents a comparison between the traditional architecture for managing a networking stack and the architecture proposed in this Chapter. This provides the context and comparisons required to understand the motivations for this work.

4.4.1 Traditional Approach

Off-the-shelf boards based on commercial Hybrid SoC platforms such as the Xilinx Zynq devices attach the Ethernet interface/PHY directly to the PS-EMAC cores, allowing operating systems such as Linux to implement complete TCP/IP networking stacks when loaded on to the ARM Cores in the PS. In order to communicate with the external Ethernet controller, a driver running on the PS must be initialised during system set up; where the driver will allocate a continuous range of n 64-bit locations within memory for both the Receive Buffer Queue and the Transmit Buffer Queue. Additionally, n corresponding locations are initialised for the Receive and Transmit Buffer locations, with their addresses (known as buffer descriptors) and status are stored in the Buffer Queues. The Ethernet driver will configure the start addresses of the buffer queues into the base address registers of the Ethernet controller.

Upon receiving an Ethernet packet at the Ethernet interface, it is tempor-

arily buffered until the frame is completely received and verified to be error free. The controller will then look up the memory location for the receive buffer descriptors, perform a DMA transaction to move the frame into memory and clear the buffer queue status from empty. The Ethernet driver on the PS will then alert the software stack or application that a frame has arrived either via internal interrupt or by polling the buffer status; at this point the raw frame can then be decoded by software. In order to perform a transmit transaction, the flow is reversed and the PS initialises a buffer location then clears the status flag, at which point the Ethernet DMA notices the flag change. The Ethernet DMA then copies the frame to its internal buffer and clears the status flag. Using the Ethernet PS driver requires that the processor is involved in all receive and transmit transactions, restricting its operation and introducing non-deterministic latency when handling RX/TX events under high computational loads.

In a traditional accelerator, an arrived packet event would then trigger the PS to begin the decoding process and extract the bitstream information required to trigger PR. Upon extracting the PR mode information, the PS could then invoke PR via the PCAP driver interface.

4.4.2 DMA Proxying

In order to initiate a reconfiguration operation from a network packet, the PS must decode and extract the relevant data from an incoming frame. This may result in the reconfiguration taking place under non-deterministic latency if the processor is busy handling higher priority tasks. An alternative to this flow control would be to offload reception/transmission into the PL and to perform decoding/encoding in hardware. The architecture presented in this Chapter utilises a method of indirectly forwarding packets into the PL known as DMA proxying [97]. DMA proxying remaps buffer locations stored within DRAM memory addresses, to alternative locations such as within the PL. This allows a DMA transaction moving data directly in/out of the remapped location, often a hardened Block RAM (BRAM) or FIFO at the destination, via the AXI GP port. To move Ethernet packets back into DRAM for processing, a DMA controller for the Ethernet may be instantiated within the PL to copy the frame via DMA transaction into DRAM over the Zynq's HP port. While this strategy reduces the latency required to receive the complete frame into the device, it still relies on the PS for unpacking, processing and decision making. It is possible to implement a complete network protocol stack into the PL and perform the functionality of the Ethernet PS Driver within hardware, however this both increases the complexity of the application and requires the designer to have a complete awareness of how to manage such a controller

within the FPGA fabric. The major drawback of implementing the entire networking stack into the FPGA is that even using vendor provided IP cores are logically resource intensive, leaving limited space for the user's applications. Larger FPGAs with increased resources do make this more viable but with the trade off of increased cost and power consumption. We propose a strategy of extending the DMA proxying process to include smart controller behaviours that extend the data path rather than completely replace the functionality of the PS Ethernet stack.

4.4.3 Network Partial Reconfiguration

With the Network Partial Reconfiguration architecture, received Ethernet frames are proxied into the PL (using DMA proxying) buffer memory within the Ethernet bridge, as shown in figure 4.1. The Ethernet bridge implements the packet handling and decoding logic that is usually done by a software task in the PS. The bridge is initialised from software upon system start-up; post initialisation, the bridge can monitor, analyse and redirect incoming packets based on the PS configuration (based on frame's header content and/or data segments), reducing the latency and non-determinism associated with packet reception and decoding within the software stack. The entire architecture is designed to support multiple accelerator slots, allowing the designer to configure the modes of operation via high-level parameters in the application design process.

As mentioned, the PS configures the Ethernet bridge upon system initialisation by writing into the register stack within the bridge itself. The register stack specifies the fields to look for, the match parameters for frame headers, specific data patterns to be observed and offset for each of these parameters, allowing a deep packet inspection to be performed on the received frame. The register stack is a memory mapped bank of registers (32-bit wide) in the PL, interfaced to the PS via the AXI GP port; locations within the register stack can be configured to match OSI layer 2/3 addresses, specific patterns to be decoded within the data segment, byte offset which marks the start of such patterns within a data frame, packet type and others. Based on the match, the bridge configures the egress path for the packet by configuring the multiplexer within the receive side arbiter (RX Arbiter) logic. The packet can be directed to one of the accelerator slots configured on the device, the PS for software processing, the PR Controller to load the received bitstream over Ethernet or be ignored by dropping it from buffer. The incoming packet is buffered into the ping-pong RX FIFO within the bridge, allowing packets to be received and processed in parallel. While the packet gets buffered into the RX FIFO, the inline sniffing logic within the data path extracts information from packet

headers (and data segments) to compare against the configured values within the register stack. The inline logic incorporates a set of shift-registers and state machines to keep track of the different segments that are extracted from the packet. The straightforward match would be a packet header match (OSI Layer 2/3 match), that would redirect the packet into one of the accelerator slots or back into the PS DRAM using the integrated PL DMA logic. The entire packet is moved to the address/offset loaded into the register stack by the PS, mimicking the ring-buffer DMA configuration for a regular PS-Ethernet system.

When a reconfiguration request is decoded by matching the header as well as a preconfigured number of bytes within the data segment at a specified offset, the inline logic further extracts the bitstream name from the subsequent bytes and raises an interrupt to the PS. The PS reads the bitstream name from the bridge register and spawns a reconfiguration task that uses the custom drivers of the PR controller to decode the bitstream name, locate the bitstream in the memory subsystem (DRAM, Non-volatile SD card) and initialises a DMA transfer into the PR controller in the PL logic. The driver organises the bitstream information into a linked-list, allowing frequently used mode names to be cached into the DRAM for faster reconfiguration speeds. The driver releases software control back to PS as soon as the DMA transfer is set up, enabling the reconfiguration task to be completed while PS executes the regular management/processing tasks of the application.

While reconfiguration from a local database would be appropriate for commonly deployed accelerator kernels, specialised kernels may be instantiated into a reconfigurable region, where the module has to be fetched over an external link (i.e. PCIe or Ethernet). Our framework supports this through a remote-reconfiguration request, which accepts an incoming bitstream over the network to be loaded on to an accelerator slot. A remote-reconfiguration request is decoded by matching the frame header and the remote reconfiguration keyword within the payload section; on the first instance of this command, the bridge records the information about the request, like the number of packets over which the bitstream is to be received and the bitstream size from payload. The bridge also configures the receiver side arbiter to forward the packet to the PR controller, while also setting up the ICAP manager to receive packets from the arbiter instead of the AXI DMA. At each subsequent remote packet, the bridge monitors and updates the frame and byte count to ensure that it stays consistent with the information setup by the initial remote-reconfiguration command. The payload section is directed to the ICAP manager as before, until the last byte of the bitstream is received. The bridge then interrupts the PS to signal that a new bitstream has been successfully loaded from the network into the slot or to indicate that an error has occurred (missed frame

or bit-error); the PS would initiate an acknowledge or no-acknowledge to the remote-reconfiguration request to complete the cycle.

In case of other incoming data packets, the layer 2/3 header information is used to determine the destination of the packet within the system. With a positive match, the bridge issues a control packet to the RX arbiter to configure the destination to the matching accelerator slot. The bridge may also be configured to direct the packet to multiple accelerator slots (by appropriate configuration of the register stack) and the arbiter sets up multiple write paths in response. The interface uses AXI-streaming protocol and a logical AND of control signals in case of multiple paths to ensure that both slots are ready to accept data simultaneously. The arbiter follows a strict FIFO system to ensure that incoming packets are correctly ordered. To ensure that an accelerator cannot stall the pipeline, a time-out mechanism is put in place which causes a packet destined for a slot to be dropped if the destination is unable to accept it. An interrupt is raised in this case and the software stack should issue a retransmission request to the source, using the packet header/data frame information that can be read from the register stack.

In the case of no-match, the arbiter issues back-to-back reads to the FIFO with no write destination setup on the multiplexer to clear the packet FIFO. Alternatively, such packets could be set to redirected to the PS by setting the no-match bit in the control register of the register stack, allowing the software to make informed decisions.

On the transmit side, the TX arbiter manages output data generated by the accelerators, allowing this to be packed into an Ethernet frame and sent out onto the network. The TX buffer uses the slot information to reconstruct the frame header (based on configurations within the register stack) and if space is available, reads the data from the requesting slot to complete the frame. A completed frame is moved into the TX FIFOs, following which a DMA request is issued to the PS Ethernet DMA to read the packet from the proxied locations within the PL.

4.5 Case Study

To evaluate our network enabled partial reconfiguration architecture we assess the case for accelerating cryptographic computations on a Xilinx Pynq-Z1 device, mimicking a network attached accelerator. We implement the application by utilising multiple cryptographic IP blocks on the Zynq XC7020 on the Pynq-Z1 device. The IP blocks are built as partial bitstreams PR regions (slots) on the design, large enough to host the core. We explore both cases of locally cached bitstreams that can be triggered from the local storage (SD Card, DRAM) as well as receiving the bitstream over the network. We

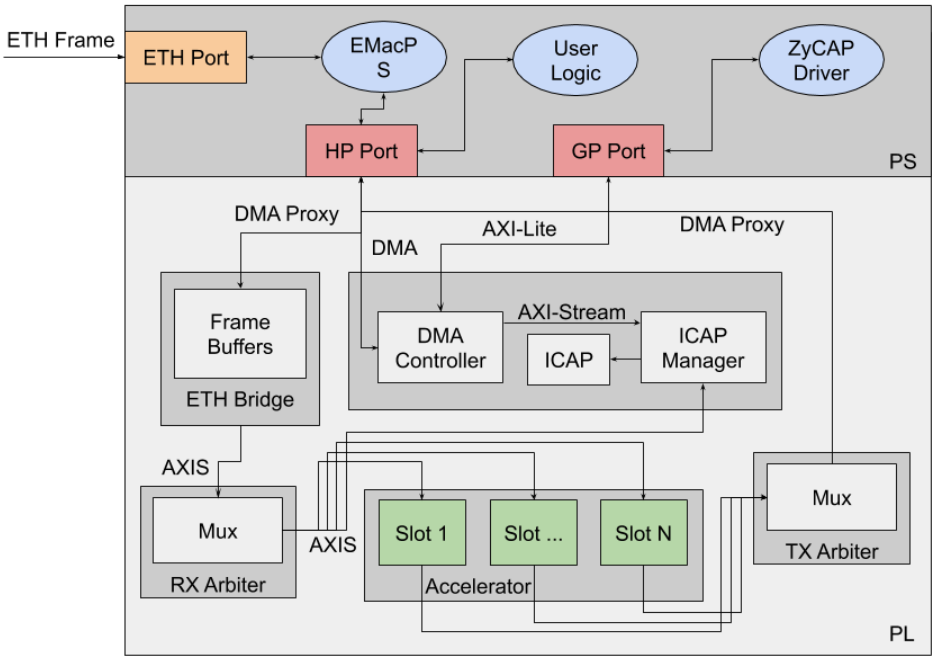


Figure 4.1: Network Enabled PR Architecture.

also evaluate an approach against the vendor provided PCAP framework by receiving the packet within the PS (for software decoding and software driven partial reconfiguration) and PL (for hardware decoding and software driven partial reconfiguration).

The case study implements a lightweight PRESENT cipher [98] and the more mainstream AES-256 cryptographic core. The two paradigms allow exploring a security infrastructure for a lightweight sensor network system as well as more mainstream compute accelerators. This allows us to demonstrate how edge nodes could use the PL offload to secure packets/information up to and/or down from a centralised server and adapt the cryptographic scheme at runtime. This scenario can be visualised as the case of a lightweight sensor array that does not incorporate the compute capability or power budget to implement such security primitives locally and thus offload computations to the edge device in both directions [99]. On the other hand, the same edge node could serve as an inline network switch that offers in-network cryptographic processing for the switch interface, aiding in reducing the routing delay within the switch itself. The edge device handles the incoming stream of data in both directions, and switches the encryption scheme based on the device/network that it is currently servicing.

Table 4.1: Resource Utilization on Zynq-7020

IP	LUTs	FFs	DSPs	BRAMs
AES-256	3204	2990	0	0
PRESENT	150	149	0	0

4.5.1 Advanced Encryption Standard

Advanced Encryption Standard is an established standard for symmetric key block ciphers and is used in many secure systems. AES allows both encryption and decryption of 128-bit blocks of data using a symmetric key of varying length depending on the chosen standard, e.g. AES-256 uses a 256-bit key. In order to encrypt, plaintext data is processed through a series of layers including key expansion, key rounding, substitution, shifting, etc. and then repeated a number of times on the output block of data. The size of the key used determines the number of repetitions required for encryption/decryption; AES-256 uses 13 repetitions. In order to decrypt the data, the receiving party must also possess the same secret key used in encryption as this is used in the reverse of the encryption process to decode the plaintext data. The AES-256 core was implemented based upon an open source BSD-2-Clause-licensed Verilog core [100].

4.5.2 PRESENT

The PRESENT cryptographic protocol is designed for compact data payloads such as those that might come from an IoT device or sensor network. PRESENT was included in this case study as it is designed specifically for constrained devices and provides a lightweight alternative to AES whilst competitive requirements for security [98]. Typically lightweight devices such as IoT systems can be categorized as low throughput, sending packets of data in short bursts to minimise the time required for the device to be active/consuming power whilst transmitting data. This is relevant to a range of systems and protocols including low power microcontrollers utilising lightweight RF protocols such as RFID, ZigBee, LoRaWAN among others [101]. Due to its small block size and key lengths 64 bit and 80/128 bit respectively, PRESENT is better suited as a lightweight block cipher for the described scenarios than other schemes such as AES. Within the application context, PRESENT is employed as the edge interface to the sensor network, allowing data to be securely exchanged between sensor elements from the array and in-network data aggregator/processing subsystems down the network for sensor fusion or analysis. The Pynq-Z1 was chosen as a representative node which offers limited resources and lower energy consumption, emulating a real-world IoT system that can adapt its compute

capability to serve the task by loading the requested compute kernel (bitstream) into the accelerator slot using partial reconfiguration. Table 4.1 highlights resource utilization in the individual cryptographic cores.

4.6 Experiments

To evaluate PR over the network, we simulate delivering Ethernet frames that request a specific cryptographic scheme for a subsequent stream of data by loading them into the available slot. Network-enabled reconfiguration is critically important in this case as it allows to deterministically set up the accelerator slots in PL, bypassing the PS, which depending on the software application, might delay the processing of the data packets, i.e. OS scheduler. For real time security applications, the time to trigger reconfiguration of the encryption/decryption cores is essential for preservation of event timings and optimal network response. A single accelerator slot is considered in the design to simplify the measurement process, leading to a partial bitstream size of 799,584 bytes for both the cryptographic cores.

An Ethernet frame was crafted composing of payload segments to denote if a packet is a reconfiguration command, an initial remote-reconfiguration command, subsequent remote-reconfiguration commands with segments of PR bitstream or a regular payload containing sensor data that needs to be encrypted. We compare the results between a PS driven reconfiguration command using Xilinx's PCAP PR flow, a PS driven reconfiguration command flow using a high-speed PR controller in PL and with the custom network PR reconfiguration manager which implements reconfiguration command extraction within the PL. We use the same Ethernet frames across all the possible combinations. In the case of a reconfiguration command frame, the frame has a specific layer-2 format and a data segment that specifies that the frame is a reconfiguration command and the name of the cached bitstream to be loaded. In case of a remote-reconfiguration request, the data header specifies the remote-request command, size of the bitstream and a counter specifying the sequence number (set to 1 for initial and increments for every subsequent frame). For all experiments, the bitstreams were cached into the DRAM memory of the PS, removing any dependency caused by loading the bitstream from the slower non-volatile storage sources. Results for the experiments can be seen in Table 4.2.

4.6.1 Frame Decoding in PS (PCAP)

This case presents the typical development scenario when utilising the vendor tool flow and resources to integrate dynamic reconfiguration into the Zynq system. The Ethernet frame containing the reconfiguration command was

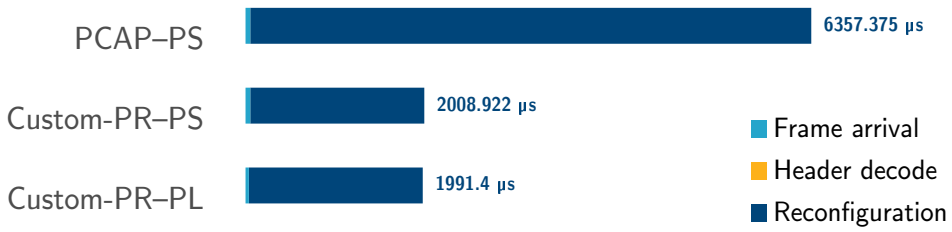


Figure 4.2: Zynq Processing Ethernet Packets in PL.

Table 4.2: Network PR Experiment Results

Experiment	RX (ns)	Decode (ns)	Reconfig (ns)	Reconfig Throughput (MByte/s)	Total Throughput (MByte/s)
Xilinx PCAP (PS Decode)	53238	297	6303840	123	122
Custom PR (PS Decode)	53246	294	1955382	398	368
Custom PR (PL Decode)	37605	N/A	1953795	399	391

generated within the software application and the Ethernet PHY set to loopback mode to receive the frame back into the device. PCAP uses Xilinx's *dev_cfg* drivers to handle bitstream movement from the PS; as mentioned, the partial bitstreams were buffered into the DRAM. The PS Ethernet RX buffers were configured to use DMA to transfer the received frames into specific locations in the DRAM and interrupt the PS when a frame had been successfully received (i.e., available for PS to read at the DRAM location). The interrupt handler decodes the frame and triggers a software task, if the frame is decoded as a reconfiguration request. The software task then decodes the requested mode name, looks up its location in the DRAM and initialises the reconfiguration via PCAP. We observed that the total time taken for the task to complete from the receive frame interrupt was 6.357 ms with 53.54 µs for the time to trigger reconfiguration and 6.304 ms for the actual reconfiguration. This is inline with the performance expected from PCAP, staying within the documented throughput of 145 MB/s [102].

4.6.2 Frame Decoding in PS (Integrated Controller in PL)

High-speed reconfiguration controllers may be implemented within the PL to improve the overall reconfiguration performance of the vendor tools. To evaluate this case, we integrate a custom reconfiguration framework which uses a hardware reconfiguration manager and software library to manage bitstream organisation. In this experiment, the software tasks are still required to decode the frame however once the mode name is known, control is passed to the reconfiguration framework. The UML diagram in figure 4.3 captures the sequence of events from the arrival of the reconfiguration packet at the Ethernet interface (PS-EMAC) to the completion of reconfiguration through the hardware manager. The PS is then free from managing the bitstream

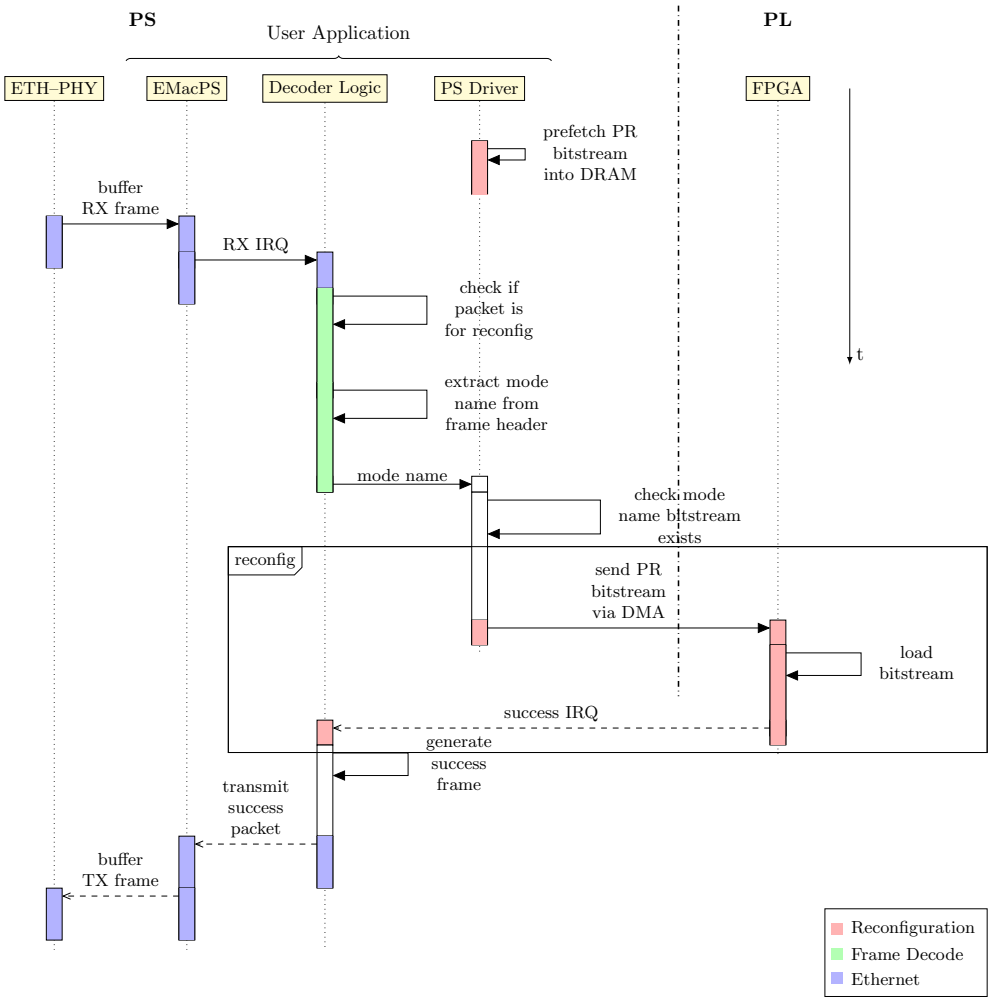


Figure 4.3: Sequence of events when Ethernet frames are handled by PS and reconfiguration is managed by an integrated PR controller in PL.

transfer and can return to processing tasks; the software core of the reconfiguration framework marks the completion of the task by raising an interrupt, allowing the PS to synchronise the system. We observed that the total time consumed for reconfiguration from the reception of frame within PS to completion of the reconfiguration task reduces to 2.01 ms. This is largely down to the improved reconfiguration speed offered by the hardware reconfiguration manager, completing the reconfiguration in 1.955 ms with an overall throughput of 368 MB/s, in line with the measurements made for custom controllers such as ZyCAP [11]. More importantly, the software framework releases the PS once the reconfiguration is set up; however, the time to trigger reconfiguration is limited at 53.54 µs, as shown before.

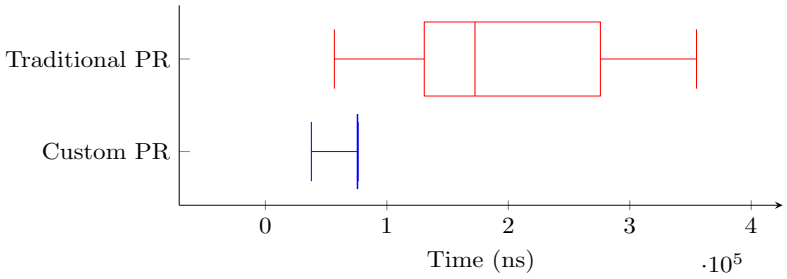


Figure 4.4: Variation in partial reconfiguration triggered over the network interface

4.6.3 Frame Decoding in PL (Custom PR Controller)

To reduce the overheads associated with software frame decoding, we utilise the custom network enabled PR architecture described in this Chapter to offload the packet decoding to the PL. As with the previous experiments, described earlier in this Chapter, the reconfiguration frame was generated within the PS and looped back at the PHY; however, this time using DMA proxying, the frame gets redirected into the PL, decoded inline by the logic and triggers PR, in the case of a reconfiguration command. This allows for complete bypassing of the PS for the trigger packet handling or deconstruction, removing the need for a software task to decode the frame. The UML diagram in figure 4.5 captures the sequence of events from the arrival of the reconfiguration packet at the Ethernet interface (PS-EMAC). Decoding the packet within the hardware allows the PR controller driver to receive a PL interrupt, read the requested mode and initiate a PR request via DMA; once again freeing the PS as in the case of the previous experiment. This reduces the time to trigger reconfiguration to 37.61 µs, performing the complete reconfiguration process in 1.992 ms. Note that the frame decoding within the PL does not incur additional latency as the frame's content is analysed inline within the data path as the frame arrives into the PL.

The benefit of offloading packet decoding to the Ethernet bridge in PL can be observed when the PS executes the decoding task while also handling non-pre-emptive critical tasks. To show the impact, we emulate the case where PS is loaded with a compute event of higher priority that gets executed in a periodic fashion (like a system management task in a server), while the packet decoding task is enabled by an interrupt from the PS-EMAC. The task reads a sequence of registers and has a variable execution time depending on the contents of the register, while the packet decoding task decodes the content of the frame (checks for reconfiguration command), performs a look-up for the requested mode and initiates a PR operation. We observe the variation in processing latency across 10000 tasks trigger in this setup, performing this test 25 times. The test was then repeated (varying the number of tasks) with packet

decoding offloaded to PL, where the PS task is reduced to the mode look-up and initiation of PR. Figure 4.4 plots the observed deviation in handling the reconfiguration command when the reconfiguration command is decoded in PS against the hardware decoding. The results show that the network PR flow offers much better predictability compared to the standard approach of decoding and managing reconfiguration within the PS, even when high-speed reconfiguration management IPs are used. The variation between the upper and lower quartiles of the network PR is only 1μs compared to the 178μs seen in the standard vendor flow.

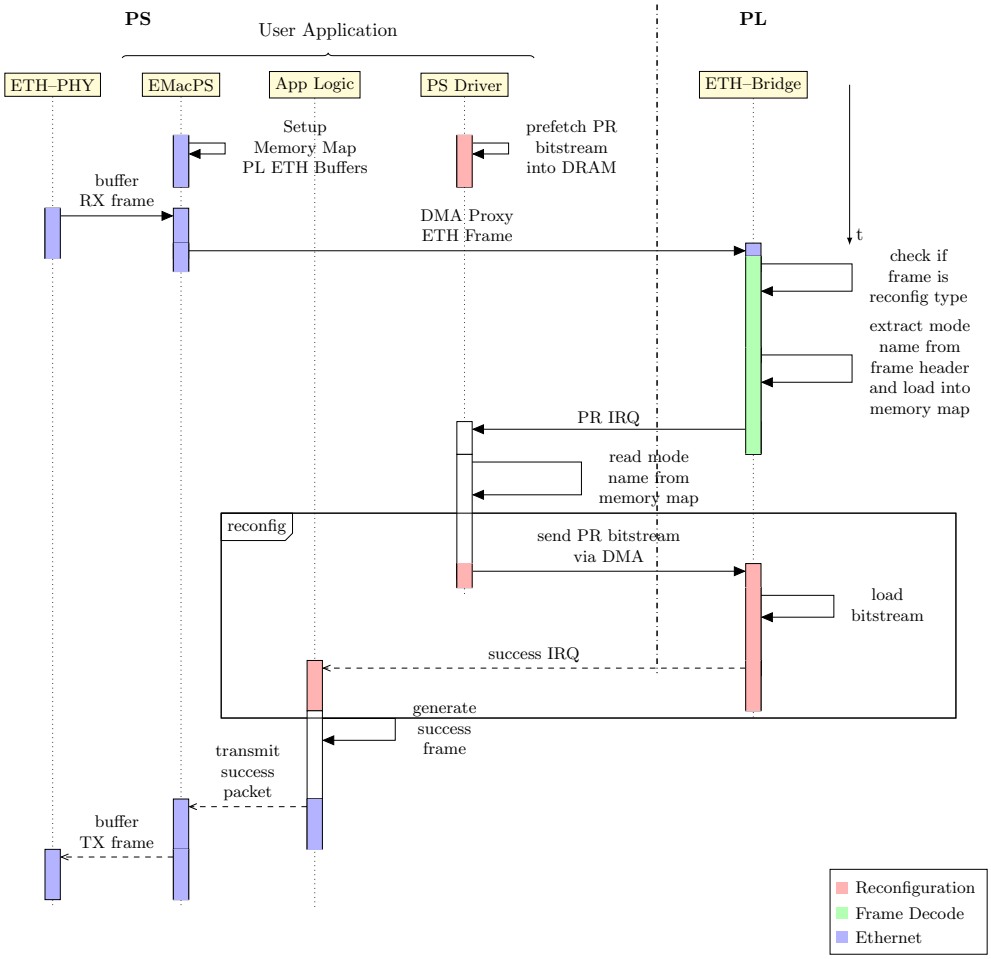


Figure 4.5: Sequence of events when the packet decoding is handled within the network interface in PL, while the reconfiguration is initiated from the PS using a custom reconfiguration manager.

4.6.4 Bitstream Over Network

A major benefit of bypassing the PS for frame decoding lies in the ability to receive new bitstreams over the network and to load them into hardware without the need for buffering/management at the software level; this is demonstrated in

research by Byma et. al. [96], where they provide a framework for virtualizing FPGA resources. This extends the use case for network attached accelerators, where sparingly used specialised kernels could be held in a central location and can be activated on request for accelerating specific tasks. Although locally cached bitstreams allow for faster turnaround times, storing all possible modes locally results in huge memory utilisation and overhead. Also, involving the PS to receive and manage bitstream over network creates unnecessary latency as well as data movement between the PS-EMAC, DRAM and from DRAM to accelerator slot (PR).

To demonstrate this case, we deconstruct the PR bitstream into remote reconfiguration request frames that trigger the network integrated PR controller to use the incoming bitstream information instead of a locally cached version. To model this use case, these frames are generated by a Python script within a host PC and sent to the target device via direct MAC addressing. On receiving the remote reconfiguration request, the bridge verifies the signature and triggers the integrated reconfiguration module to use the bitstream information. Subsequent reconfiguration data frames are also directed into the reconfiguration module (omitting the data headers) until the last frame is successfully received, completing the loading of remote accelerator into the PRR region on the device. In the test setup, we observed that the crypto-cores could be loaded from the host PC in 53.59 ms, measured from the arrival of the PR request frame within the PL to the completion of PR operation. Additionally, the received bitstream could also be buffered into the PS DRAM (caching), allowing the accelerator to be subsequently loaded from the local storage. We do not compare this to the time taken using a PS controlled flow as the performance benefits of PL decoding are surpassed by the time taken to transfer the bitstream across the network.

4.7 Summary

Heterogeneous architectures such as the Xilinx Zynq platform are a key aspect for the design of distributed in network accelerators. We present a strategy for targeting, initiating and loading partial reconfiguration over a network interface, bypassing the processing system. This strategy enables high performance, low latency partial reconfiguration for streaming applications that utilise custom hardware accelerators such as in network cryptography. We demonstrate that this methodology significantly improves upon time to reconfigure when compared to traditional PR strategies that require processing of Ethernet frames within the PS. Our case study demonstrates that the PS bypass approach achieves a 29.76% decrease reconfiguration trigger latency, with the addition that it frees the PS for other computation and reduces the variation in time to

handle incoming Ethernet frames.

We are looking to further reduce round trip time-to-reconfigure by performing the mode decoding and address lookup using a mapping in the register stack (performed in the PL). This would allow the software overhead to be further reduced as the frame decoding within the PL does not incur additional latency due to the contents being analysed inline within the data path upon streaming a frame into the PL. Additionally, we intend to develop the architecture presented in this Chapter to support Linux; this will allow for more complex accelerator designs, including higher levels of abstraction and integration with existing acceleration frameworks. We are looking to further build on the concept of intelligent accelerators by enabling them to actively manage caching of acceleration PR bitstreams. This would allow the accelerator to effectively prioritise recently used bitstreams and move bitstreams in/out of cache with situational awareness; close integration with an accompanying software framework would allow this to be intelligently managed on a PS process.

The work introduced in this Chapter lead to the discovery of the complexity and difficulty of designing PR based applications for the FPGA SoCs. The following chapters 5 and 6 investigate how build tooling and runtime management can be utilised to reduce this complexity.

Chapter 5

Design and Build Framework for Partial Reconfiguration on FPGAs

5.1 Introduction

FPGAs are capable of providing high performance custom computing for resource heavy data center applications as well as high efficiency and low power embedded edge scenarios. While FPGAs excel at accelerating specific computations, system tasks such as hosting an operating system and managing high level networking, are better suited to general purpose processors such as CPUs. The combination of FPGA hardware for high performance accelerators and general purpose processing systems or CPUs has led to popular heterogeneous SoC platforms from leading manufacturers, such as the Zynq and Zynq UltraScale+ from Xilinx and the Stratix, Arria, and Cyclone families from Intel.

While such devices have the benefits of both generalised and accelerated computing, managing the abstraction of custom accelerators in the programmable logic or FPGA fabric from the application processor can be challenging as it demands expertise in integrating low level hardware design with the higher levels of abstraction used for OS networking, and this is further complicated for domain specific design frameworks such as for machine learning. This challenge is further extended when the designer wishes to exploit specialised FPGA features such as Partial Reconfiguration, which allows the FPGA to modify/update specific regions while still processing data. Managing the state of the FPGA hardware logic from software running on a CPU is challenging, making designing and deploying such systems extremely complex.

In order for PR to finally become feasible in mainstream applications, a number of challenges must be addressed: (1) designing and building PR

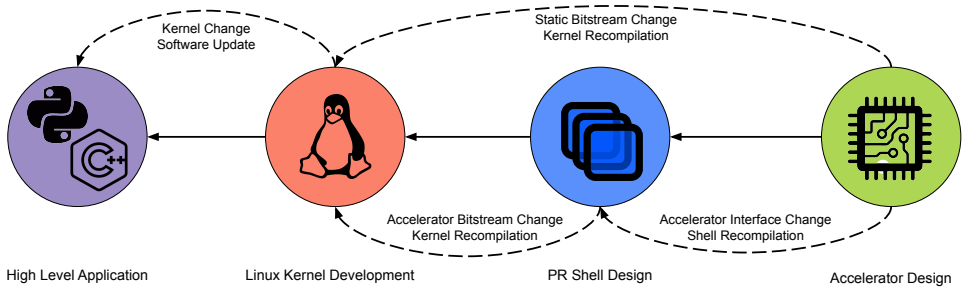


Figure 5.1: Example Linux PR workflow. Designers are required to propagate their changes up from the accelerator, through to the shell, the Linux kernel, as well as track PL changes from their high level applications.

systems should be possible by non-experts; (2) abstractions between hardware and software must be managed such that both PR and accelerator performance is not impacted; (3) interacting with hardware accelerators should not require driver level access — applications should run from OS userspace; and (4) fragmentation of vendor hardware should be managed by tooling; the tools should be modular to support new architectures.

Existing vendor as well as current academic tools typically either demand bespoke expertise, tightly weaved into the development flow from start to finish or isolate each step, requiring build tasks to be managed by separate domain experts [10]. Fig. 5.1 shows the four major stages of building for an FPGA operating system and how under a traditional vendor flow, changes at any stage (prior to the high level application) require the designer to adjust other aspects of the build. An OS runtime's control over hardware typically has no context of the build process, being only aware of what is in the FPGA through logic implemented by the end application designer, often using custom drivers, bespoke memory mappings and data transfer mechanisms. This applies significant overhead to either the knowledge requirement of the end application designer or the cumulative team that is building the various stages. Some frameworks such as Xilinx's PYNQ [103] platform do provide some hardware abstraction under a Python SDK and allow for independent RTL compilation, to reduce the need to recompile the Linux kernel, however this is typically at a performance loss, using slow and unmanaged methods for configuring hardware.

It is still difficult for edge application developers to build high performance accelerators using off-the-shelf hardware IP cores without highly specialised knowledge. Our motivation for developing this tool is to enable an independent designer to build PR accelerated Linux applications for modern edge applications. Designers should be able to use existing HDL projects or IP cores, leaving the tools to determine compatibility and build hardware infrastructure to support them.

In order to fulfil the demands of a workflow targetted for an independent

designer building PR accelerated edge applications on Linux, the build and runtime tools should be able to automatically: (1) Determine compatibility of PR modules and the static regions. (2) Generate PR configurations based upon a user supplied config file. (3) Export PR configurations (memory maps, bitstreams, register values) to a Linux image. (4) Implement runtime abstractions that allow software centric control of hardware state. (5) Support non-PS centric data generation/acquisition.

Meeting these criteria is an important step to democratising the use of PR in embedded applications, significantly reducing the complexity of heterogeneous systems design and deployment. introduced an early set of abstractions to support this automation workflow; we build upon these tools to release this end-to-end (E2E) tool allowing for complete abstracted PR application design and development. This Chapter and the following introduce the build and runtime components of a customer workflow designed to automate developing PR applications.

The work presented in this Chapter has also been discussed in:

- Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq UltraScale+. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 215–220, 2020. doi: 10.1109/ICFPT51103.2020.00037 [14]

- Alex R. Bucknall and Suhaib A. Fahmy. ZyCAP2: End-to-end build tool and runtime manager for partial reconfiguration of FPGA SoCs at the edge. In *submitted to: TRETS*, 2021 [18]

5.2 Contributions

The key contributions of this Chapter are:

- *An extensible end-to-end FPGA PR and Linux build tool* written in Python, for partially reconfigurable designs that automates the generation of infrastructure to support user logic and manages device tree overlays, drivers, and memory mapped IO

- *A standalone Python library for interface extraction and wrapping* PR module interfaces to enable easily building soft SoC infrastructure

- *An extension to FuseSoC EDA library* to enable building PR based workflows and simplify loading acceleration function from a common libraries

- *A comprehensive hardware-software build abstraction* to allow for simplification of hardware management from a user's software application based around the AXI standards
- *Support for edge-oriented acceleration* where non-PS sourced data paths are allowed for chaining reconfiguration regions as well as from external PL peripherals

5.3 Related Work

To effectively explain the current state of research for PR design flows, both vendor tooling and academic works are evaluated. Some of these topics have been explored as an overview in earlier chapters (see chapters 2 & 3), here they are critical analysed for their constraints and limitations.

5.3.1 Vendor Tools

Major FPGA vendors, including Xilinx and Intel offer their own implementations of PR, with varying degrees of support on recent FPGA SoC platforms. The build tools specifically target Xilinx's Zynq and Zynq Ultrascale+ devices as support for PR is more widely documented and there is a larger user community than for other vendors. There is potential to support other vendors such as Intel, within the tools presented in this Chapter, in future work.

5.3.1.1 Xilinx Vivado Design Suite

Xilinx Vivado is the suite of hardware tools for FPGAs, encompassed by the Vitis platform, to help users design, build and deploy custom bitstreams onto Xilinx FPGAs. This tool includes the workflows for synthesis, implementation, and place and route. While this software does provide workflows for some automation of the PR design flow using the DFX wizard tool, this is limited such that the designer must ensure that their PR modules correctly match the base design including interfaces, the allocated pBlocks or Reconfigurable Partitions (RP), as well as assembling and extracting any memory mapped addresses from such modules if required to be exposed at runtime to the Linux kernel. For these tools, Vivado from version 2019.2 onwards is specifically examined, along with the introduction of the DFX tools.

5.3.1.2 Dynamic Function Exchange

DFX is a Xilinx device feature that allows a user to dynamically modify blocks of logic by downloading partial bitstreams while the remaining logic continues to operate without interruption, referred to elsewhere as DPR. DFX provides

a simplified wizard for reducing repetition when creating PR modules but this only extends as far as allowing the user to automate the swapping in/out of PR modules into the build flow. The DFX suite does offer IP cores for managing reconfiguration including tools for managing the flow of data into and between static and PR regions but these do not support loading from an AXI-Stream transaction, such as with DMA from the PS.

5.3.1.3 Xilinx Vitis

The Xilinx Vitis unified platform comprises a collection of hardware and software layers for building embedded applications. In relation to the tools, we refer to Vitis SDK as Xilinx's previously named XSDK software suite for building bare metal and Linux applications. Vitis is typically designed to build C/C++ Linux applications for the Zynq and ZynqMP platforms where the user's code may be required to interface with kernel drivers or modules. It is intended to be used with the PetaLinux build workflow, for access to the kernel headers and dynamic libraries required for linking.

5.3.1.4 PetaLinux

PetaLinux is Xilinx's build tool for embedded Linux deployments and is based on the open source Yocto build tooling, using the same recipes and build structures to generate custom Linux images. Xilinx supports the ability to pass hardware configurations between Vivado and PetaLinux using their XSA compressed object, which specifies information about hardware, such as Memory Mapped Input/Output (MMIO) addresses, support for their own IP cores, and drivers such as the DMA controller. Fig. 5.2 shows the pipeline from Vivado, Vitis, and PetaLinux for generating the *boot.bin* Linux image that contains bitstreams, first stage bootloaders, as well as other power management firmware. While this pipeline supports static designs, due to the changing interfaces and modules, it is unsuitable for PR workflows as definitions such as MMIO register controls are not tightly defined, as the generated XSA file includes the build information for only the base design, not the subsequent PR modules.

Fig. 5.4 shows how the Xilinx tools are used together to design and develop embedded Linux applications. While there are automated aspects of this design flow, such as the exporting of AXI addresses (MMIO) to be used for generating a static Linux device tree, this only supports a traditional static design flow, with no support for PR hardware.

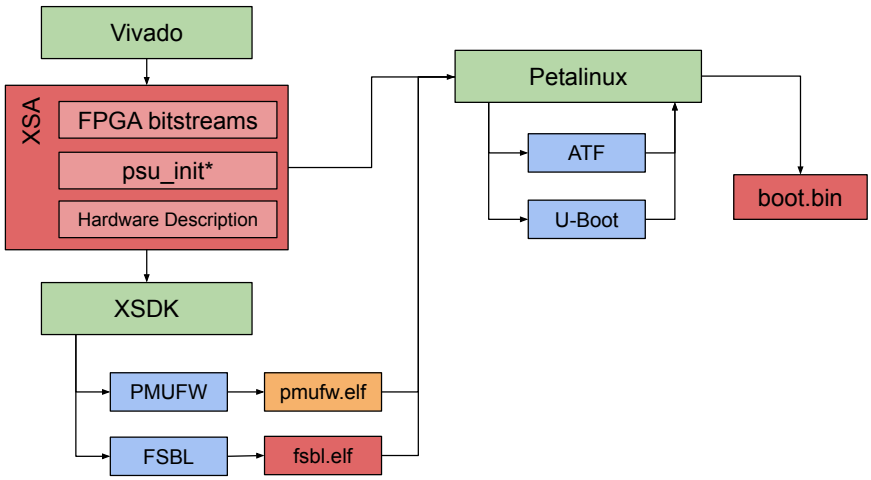


Figure 5.2: The Xilinx Linux build flow.

5.3.2 Current Research

Significant work has been conducted across various aspects of the PR workflow, in particular on the FPGA floor planning process, where floor planning is the spatial placement of logical designs on the FPGA, and/or on the scheduling of PR loading from the processing system. Tools like GoAhead [49] leverage vendor tooling for the building of reusable Reconfigurable Modules (RM), intending to make designs more portable and potentially compatible across different FPGA devices and PR applications. GoAhead can generate interfaces for PR modules to enable simple integration with the static region and abstract low level designing. [51] focuses on the DPR workflow for building RMs and automating RP generation; it is built on top of Vivado TCL scripts and offers a simplified workflow for building PR applications. Their tool, however, impacts the size of the partial bitstreams and thus further increases the time taken to load over the PCAP interface. More recently [104] provides an automatic floor planning methodology that enables the generation of IP, independent of architecture and able to target larger SoCs such as the Zynq Ultrascale+ devices. The work in [105] introduces a hardware architecture that packages FPGA resources as blackboxes, configurable at runtime where a software stack allows for task scheduling in the FPGA using PR to load accelerators. Their design focuses predominately on task scheduling and utilises a slot based architecture with fixed interfaces. The results demonstrate high efficiency when accelerator execution time exceeds the cost of reconfiguration delay.

Open source tools are essential to this field of research as there is not a single approach that can solve all of the issues of PR application development. Tools such as FPGA Operating System [10] are designed to isolate each element of the workflow such that designers with expertise can independently develop these components, while others [61] offer highly integrated workflows that

Table 5.1: Build Tool Comparison.

Framework	High Level Spec.	Partitioning	Floorplanning	Phys. Impl.	HLS Support	ZynqMP Support	Custom Static Logic	Device Tree Config	Driver Automation	Post Build Modules	Bitstream Prep.	Custom Root FS
Xilinx Vivado	○	○	◐	●	○	●	◐	◐	○	○	○	○
Intel Quartus	○	○	◐	●	○	○	◐	◐	○	○	○	○
ReconOS	○	○	○	○	○	○	◐	◐	○	○	●	◐
FOS	●	◐	○	○	◐	●	◐	◐	◐	●	●	●
ARTICo^3	●	◐	○	○	◐	●	◐	◐	◐	●	●	●
CoPR	●	●	◐	○	○	○	○	○	○	○	○	○
ZyCAP2	●	◐	◐	○	●	●	●	●	●	●	●	●

● : The step is fully automated by the tool requiring no designer intervention. ○ : No automation in this operation.
◐ : Partial automation is provided by tool.

tightly integrate the movement of data between PS and PL using custom APIs.

5.3.2.1 FPGA Operating System

FOS [10] is a recent build framework and runtime that modularises the design flow of PR applications for the Zynq and ZynqMP. It provides optimisations in the domain of abstracting the FPGA-Linux barrier to focus on resolving challenges to do with segmenting the design flow for interoperability across a team, enabling stages to be offloaded to bespoke designers with domain expertise. In their flow, the designer(s) must be aware of non-trivial considerations for PR hardware such as creating PL device constraints, generating a custom Linux device tree, as well as how to interface high level software applications with hardware accelerators, either through kernel drivers or userspace abstractions on kernel drivers. FOS supports compiling PR modules independent of the shell interfaces, performed by bitstream manipulation to dynamically assign physical fabric mapping to the PR modules allowing them to abstract resource allocation [3], as opposed to the traditional flow which requires locking a static design to implement a PR module. However, similar to other frameworks that utilise custom PR shells, there is limited support for interfacing peripherals such as Ethernet controllers, MIPI cameras, etc., directly with the accelerators, first requiring conversion into a standard shell interface (AXI4 or AXI4-Lite) which is not factored into the design flow. FOS targets applications where a development flow might be implemented across multiple designers working in isolation; the tools aim to enable a workflow for a single software designer to build accelerated applications. The FOS runtime supports a multi-tenancy scheduler tha manages provisioning for multiple clients. For the runtime, intended for a single user edge application, we choose not to build in unnecessary complexity to the framework, arguing that multi-tenancy is bettered intended for centralised offloaded compute applications.

Other tools [105] have also focused on scheduling of PR systems, aiming to provide a framework for efficient task switching and provisioning of the PL

under heterogeneous systems. While there is a significant body of work in this domain, the tools predominately focus on the abstraction as well as the build process for the designer. Instead an open runtime API and abstraction are provided to enable task scheduling to be built around the hardware and software infrastructure.

The work in [106] suggests extensions to FOS that provide plugin support for the ICAP. However, the driver and HDL code for has not been published to the open source code repository for FOS. It is difficult to quantify the features of their driver, such as if it supports asynchronous triggering of PR, without additional insight into how the driver functions. It does however appear to suggest that it offers reconfiguration directly from the network, described as remote configuration, which offers similar advantages as described in [92]. Table 5.1 show a comparison of current competing tools that target relevant devices and current vendor place and route tooling.

5.3.2.2 ReConOS

ReConOS [61] allows generation of PR bitstreams but these must be compiled along with the kernel as custom drivers and bespoke hardware components. This means that future PR modules require the kernel to be recompiled with the new bitstreams and associated drivers. Additionally, no device tree configurations for underlying modules are generated so PR modules that require specific configurations, such as differing memory maps, require recompilation of the kernel.

The group behind ReconOS recently expanded this vision for a hardware abstracted Robot Operating System (ROS2) platform built on top of ReconOS framework, ReconROS [107]. ReconROS features multithreaded programming interfaces for both hardware and software control with APIs for consistent programming models across hardware and software boundaries. They utilise the shell interfaces present in ReconOS with additional APIs that wrap the ROS2 subscriber/publisher methodologies to interface between hardware and software. A ReconROS application is designed in a similar manner to the ReconOS workflow, extending the original tooling to generate a hardware and software output but with ROS2 middleware accompanying. Their platform targets the Zynq-7000 platform and not the Zynq Ultrascale.

While existing PR frameworks provide access to PR with the compromise of overhead, portability, and performance, we offer a combination of lightweight build abstractions that extend vendor tooling with limited modifications to kernel drivers, focusing on abstracting control from the userspace while providing the highest possible performance for both PR and PL accelerators. Frameworks such as FOS divide the build process between multiple domain experts and

provide support for multi-user distributed accelerators. The abstractions focus on enabling a single user to develop and deploy high performance adaptive system applications within Linux. The complexity of standard vendor tooling presents a significant barrier to entry for new users and we attempt to address this with the tooling abstractions. These extend to the runtime API which means a developer without PR application experience can write the software to manage the reconfiguration seamlessly at runtime.

5.3.2.3 ARTICo^3

[9] is another automated toolchain designed for PR application deployment. They describe their tool as being able to dynamically adapt between trade offs with computing performance, energy consumption, and fault tolerance, meeting the demands of a cyber physical system. ARTICo^3 extends the ReConOS system bus; the designer is expected to conform workflow APIs for controlling PR kernels, which provide abstracted access to hardware, with the caveat that HDL or C/C++ kernels must conform to a shell defined interface. This means that accelerators (referred to as kernels in [9]) must be designed to fit the shell interfaces and thus their runtime API. This does provide an advantage to discretely allocate resources per PR shell such as local memory banks for each shell, where accelerators act as virtual slave peripherals in the AXI infrastructure. HLS libraries are offered to simplify kernel design, all custom accelerators should be designed with the expectation that the local memory blocks for each accelerator is used to move data between the accelerator and the rest of the infrastructure. This simplifies PR with a standard interface between the static and reconfigurable regions but adds design time complexity.

The ARTICo^3 runtime executes from the Linux userspace, written in C. Their API uses a custom kernel platform module to manage virtual-physical memory management as well as DMA control. While the kernel platform module is relatively lightweight, it means that any changes in the Linux kernel must be fixed in the framework, as opposed to using vendor drivers such as Xilinx's DMA driver.

5.3.2.4 Summary

Across the most recent and notable toolchains that target simplification of the PR build process, we identify a number of key issues concerning the build tools and runtime managers. Hardware kernels, the layers of software required to communicate with accelerator logic, are common across the different tools and push designers to learn custom workflows for designing/wrapping their acceleration functions. The drawbacks of tool specific hardware kernels include:

- Requirement to learn and implement accelerator logic adhering to the specifics of the target toolchain.
- Adapting application software to interface with tooling specific APIs.
- Increased software overhead as a result of layered abstractions to access hardware.

Additionally, there is no support for non-PS centric data transfer; accelerators are treated as isolated co-compute for the PS and externally connected devices such as high speed cameras and sensors must be connected first to the PS before data can be accelerated in the PL. The runtime managers included with these tools use Xilinx's FPGA Manager driver for reconfiguration, restricting the potential PR throughput and latency to that available with the PCAP interface.

5.4 Concepts

We define a number of important concepts in the context of our custom tools and specify their relevance to this work; in order to explain abstractions and concepts, we provide some definitions. We describe **states** as hardware changes that may be configured by setting AXI registers or by communicating with the hardware from another bus protocol (e.g. DMA stream path). We define **modes** as the functional hardware accelerators which can be loaded and unloaded from the FPGA using partial configuration. This includes the partial bitstreams that define the hardware accelerators. We refer to a **configuration** as a functional arrangement of modes that may perform an abstracted acceleration function such as multiple modes in the datapath, sampling and filtering the incoming data. A configuration may consist of a number of modes, abstracting the operating state of the hardware with both partial regions and the MMIO within the PR modules. In context of the build tool, a **specification** file is a file type that defines the parameters of the building workflow. This can be considered as a initial setup file, not to be confused with the definition of configurations; file is referred to as the `spec.json`.

5.4.1 Heterogeneous Systems on Chip

Traditional edge computing has utilised generalised computing, typically application processors (or CPUs), such as ARM A-Series processors. Application processors excel at tasks such as scheduling software running on top of an operating system, managing networking interfaces as well as generalised data manipulation and support for high level user applications and libraries. Contrastingly, FPGAs are programmable hardware devices best suited for

accelerating parallel tasks through custom datapath design. They are ideally suited to high data rate applications such as image processing or machine learning, where generalised compute might only be able to provide limited performance. However FPGAs typically operate at significantly lower clock rates than application processors and need to implement soft-CPU cores to execute software, limiting their performance for general computing. Heterogeneous SoCs couple application processors and FPGAs to leverage high performance and efficiency in both generalised and accelerated computing respectively, using high performance interfaces on the same chip. The application processor is connected to a wide range of external interfaces and comprises the Processor Subsystem (PS). The FPGA logic is generic as in other FPGAs and comprises the Programmable Logic (PL) region. Managing the abstraction interface between a CPU and an FPGA has been a long standing topic of research discussion, leading to many approaches to manage and ease the workflow between systems, in particular from the perspective of how the OS views the hardware in the FPGA. At the time of this work, Xilinx offers two main device families for Heterogeneous SoCs, the Zynq and Zynq Ultrascale+; the framework presented in this Chapter supports both devices.

5.4.2 Operating Systems

To control and manage a tightly coupled FPGA, an operating system may consist of a number of management layers, that include both the hardware and software infrastructure. We split the hardware management into three layers: controlling the status of the FPGA logic (PR or full reconfiguration), controlling the shell interfaces for moving data between the PS and the PL, and finally controlling the PL accelerator, such as managing modes, starting/stopping/interrupts, etc. While other embedded operating systems exist, in the context of this work we specifically refer to embedded Linux as it is widely supported on ARM processors and is the target operating system of major vendors' build tooling, such as Xilinx's PetaLinux.

5.4.3 Partial Reconfiguration

Partial Reconfiguration is the modification of one or more sections of an FPGA's logical resources during which the remaining sections or *static* regions are unaltered. [108] provides a wide overview of technical aspects of PR as well as academic work in the area that examines and addresses improvements and benchmarks concerning the technology. Dynamic PR describes the FPGA's ability continue to perform operations while undergoing the reconfiguration. Conversely, we label complete reconfiguration of the FPGA under a reset condition as static or full reconfiguration. PR is achieved by writing partial

bitstreams, generated specifically from the FPGA build workflow, to a configuration port on the FPGA and in the case of a heterogeneous SoC, is typically initiated by the application processor. PR has a number of benefits including time-multiplexing of hardware, making more efficient use of the logical resources available in the FPGA, effectively allowing for larger hardware applications to be deployed. Additionally, the time taken to update these partial regions is considerably shorter than static reconfiguration as reconfiguration is proportional to the size of the bitstream that is written to the FPGA's configuration port.

5.4.4 PR Design Workflow

The workflow for designing PR applications is complex, requiring expertise across multiple domains, including RTL design, operating system configuration, kernel driver, and high level software design. A typical workflow will include: designing the low level shells for PR modules, development of the PL accelerators, designing a custom Linux image with drivers and kernel support, as well as the high level application that will consume and control the the PL.

Complexity in the design of accelerator hardware can be expressed as a design challenge, beyond the scope of this research where numerous academic tools as well as vendor supplied frameworks, such as Xilinx's Vitis HLS, provide a software centric approach to designing hardware accelerators using simplified constructs in common software languages like C++. This Chapter presents abstractions for the reduction of complexity in the build times and run times specifically for PR applications. Reducing the development complexity of the hardware accelerators themselves, remain a design challenge as the paradigms for designing hardware are not a direct translation to high performance software design. We are able to leverage the fact that many of these higher level accelerator design flows have consistent interface generation, e.g. AXI.

5.5 Build Toolflow

The workflow differs from current tools with its aims and implementation details, generating PR infrastructure specific for serving edge applications. The ZyCAP2 tools provide a complete E2E FPGA to Linux workflow to tightly integrating PR systems design as a simple software centric design solution. Tools such as FOS [10] offer a design flow intended for multiple designers requiring specific domain knowledge, albeit compartmentalised for offloading, meaning that for an optimal design workflow, a diverse team of domain experts is required. The framework described in this Chapter offer a number of motivations:

- Trading fine-grain control at each stage of the design process for deeper abstractions, requiring less expertise from a single designer
- Offering better support for varying accelerator interfaces, by generating support infrastructure at build time rather forcing standardised shell interfaces
- Providing an alternative to a PS driven dataflow; with the expectation for high data rate sensors and peripherals to be attached to the PL
- Reduce the low level complexity of end-to-end PR software application development by use of mainline userspace-kernel software drivers
- Improving on loading throughput and latency for PR bitstreams for Linux applications

While academic works have examined the runtime of PR management, many of these tools still require the designer to use the standard vendor build workflows. This approach is appropriate for static designs as hardware details such as hardware memory address locations are fixed and can be passed between Vivado and the Vitis SDK tools using XSA export files, however this is not compatible with a workflow that generates multiple PR bitstreams. We provide an integrated hardware build toolflow that generates structural outputs that are used to implement the driver, software, and abstraction components required by the Linux build process and PR runtime.

The workflow is designed to allow a single user to implement accelerator cores directly with their high level software applications. The tools manage the shell generation through to Linux kernel changes to support the various generated PR modules, abstracted through configurations and modes. The build flow, both FPGA and Linux components, are written as an extensible Python command line interface (CLI) tool, allowing them to be used either as a CLI or Python library for custom build projects, such as automating for multiple device types.

5.6 Edalize & FuseSoC

The build tool utilises the *Edalize* Python library for interacting and controlling EDA tools programmatically [109]. Edalize is an open source community project, specifically selected for its extensible code base and its support for various vendor architectures. Edalize takes a configuration specified in a Python script such as parametrization for compile- and run-time (e.g. plusargs, defines, generics, parameters), library sources and any other tool-specific (in this case out-of-context workflow for synthesising PR modules) and creates

the necessary project structure (files and directories) along with build and execution strategies for the project. In the case of our tools, Edalize is used to inject, populate and template the TCL scripts with user proved variables and settings to specify the custom PR workflow generated by our tools. The tools generate a base project using Edalize (in this instance a Vivado project) and inject the custom configurations for the PR build tooling, based upon the data provided by the designer in the `spec.json` and from the interfaces that were extracted by the interfacer tool. The tools extend Edalize to support a sequence of out-of-context design runs that are created for the PR modules to synthesise design checkpoints that are required for the PR workflow.

Built above the Edalize library is FuseSoC, a library and package manager for HDL code. FuseSoC is designed to simplify the reuse of IP cores and allow for creating, building and simulating FPGA applications, written in Python. FuseSoC can also be used to create compile-time or run-time configurations for individual IP cores, test against simulation engines, easily port designs to new targets as well as set up continuous integration. Figure 5.3 shows how FuseSoC and Edalize work together to abstract the building process for example EDA tools shown. HDL cores or hierarchical descriptions of hardware in languages such as Verilog, VHDL, nmigen [110] and others are organised into libraries that can group and store their behaviour.

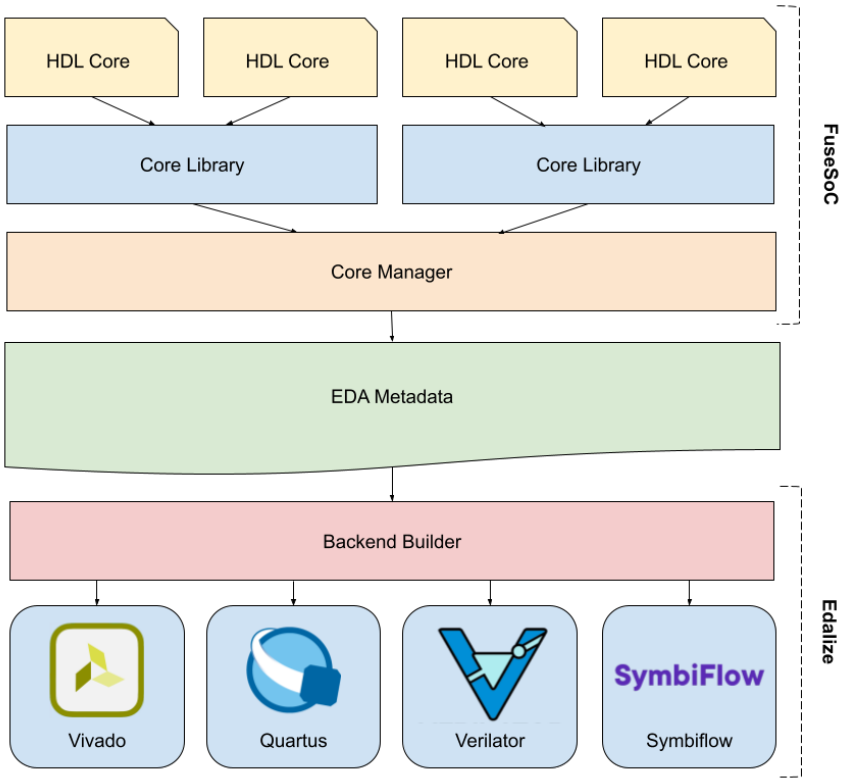


Figure 5.3: FuseSoC to Edalize workflow with example EDA tooling

FuseSoC translates the description of the core into a tool-agnostic metadata that is converted into project files and sent to the respective tool by Edalize. Listing A.1 shows an example core for a blinky project running on the Ultra96v2 device. Lines 2 to 15 define the groups of files and their paths that constitute a fileset and are used to compose different targets for simulation, testbenches or building of the cores. Lines 16 to 43 define the targets, in this instance the listing shows a simulation and build target. The core shown is standalone, there are no additional requirements beyond the files specified under the filesets in order to initiate building or simulation. Despite currently supporting a variety of tools including Xilinx's Vivado, Intel's Quartus as well as open source tools such as Symbiflow [111], there is no mechanism to instruct FuseSoC to generate output files required by a PR workflow, such as design checkpoints or the ability to route and lock the static region for interchanging PRMs during implementation.

As an extension project to this thesis, part of the Edalize was extended as an open source collaboration with GCHQ for their clustered FPGA acceleration computing group. The objective of this project was to investigate and extend FuseSoC and Edalize to support a PR workflow using Xilinx FPGAs, as a mechanism for building and deploying new acceleration functions/hardware. This work was inspired by an FPGA cluster managed by GCHQ that used a runtime to manage flashing bitstreams into the Intel FPGAs using static reconfiguration via a JTAG provisioning mechanism. Due to the size and complexity of the bitstreams being loaded, this was measured to take upwards of 30 minutes to perform provisioning, severely limiting the adaptability and the utility of their cluster. Their existing build workflow already used FuseSoC for design of their static acceleration functions as well for testing, simulation and verification and thus wanted to extend this workflow to be able to support PR. Their comprehensive FuseSoC library of IP cores was used to manage updating and testing of their VHDL-based hardware as well as continuous integration on this platform.

Initially we added additional functionality to Edalize to generate design checkpoint files (DCP) as outputs to the synthesis runs. Previously the Edalize workflow provided no mechanisms for interrupting a complete end-to-end static workflow, i.e. synthesis, place & route and bitstream generation. We modified the Python library to allow for passing optional parameters for the PR generation to be loaded into the configuration object that instructed Edalize's behaviour via a custom Vivado template-engine TCL script. This enabled the generation of the DCP outputs, required for building the PRMs. Following on from the PRM flow, the static region along with PRRs needed to be generated as this was also a non-supported workflow. The static region implementation run was also saved as a DCP and exported as later required. A build parameter

was added for the static region target that would allow for the pBlocks used in floorplanning to be passed to Vivado alongside the fileset. FuseSoC was then extended to allow for further additional parameters to specify if a core's fileset was for a PRM or a PR static run and which Vivado TCL templates to use. DCP files could be used as input source files, as PRMs or static checkpoints and to specify which arrangement of PRMs should loaded into the PRRs of the static checkpoints.

At the time of writing this thesis, these custom extensions made to FuseSoC have not been upstreamed into the ZyCAP2 project, in order to replace the current file management workflow with the FuseSoC PR version, due to time constraints. As such, we leave the integration of FuseSoC into the ZyCAP2 build tools as future work. Converting the ZyCAP2 project to use FuseSoC, in addition to the already existing Edalize backend, will enable more reusable IP within designs, better modularity, clearer separation of source and generated products as well as encourage greater adoption via the use of already popular tooling.

The upstream FuseSoC and Edalize source code is openly available under a GPLv3 and BSD-2 license, allowing for modification and implementation within the ZyCAP2 tools. The maintainers of the project are receptive to features and extensions and have supported bugfixes along the way to enable the features described across this section. As the project is versioned controlled using git as well as hosted on GitHub, it provided a mechanism to converse and work with the project's maintainers to adjust and tweaks our proposed changes to both FuseSoC and Edalize, ensuring they aligned with the project's goals. The intention is for this work to be contributed back to the upstream source code repository for the community to utilise.

5.7 Hardware Abstraction

Under the build framework, states, modes, and configurations are defined to abstract hardware control from the designer's software application [17]. 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. Fundamental hardware structure may change through modification of access to specific sensors or actuators (such as a radio switching from sensing to communication modes). These are referred to as distinct hardware *configurations*, that may require a different set of data interfaces between software and hardware. At

runtime operation, the decision logic communicates *configuration* changes to the hardware through a runtime or *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).

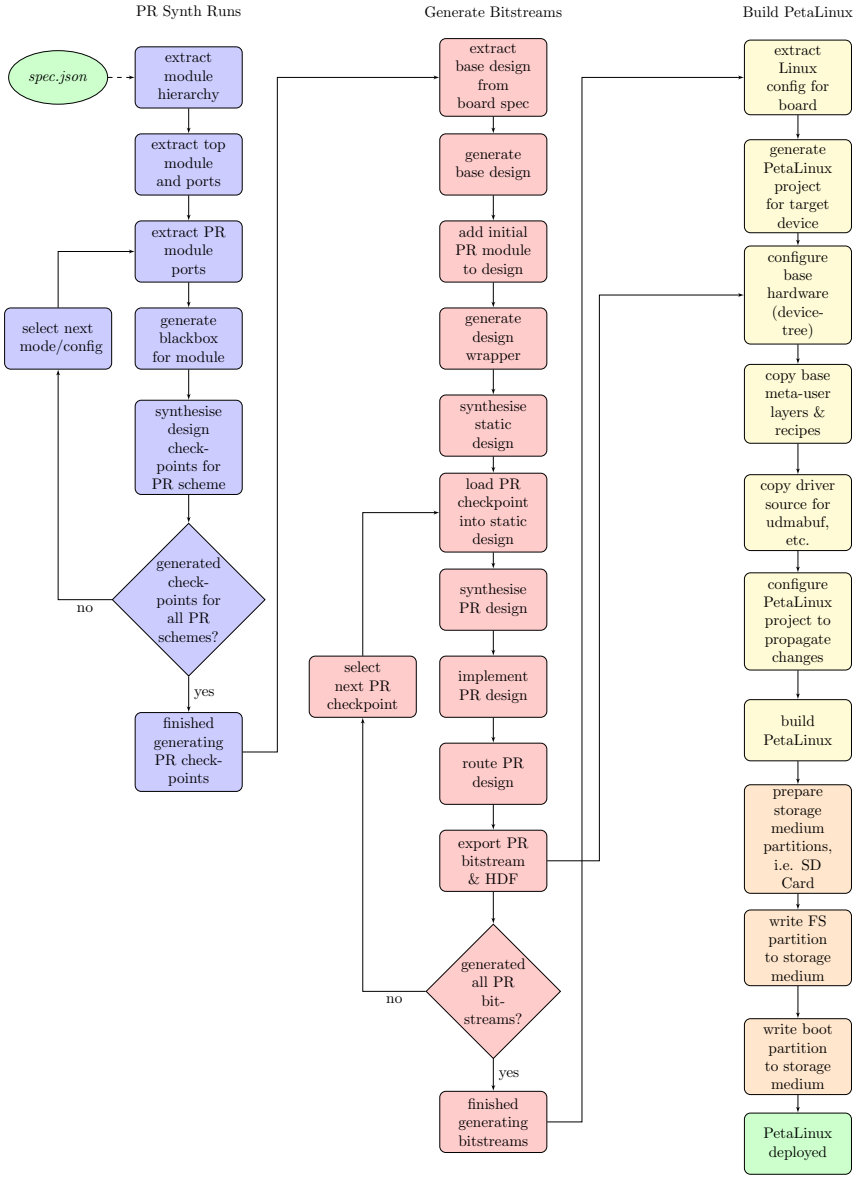


Figure 5.4: Stages of the PR build flow [14]

5.8 Infrastructure Generation

To resolve the complexities of integrating custom accelerators, the tools attempt to automatically build internal FPGA logic to accommodate interfaces and peripherals of the user's design. Fig. 5.5 shows the generalised architecture for

how shell logic and infrastructure is generated for corresponding PR designs. Currently the tool is capable of parsing Verilog top level modules for their required interfaces as the port and interface extraction leverages Pyverilog [112], an open source Verilog design processing toolkit written in Python. This however does not limit the provided IP from only being supplied as Verilog sources, the tool can also accept VHDL libraries and TCL scripts that are required to build the target modules. The only restriction is that the top level ports must be provided in Verilog so the tools can extract interfaces. High Level Synthesis generated IP cores can also be used as input sources for PR modules, supporting building directly from the imported core or pre-generating the IP cores to be consumed within the user's logic. We utilise Xilinx's AXI interconnect and AXI-Stream arbiter IP cores for routing data paths between the DMA controller and MMIO reads/writes from the PS. Any signalling within the PL is managed by the ZyCAP2 runtime, which will set the AXI-Stream arbiter master and slave addresses according to the applied configurations. To extend this further in the future, port extraction could be improved by providing full support for VHDL and SystemVerilog.

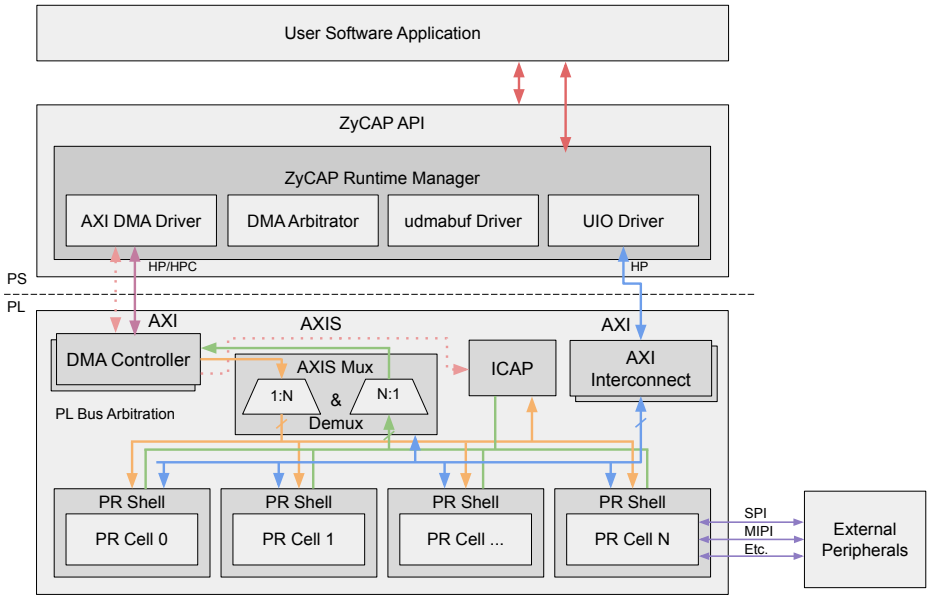


Figure 5.5: PL architecture generated using the ZyCAP2 build tooling.

5.8.1 Compile-time Generated Interfacing

To allow for a variety of PRRs, the build tool generates infrastructure at compile-time to support the accelerator interfaces. Parsed interfaces available on the module using a custom Python library *interfacer*, which is able to extract supported protocols, used by the build tool to match and generate infrastructure for including AXI interconnects and AXI-Stream arbitrators

(multiplexer/demultiplexers). Currently this supports AXI Full/Lite, AXI-Stream, General Purpose IO, Interrupts, Clocks, and Resets; it is designed to be extensible by the designer allowing them to specify their own protocols, such as using I2C and MIPI interfaces. The Interfacer library is able to extract interface widths and pass this information upwards to the toolchain which then determine the infrastructure to generate, such as setting default widths for interconnects as well as bus width converters, if required. The library does not currently handle differing/crossing clock domains between PRRs as this requires specific attention when floor planning. However the designer may specify their own pblock placements for accelerators and can manually accommodate for crossing clock domains with custom logic and pblock locations.

```
"VERSION": "0.0.2",
    "PROTOCOLS": {
        "AXI": {
            "STREAM_MASTER": {
                "DIRECTION": "output",
                "PARAMETERS": {
                    "PRAGMA": "(* X_INTERFACE_PARAMETER = \"{0}
    \" *)",
                    "PARAM": {
                        "HAS_TLAST": "bool",
                        ...
                    }
                }
                "INTERFACES": {
                    "TDATA": {
                        "REQUIRED": true,
                        "DIRECTION": "output"
                    },
                    ...
                }
            }
```

Listing 5.1: AXI-Stream Master Interface

Listing 5.1 shows the *interfacer* protocol definition for the AXI-Stream master interface, used to parse AXI-Stream interfaces and extend the generated AXIS arbitrator from fig. 5.5.

5.8.2 Automatic PR Region Generation

At present the generation of PR Regions takes place as part of the Edalize backend that the build tools are constructed upon but do not use the previously mentioned FuseSoC library management, this is left as future work.

The build tools will search the users's *specification* file for PRRs, assign the specified *pBlocks* and build the PR logic accordingly. Listing 5.2 shows a *specification* file, with a 2 region PRR, where a *chroma filter* is generated for

both regions but the *image resize* and *gaussian filter* may only be generated in *region_a* and *region_b*. A user may choose to do this if they know a module is resource intensive as there is only a finite selection of logic available in the PL. The tools will assign the RP and then create synthesis and implementation runs for each region in an *out of context* workflow, allowing for the modules to be used in PR bitstream generation. Currently the tools will warn the user resource requirements exceed the PRR specified (specification shown in listing 5.2). These warnings are generated when the logic demanded by the accelerators cannot be provided by the region specified by the PRR.

Fig. 5.6 shows an example of the post-build generated wrappers for the user's HDL modules. In this instance, the tools generated 2 wrappers (supporting 2 configurations), an AXI Lite for control and an AXI-Stream interface for data streaming. The generated wrapper will at least contain a union of the interfaces of the underlying modules, where any interfaces unused by a module are automatically tied off. This can support varying sized interfaces, for example one module with a 32bit AXI-Stream and another with a 64bit interface, with the caveat that performance may be degraded if data-width conversion modules/IPs are used.

We choose not to address the issue of floorplanning within the tool and instead provide standard slot-based PRR for the supported devices, with layouts for 1 to 4 accelerator partitions. These slot definitions are stored with the board files and the tools provide a flexible mechanism (via Python API) to automate this resource allocation, if required. We provide the ability to specify the pBlocks via the specification files such that external floorplanning tools may be used in conjunction with the build tool. Significant research [113] [114] [115] has already been conducted in this space and thus we consider custom floorplanning to be out of the scope of this work. In future work, we intend to automatically allocate the pBlock regions intelligently, without manual input from the designer, based on methods such as those described in [116], [117].

```
{
    "pr_regions": {
        "region_a": {
            "pblock": [
                "SLICE_X36Y121:SLICE_X47Y155 DSP48E2_X3Y50:DSP48E2_X4Y61 ..."
            ],
            "default_config": "config_a",
            "configs": [
                "chroma",
                "resize"
            ]
        },
        "region_b": {
            ...
        }
    }
}
```

Listing 5.2: Multi-region specification with pblock definition

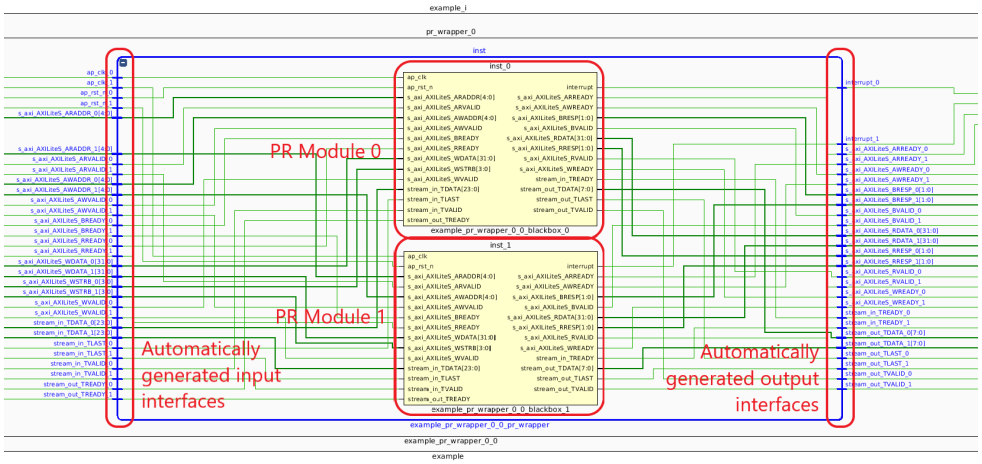


Figure 5.6: Synthesis schematic after build tool generates wrappers for each PRR

5.8.3 PR Module Chaining

The tools provide a chained region generation feature for edge applications, where multiple accelerators may be connected directly together to better serve streaming data such as image processing or in-network packet processing. Currently this feature supports AXI streaming interfaces where the user can specify that the master and slave AXIS interfaces are connected to another accelerator rather than directly back to the PS (via DMA). The compile-time generated infrastructure allows PRRs to be connected to each other, in the

described data chaining pipeline. Traditional shell based accelerators must move data first to the PS before it can be redirected to another accelerator, which disadvantages PL acceleration when the PL is the data ingress for sensors and peripherals. The tool allows a user to declare in the specification file, if regions should be connected to each other or to even external IO on the FPGA for outboard sensors or peripherals. This type of design is amenable to edge acceleration where accelerators are likely to ingest data from sources either connected directly to the PL before the data arrives at the PS or where data might require manipulation in a sequence of accelerators.

5.8.4 Customised Base Design

Under the default settings, the base design will encapsulate the user's accelerator with a compile-time generated shell, built from the interface information extracted by the interfacer library. This can be overridden with custom base designs to allow interfacing with external interfaces for example, high speed camera interfaces such as MIPI CSI, assuming the underlying hardware supports this. The default layout utilises a single DMA controller with generated arbitrators for both the ICAP control interface and any accelerator modules that use AXI Streaming interfaces. The AXI Streaming interface is particularly important as it enables a continuous block of memory to be transferred between the PS and PL, such as with edge applications such as image processing or network traffic analysis. The default workflow scales according to the number of user PRRs and exposed AXI interfaces identified within those PRRs. Additionally this may also be overwritten to isolate the reconfiguration DMA (for ICAP) and a unique DMA for the accelerators (using a dedicated HP(C) port), where the DMA may also be configured as video DMA or standard DMA. We target a base clock of 200 MHz for designs but can split the clock into a 200 MHz clock for the ICAP and a lower speed clock for the accelerator, if the accelerator does not support such frequency. The memory addressing for the control of the DMA arbitration to ICAP and accelerators is abstracted within configuration files and set by the Linux runtime manager.

5.8.5 Internal Configuration Access Port

For high performance reconfiguration, we choose to utilise a hard ICAP primitive within the PL for streaming partial bitstreams. The ICAP is a hard macro available in modern Xilinx FPGAs, with minor differences between the ICAPE2 macro on the Zynq and the ICAPE3 on the ZynqMP. The ICAPE3 officially supports transferring bitstreams at a clock frequency of 200 MHz and provides more output signalling than ICAPE2, such as with error statuses and the ability to trigger status interrupts.

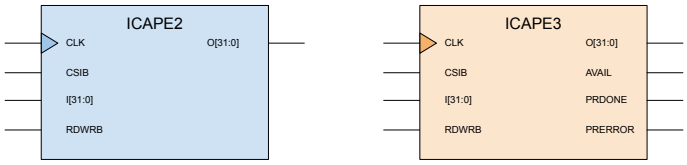


Figure 5.7: ICAPE2 and ICAPE3 macros.

The tools automatically determine the target device and deploy infrastructure accordingly. If targetting a Zynq device, the ICAPE2 will be used at 100 MHz and if a ZynqMP is selected then the infrastructure builds for a 200 MHz clock and the ICAPE3 signalling. The tooling can also be told to share a DMA controller between the ICAP and the user's accelerators, in this case the tool will prioritise the accelerator clock frequency and clock the ICAP according to the slowest common clock. We use a Python toolbox for building digital hardware, nmigen [110] to generate the interfaces for the ICAPE2 and ICAPE3 at compile-time. This allows us to parametrically build the interfaces and signalling for the PR controller, routing ICAP status signals back to the PS over a common AXI-Lite interface.

5.9 Linux

We support building from the hardware flow output (Vivado) directly into a pre-prepared Linux image (via PetaLinux), implementing and generating the requirements for each configuration and its supported modes from hardware. This information is passed from the FPGA build stages in the form of configuration JSON objects consisting of the memory address mappings, PR bitstreams and default values for MMIO registers. This is used for configuring the Linux image to allow for ICAP access, generating device tree overlays, preparing userspace drivers, preloading bitstreams and configuration files. The final compiled Linux image contains a pre-built runtime for the Zynq or ZynqMP, which is discussed in Chapter 6. The Linux image produced as a result of this toolflow uses our custom kernel (assembled with PetaLinux) and can used any provided Root Filesystem; by default the standard PetaLinux RootFS is packaged but versions of ARM Ubuntu and Debian have been tested.

5.9.1 PMU Firmware

In order to control the ICAP from the ZynqMP's PS, the control registers in the Configuration Security Unit (CSU) must be whitelisted for access; to do this, it must be enabled from the PMU firmware upon booting. The PMU is a hardened Microblaze [118] processor embedded within the ZynqMP's processing system, responsible for power, error management as well as managing access to

the CSU control registers. Under the vendor provided firmware, these control registers are blacklisted and the Linux kernel may not access the registers that allow for toggling between PCAP and ICAP control [15]. Due to this restriction a modified version of the PMU firmware must be built, that enables secure access to specific register addresses, in particular the `0xFFCA3008` register which toggles bitstream loading between PCAP and ICAP (it defaults to PCAP). The design tools automatically handle the versioning of this firmware, pulling it from its source, injecting the required flags and building it within the context of the ARM ATF to grant the appropriate permissions.

5.9.2 Device Tree

The Linux kernel builds a mapping of the hardware made available to itself using a Device Tree (DT). Typically on embedded ARM based architectures, this DT is constructed at build time to allow the kernel to load drivers in relation to the hardware described as connected, for example hardware that is memory mapped or made available over a specific interface such as I2C. Considering that the PL may be treated as generically definable logic, a designer may choose to implement a number of custom processor peripherals such as memory mapped or streamed interfaces (via memory mappable DMA, that may require internal switching), thus it is important to track hardware changes with a dynamic device tree. As the tools do not enforce strictly defined shell interfaces, as such the ability to update the device tree is important for allowing the kernel to track the location of memory maps within the FPGA.

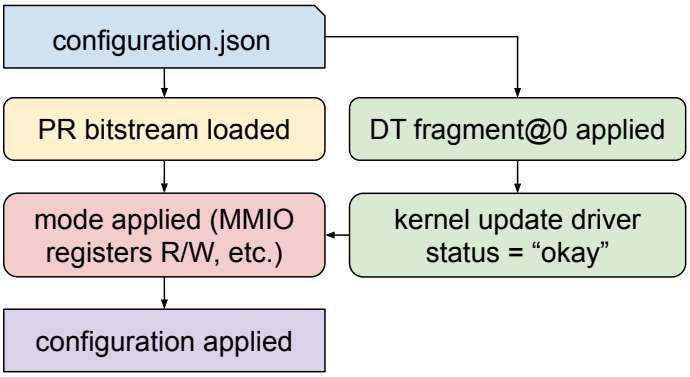


Figure 5.8: Applying DT fragment via configuration

5.9.3 Device Tree Overlay

The 3.18 release of the Linux kernel introduced the device tree overlay (DTO), an implementation of the in-kernel device tree which can be used to modify the kernel's live tree and affect the running kernel, such as applying driver changes, registering and deregistering nodes, in turn loading/unloading modules. This

```
axi_accel0: axi_accel@0 {
    compatible = "linux,axi_accel";
    status = "disabled";
};

fragment@0 {
    target = <&axi_accel0>;
    __overlay__ {
        status = "okay";
    };
};
```

Figure 5.9: Example of Vitis HLS Pragma for AXI Stream Slave/Masters.

can be performed through the use of device tree fragments or sections of the device tree that should be swapped in for new functionality. Xilinx's FPGA Manager offers the ability to update the DTO while programming bitstreams but does not possess the ability to import these overlays from the Vivado hardware build process or produce the overlays from a DFX workflow. The build framework uses the data from the Vivado build process to generate custom DTOs, describing required modes and configurations which are then stored alongside the bitstreams for PR. Currently this is performed for AXI and AXI-Stream based accelerators, using the MMIO addresses and the DMA arbitrator location, generated at build time, respectively. It is possible to generate custom DTOs as generic drivers are used for PS-PL data transfer; IP core specific drivers can be used but must be manually specified. DTO loading is crucial when applying configurations that require changes to nodes of the live device tree, for example alerting a driver the status of a hardware module as shown in listing 5.9 and fig. 5.8. Listing. 5.9 shows an example AXI accelerator device tree node that would be overlaid with the `fragment@0` fragment to enable the node and associated drivers.

5.9.4 Kernel Drivers

To accommodate varying PL peripherals, we opt for generic PL drivers to handle data transfer between the PS and PL rather than rolling custom framework specific drivers. The configuration of the drivers, required in the device tree overlay, are generated for a specified configuration during the Vivado build process. Device tree nodes are generated for all of the available MMIO (AXI) addresses as well as the position of any AXI-Stream interfaces, connected to the input and output of the PL DMA controller. The drivers utilised in this framework are discussed in more detail in Chapter 6, within the context of the runtime manager.

5.10 Evaluation

We evaluate the E2E build and runtime tooling in terms of both runtime performance and build complexity. It is important that the provided abstraction has minimal impact on both user accelerator and software performance. The evaluation is performed on a ZynqMP development kit, the Ultra96v2 (Xilinx Zynq UltraScale+ MPSoC ZU3EG [119], shown in Figure 5.10), and build time evaluations are conducted using Vivado 2019.2 on a 6 core 12 thread Intel i7-10750H running at 2.60 GHz with 32 GB of RAM.

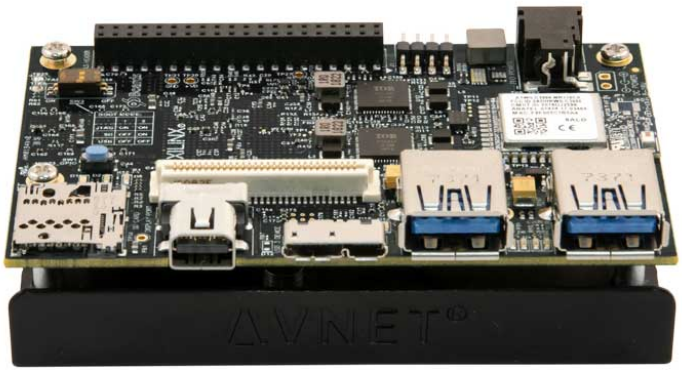


Figure 5.10: Avnet Ultra96v2 Development Kit

5.10.1 FPGA Resource Consumption

Logical resource consumption is an important metric for custom infrastructure as the more resources consumed by the framework, the less that is available for user accelerators. The framework scales the shell according to the number of modules and required interfaces provided by the user. For example, if there are no AXI-Stream interfaces in any of the user's accelerators, the tools will not generate an AXI-Stream switch.

5.10.1.1 Compile-time Generated Infrastructure

The overhead of the compile-time infrastructure is demonstrated with a varying selection of custom accelerators (see case study in Chapter 6) with the respective interfaces that they expose and measure the resources consumed by the infrastructure generated required to support the described interfaces. Table 5.2 shows the resources across a selection of arrangements; the consumed resources never exceed 9% of the LUT utilization of the PL. This leaves over 90% of the PL resources for the user's accelerators or additional shell logic, if required to interface with external hardware. Considering that this is a small UltraScale+ device, larger devices will suffer even less of a fractional overhead.

Table 5.2: PR Manager static PL resources.

PR Region Interfaces	FFs	LUTs	BRAMs	Total LUT Utilisation of PL (%)
1 AXI4-Lite (32-bit) + 0 AXI4-Stream (32-bit)	7688	5132	5	7.27
2 AXI4-Lite (32-bit) + 0 AXI4-Stream (32-bit)	7738	5142	5	7.28
1 AXI4-Lite (32-bit) + 1 AXI4-Stream (32-bit)	8817	5619	5	7.96
2 AXI4-Lite (32-bit) + 2 AXI4-Stream (32-bit)	9063	6051	5	8.58

The generated infrastructure has some limitations enforced by the IP cores that are used to generate bus routing. Both the AXI-Stream Switch (v3.0) [120] and the AXI Interconnect (v2.2) [121] can support up to 16 Master and/or Slave interfaces. The tooling has a soft limit to prevent the user creating more interfaces than a single switch or interconnect can support, although in practice it could be possible to support more interfaces. We measure the resource consumption by subtracting the resources consumed in the PR regions from the overall resources required for the complete design.

5.10.2 Build Time Complexity

While it is difficult to quantify the impacts of abstracting the build workflow, given the variation of efficiency due to designer's knowledge, development machine performance among other variables (i.e. available processing threads and memory), a timed build run for the tools is provided, that shows the generation of 4 specified partial modules combinations. The tools automatically assemble the project for a given *spec.json* and thus we quantify the time taken for each of these steps to be performed by the build tool. Aspects of the build process are handled by Xilinx's own tools, Vivado and PetaLinux and are applicable to any PR build flow however we can measure the time taken for assembling the build projects, extracting ports and interfaces as well as crafting the Linux build inputs such as device tree overlays.

We measure the time taken to run the Vivado automated build tooling for the case study design, where this is representative of the first two columns of fig. 5.4. To quantify the complexity, time taken for port and interface extraction is captured as well as the time for a complete Vivado build. For a 3 configuration design with 1 AXI-Lite (slave) and 1 AXI-Stream (master & slave), port extraction takes **37.94** seconds and the total build time is **2427** seconds for each partial bitstream and the full bitstreams to be generated. To compare this against a complex design, assembled and constructed manually, this equivalently could take hours or even days, given build failures, user error, etc. Comparatively the overhead for the extensions to the workflow are negligible compared against the vendor locked aspects such as synthesis and implementation.

5.11 Summary

In this Chapter, a toolflow for automating the build process of PR designs, from HDL through to a final ready-to-go Linux image populated with hardware configurations is presented. The tools take the logic from user applications, extract interfaces and generate infrastructure at compile-time to support data transfer between the PS and PL. These tools hide the complexity of the PR build process and the hand over to PetaLinux for Linux kernel compiling and filesystem construction. A demonstration is later provided showing how the framework can be used, with a vision processing case study that uses the discussed build framework to generate a design for HLS-based accelerators chained together for tuning image quality in Chapter 6.

We recognise that there are limitations to scope of this framework in its current state, as such it is useful to address what is **presently unsupported**:

- Further simplification of writing RTL (Xilinx's Vitis HLS and others already exist)
- Explicit floorplanning optimisations (tools allow for use with other tools/default pblock placements per device)
- Runtime PR resource scheduling (existing academic tools already perform this function)

We offer these features for future research as well as an intention to integrate with the work conducted with the FuseSoC [122] IP library tool, to allow for users to easily fetch packages from libraries and include/build them in their PR designs. This would allow for users to easily include modules for their PR designs and reduce the RTL design complexity. We intend to further extend HLS support to expose the HLS generated module's internal registers and allow generated MMIO register maps to be exposed via the runtime API. As well as extending the feature extraction from HLS modules, we would like to extend the FuseSoC workflow mentioned in Section 5.6 to allow for directly configuring, generating and building synthesised PR design checkpoints from Vivado HLS and export associated register metadata to the Linux builder for the generation of configuration and mode files. Furthermore it would provide additional abstraction to integrate this workflow with Xilinx's PYNQ tools, implementing low level optimisations under lightweight Python wrappers. The framework is available at `http://github.com/warclab/zycap2` and the Python library at `http://github.com/warclab/interfacer` as open source repositories for wider adoption and contribution by the community.

The following Chapter 6 follows on from the build process, with a runtime management tool and API for initialising the PR process as well as controlling

the data exchange between the PS and the PL.

Chapter 6

Partial Reconfiguration Runtime & Configuration Management

6.1 Introduction

The design time build workflow discussed in Chapter 5 is essential for increasing the abstraction presented to a PR systems designer. This workflow introduces a number of generalised runtime decisions required to manage adaptive systems, of which are more suitable to software running on a general purpose processor. As as output from the build process, a Linux image along with artefacts such as the configuration, mode and device tree overlay files are exported, ready to be applied as needed at runtime. At runtime the operating system must know how to apply these configurations from software based upon event-drive scenarios, such as loading an image processing accelerators when motion is detected. Existing tools provide mechanisms to perform reconfiguration as well as read and write to hardware but these are low level interfaces that require explicit knowledge over the address spaces where accelerators cores exist, which partial bitstreams are associate with which configuration as well as how data should be moved between the PS and the PL (e.g. via DMA transactions). An adaptive systems designer is required to possess the bespoke knowledge of loading PR bitstreams as well as explicit control over the hardware mechanisms from within their application's code (typically running between both the Linux kernel and the userspace). In addition to this existing vendor tools that provide abstraction over PR suffer from poor performance, where limitations can negatively impact the responsiveness of the system. The introduction of the ZynqMP architecture also brought complexity for managing PR; access to registers that control the ICAP are restricted from the standard Linux secure privileges. A number of features are needed by a modern runtime manager to

hide the complexity of these processes from the user. This Chapter introduces the work undertaken to design and develop a high performance, non-blocking PR runtime and configuration manager for the Xilinx Zynq and ZynqMP architectures.

The work presented in this Chapter has also been discussed in:

- Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq UltraScale+. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 215–220, 2020. doi: 10.1109/ICFPT51103.2020.00037 [14]
- Alex R. Bucknall and Suhaib A. Fahmy. ZyCAP2: End-to-end build tool and runtime manager for partial reconfiguration of FPGA SoCs at the edge. In *submitted to: TRETS*, 2021 [18]

6.2 Contributions

The key contributions of this Chapter are:

- *Improved high performance asynchronous PR controller* for loading the ZynqMP ICAP interface at near theoretical throughput (approximately 757 MiB/s)
- *A runtime PR configuration API* for PS-PL management that enables simple software abstraction of memory mapped IO, DMA streaming as well as loading/unloading partial and complete bitstreams as part of the described mode and configuration abstractions
- *A case study using Vitis HLS generated OpenCV edge accelerators in a PR application* that demonstrates the software abstraction and benchmarks the performance impact and resource consumption

6.3 Related Work

Most PR managers for FPGA SoC support the Xilinx Zynq, however the differing architecture of the ZynqMP requires the PS for loading bitstreams and thus a software layer is required to perform initial reconfiguration, either from within an OS or on bare metal. Most modern tools are built on top of Xilinx's FPGA Manager tool, including those noted in Table 6.1, for loading PR bitstreams and use the PCAP interface. Some of the mentioned works extend the functionality of FPGA Manager but are generally limited at runtime by the drawbacks of FPGA Manager, including those discussed in subsection 6.3.1. We

Table 6.1: Runtime Comparison

Runtime	Linux Support	Bit. Management	Non-Blocking PR	Multi-User	ICAP Support	ZynqMP Support	PR Scheduler	Userspace API	Generic Drivers
FPGA Manager	●	○	○	○	○	●	◐	◐	○
ReconOS*	●	●	○	○	●	○	●	◐	○
FOS*	●	●	○	●	◐	●	●	●	◐
ARTICo^3*	●	●	○	●	○	●	●	●	◐
ZyCAP	○	●	●	○	●	○	◐	○	○
ZyCAP2	●	●	●	○	●	●	◐	●	●

● : Fully supported. ○ : Unsupported. ◐ : Partial support or allows other tools to implement. *Built on top of FPGA Manager.

argue that FPGA Manager is unsuitable for high performance PR applications such as image processing or inline data streaming as the reconfiguration latency and throughput is comparatively low compared to other methods, later highlighted in subsection 6.7.1. Reconfiguration time may be in the region of tens of milliseconds, for typically sized bitstreams, of which crucial data may be lost/missed in this period, given a high performance application.

The work in [123] examines high throughput reconfiguration channels using the PR performance of the ICAP (Internal Configuration Access Port), while others have overclocked the primitive to see throughput of close to 800MB/s [124]. These works have targeted standalone FPGAs or older architectures and do not provide programming capacity for a tightly coupled processing system such as on the Zynq or ZynqMP devices. ZyCAP [11] introduced the concept of a high-throughput hardware controller along with a high-level software controller running on the Xilinx Zynq-7000 processor. It achieved a reconfiguration throughput of 382MB/s from the PS to PL over ICAP but was limited by a lack of support for a full OS such as Linux as well as limited support for high-levels of hardware abstraction. Our work in [17] showed how the ICAP could be programmed over DMA on the Zynq Ultrascale+ devices and the authors of [106] increased the clock frequency of the ICAP to 200 MHz, further increasing performance to close to the theoretical 800 MB/s.

6.3.1 FPGA Manager

Most Xilinx Zynq and all current Zynq Ultrascale+ PR runtimes use Xilinx's supplied FPGA Manager driver which abstracts the Processor Configuration Access Port (PCAP) interface for loading PR modules. FPGA Manager is a general reconfiguration driver available in the Linux kernel for controlling/provisioning tightly coupled FPGAs from Linux. FPGA Manager uses the PCAP on the SoC to load the PL with either static or partial bitstreams. In order to

load the PCAP, FPGA Manager must perform a sequence of register reads and writes to the Configuration Security Unit (CSU) registers which are managed by the ARM Trusted Firmware (ATF) and then the Platform Management Unit (PMU).

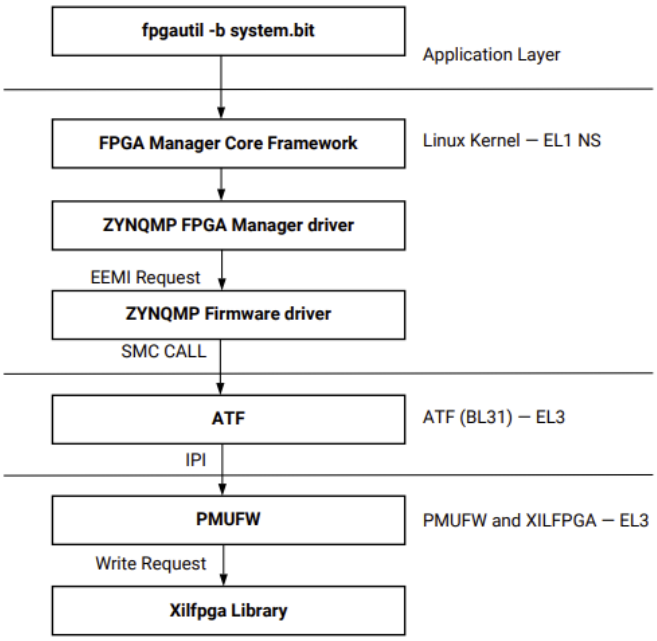


Figure 6.1: Loading of the PCAP from FPGA Manager (ZynqMP)[15].

Once the PCAP is set up and prepared for loading, a DMA transfer can be performed to the CSU, containing the target bitstream for flashing, as shown in Fig. 6.1.

6.4 Runtime Abstraction

Managing the FPGA abstraction from the PS at runtime requires a software layer to determine which hardware interfaces are exposed to the user and how to apply the target configurations. The runtime is designed to run in the Linux userspace, providing an API to user applications and abstract how PL hardware is controlled and how data is moved between the PL and the PS. This is one area where many existing PR frameworks have not dedicated much effort, assuming that the designer should define the specific loaded bitstreams at runtime rather than abstracting this based on the modes defined during the build phase. The abstraction aims to enable the high level adaptation logic to be written independent of the low level reconfiguration details, without needing knowledge of where bitstreams are stored, how to load device tree overlays or read/write from specific memory addresses in the PL, etc.

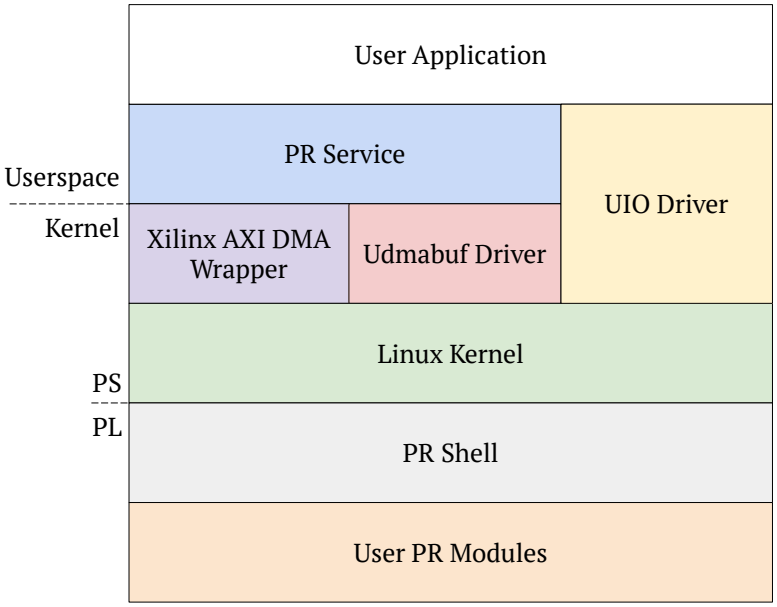


Figure 6.2: ZyCAP Linux Stack [14].

6.4.1 Hardware Resources

Hardware is abstracted into JSON configuration files that describe the target state of the FPGA, expressed by the modes and configurations, and how the userspace can interface with the current logic in the PL. The location of the PR bitstreams and arrangement of RMs and RPs is handled by the configuration manager.

6.4.2 Device Tree Overlay

The Linux kernel uses the device tree to instruct the operating system to what physical hardware interfaces are available to the kernel. In recent versions of the Linux kernel, support for device tree overlays has allowed changes to be made to the device tree during runtime and can be applied with the kernel's *configfs* interface. For the PL, this can be used to dynamically load and unload hardware accordingly what is loaded in the accelerator slots at any point in time. During the building process, Vivado generates a compressed export directory, an XSA (HDF in older versions), that contains a map of IP cores that have accompanying Linux drivers. The PetaLinux build process uses this to construct a device tree, providing driver and memory mapped support for supported IP cores. While this works for static PL bitstreams, it generates a full device tree for the static hardware, without support for dynamic logic in PR regions. The PetaLinux generated device tree is used to construct design specific DT fragments, injecting extracted memory address maps and clocks logic from the build time generated infrastructure to create fragments for each

configuration.

6.4.3 Linux Userspace Drivers

The runtime service executes exclusively from userspace to utilise the multitude of software libraries available, unlike the restrictive nature of kernel drivers. We make use of existing mainline drivers such as UIO and a lightweight userspace wrapper for Xilinx DMA Driver, to reduce dependency on kernel compatibility. This has the advantage of being independent of kernel, reducing security risks of providing direct hardware control to the user, reducing the likelihood of dangerous bugs impacting the kernel and allowing for the reliability of existing upstream vendor drivers as opposed to custom drivers.

6.4.3.1 Generic Userspace IO

The Linux kernel ships with a module known as the Userspace IO (UIO), which can be used to communicate directly with memory mapped devices, from the Linux userspace. Using memory addresses generated from the PR build process (extracted to PR configurations), the ZyCAP runtime gives the software developer abstracted access to these UIO registers, without requiring them to directly initialise and setup these modules, themselves. For HLS generated modules, this can be extended to provide internally addressable registers as this is stored within the modules upon exporting for use as IP core.

6.4.3.2 u-dma-buf

The *u-dma-buf* module is designed to allocate contiguous physical memory blocks in the kernel space for use as DMA buffers and provide access from the userspace [125]. These blocks may be used as DMA buffers when a user application interfaces with UIO mapped IP, such as a DMA Controller in the PL for streaming data. We use *u-dma-buf* to cache PR bitstreams, preparing reconfiguration bitstreams in contiguous memory buffers for rapid loading into the PL.

6.4.4 Xilinx AXI DMA

We use an open source userspace-accessible module for wrapping Xilinx's DMA controller kernel driver, enabling access to both the DMA and Video DMA IP cores [126]. It allows for zero-copy, high-bandwidth DMA transfers between the PS and PL allowing data to be moved rapidly between either system. This wrapper driver support transmit, receive as well as two-way DMA transactions between the PS and PL. Users can use this module as well as *u-dma-buf* to create contiguous physical memory blocks mapped to the userspace to transfer

their application data into and out of the PL. We use this driver for both provisioning the ICAP as well as moving data into user accelerators. This is used in part with the UIO driver to control the AXI-Stream bus switches which may be shared between ICAP and n number of user accelerators interfaces. The driver supports both synchronous and asynchronous transfer modes, allowing for callbacks to be registered against the completion of asynchronous transfer. Asynchronous transfers are used by the runtime enable high performance and non-blocking reconfiguration of the PL. The original wrapper driver does not support Linux kernel releases greater than 4, so we use a modified version [127] of the driver that is compatible with both the Zynq and ZynqMP and has been tested in PetaLinux 2019 (Linux Kernel 4.19.0). The modifications includes kernel API changes to match the changes found in the 4.19.0 kernel as well as changes to Xilinx DMA driver. Xilinx has since published documentation on how to wrap their mainline driver for userspace control, we intend to implement this to stay in better sync with Xilinx's own changes. The modified driver has been tested against the 5.x kernel and supports PetaLinux 2020.

6.5 ICAP DMA Provisioning

To provide high performance PR of the PL, we leverage DMA provisioning of the ICAP. Previous academic work demonstrated a management platform for improving the performance of partial reconfiguration via a high throughput direct memory access to the ICAPE hardware macro [11]. The tool builds upon this by providing the missing Linux controller for this interface, using an open source DMA driver as well as physical to virtual memory mapping driver to enable the tool to be controlled entirely from the userspace. Full and PR bitstreams can be stored at image build time as well as added to the system at runtime. Additionally the mechanism for provisioning provides a non-blocking software routine that can raise an interrupt on completion, freeing the PS while PR is ongoing and the PL is ready to receive data. This can be done by either modifying existing or creating custom configuration files as the ZyCAP system abstracts the hardware control. Given the performance advancements of the Zynq Ultrascale+, we are able to clock the hardware controller for ICAPE3 at 200 MHz, which results in a throughput of 757.2 MiB/s as the DMA transaction approaches transfer saturation of the AXI4 bus (800 MiB/s).

6.6 Configuration Manager

The runtime or configuration manager (CM), enables the designer to abstract control of the hardware modes and configurations; a key feature of the framework. Fig. 6.3 provides an example sequence diagram of the API calls made

by a user's application to the runtime. Rather than requiring the user to know which bitstreams contain which selection of modes and configurations as well as the location of the target bitstream files, the runtime can apply these changes by passing it just the name of the configuration. Referring to fig.6.3, the CM provides an abstracted means of preparing contiguous memory buffers via *u-dma-buf* (*A*), managing PL MMIO addresses (*C*), transferring streamed data into the PL by selecting the desired AXI Switch channels (*D*) as well as handling provisioning of PR and static bitstreams into the FPGA (*B*).

In *A*, the CM checks to see that the required kernel drives exist and initialises contiguous memory buffers to cache target bitstreams with the *u-dma-buf* driver. This is performed to enable rapid loading of these buffers into the DMA controller in the PL and thus triggering of reconfiguration of the FPGA. The number of bitstreams to be cached can be configured at build time or at runtime, to manage memory usage. *B* highlights the CM passing a bitstream buffer pointer to the DMA driver which triggers a DMA transfer into the PL. The CM ensures that the AXI-Stream Switch is set to the ICAPE macro and starts the DMA transaction. The loaded bitstream may also be read out from the ICAPE using this same method. *C* demonstrates how the CM abstracts the addressing of memory mapped PL peripherals (AXI & AXI Lite). The user is not required to track these PL memory addresses, they simply need to load the required mode and the CM ensures that the correct registers are populated. When building modes from HLS generated IP cores, the build tool is able to generate hooks for the internal registers and provide granular access; without this, the designer must manually specify the address spaces. In *D*, the CM checks which AXI-Stream path is required for the user's transfer and changes the transfer path such that it points at the target accelerator. The AXI DMA driver allow for single direction transfers as well as bidirectional transfers both of which can be set to trigger on an interrupt from the PL, providing non-blocking transfers to the PL. At release we provide programmatic access to the CM using the C++ API but intend to expose it generically as Linux service.

6.6.1 Runtime API

We provide a lightweight C++ API for controlling and provisioning modes and configurations between the PS and PL. This API provides abstractions to the configurations and modes as well as manages the data flow between an application and the PL. We use `z` for the ZyCAP2 Manager and `s` for the configuration.

- `Zycap z(hardware,configs)` – ZyCAP constructor takes overrides for default bitstream and configuration directories.

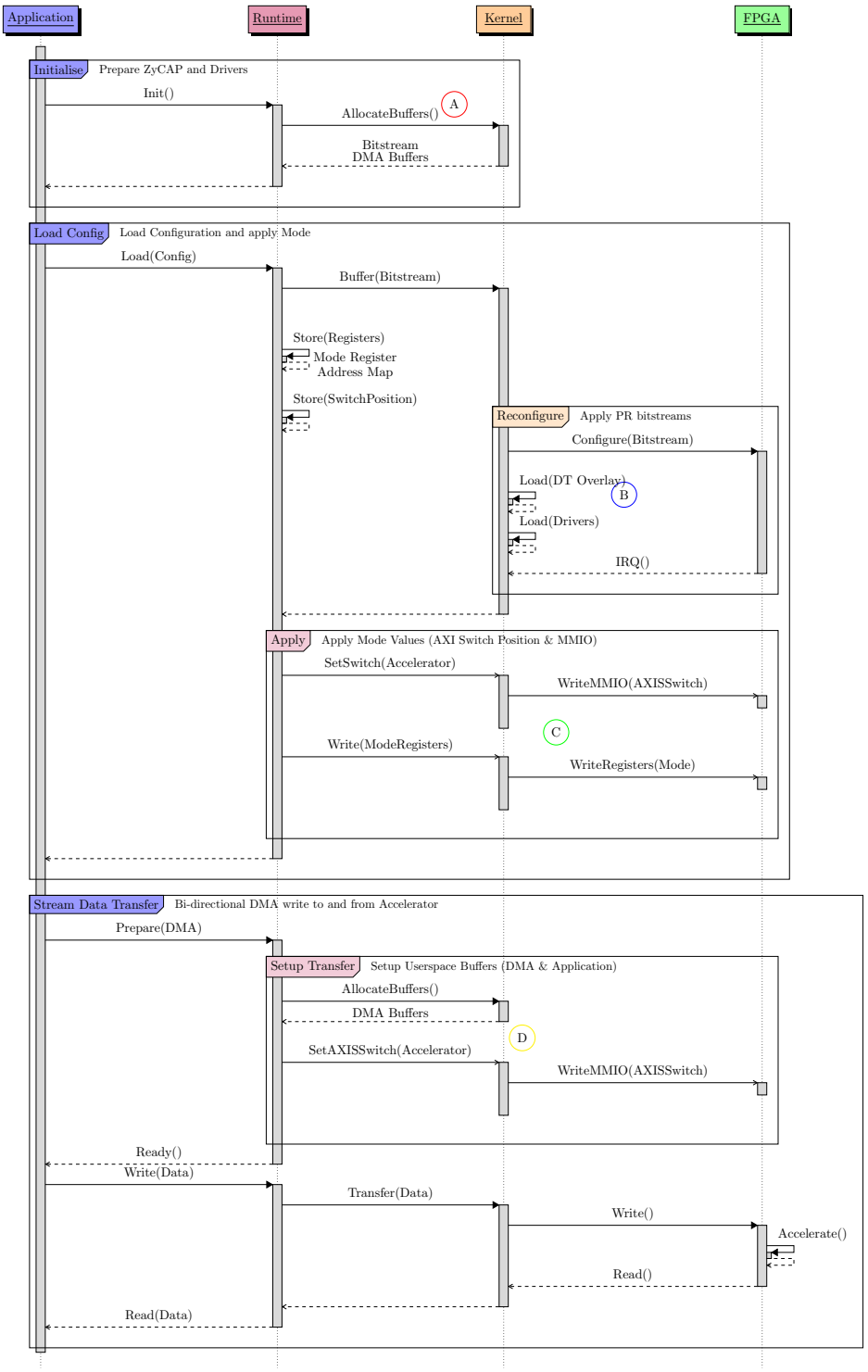


Figure 6.3: Sequence diagram for the ZyCAP Runtime (Loading and Data transfer). (A) Setup of the ZyCAP and driver. (B) Application of PR bitstream. (C) Application of PR modes. (D) Data transfer between accelerator and software application.

- `z.init(config)` – load a default configuration into the PL. This uses the FPGA manager driver as the initialising bitstreams must be loaded over PCAP before the ICAP can be used.
- `z.status()` – returns a struct containing the status of the PL including the currently loaded `string:config`, `string:mode`, and `bool:pl_busy`.
- `z.configs()` – returns the available configurations in the default location.
- `z.config(config)` – load a configuration into the PL.
- `z.alloc(size, name)` – allocates contiguous memory for accelerator use.
- `z.exit()` – cleanly tears down the ZyCAP runtime.
- `s.modes()` – returns the available modes within the configuration.
- `s.mode(mode)` – load a mode into a configuration.
- `s.write(reg)` – read from a register specified in the mode.
- `s.read(reg)` – write to a register specified in the mode.
- `s.transfer(buffer, type, direction)` – read/write a buffer in PS to a configuration in the PL either via DMA stream or a memory transfer. `enum:type` may either be `dma` or `axi`. `enum:direction` may also be a `two_way_transfer`, which sends data from the PS to the PL and waits for the PL to write back into the PS.

The runtime API is designed to predominately use the pre-generated JSON configuration objects built by the tools in the build workflow. This allows the user to manually tweak the configurations to suit their applications as well as easily group behaviours such as associated bitstreams, MMIO maps and values.

6.7 Evaluation

We evaluate the E2E build and runtime tooling in terms of both runtime performance and build complexity. It is important that the provided abstraction has minimal impact on both user accelerator and software performance. The evaluation is performed on the Ultra96v2 development kit and an Intel i7-10750H for build tooling as described in Section 5.10.

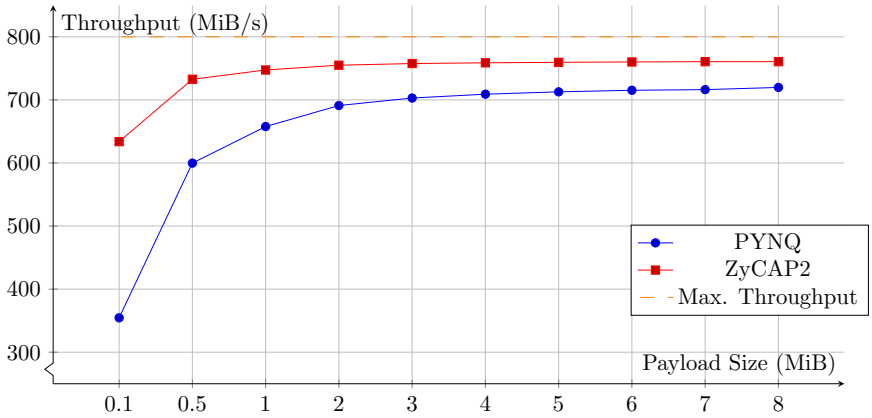


Figure 6.4: DMA Driver Benchmark across 1000 transfers (PL clocked at 200 MHz)

6.7.1 Accelerator Performance

To evaluate the impact of the tool's custom generated infrastructure, we quantify the AXI and AXI-Stream transfer performance. While this evaluation is indicative of the performance of the userspace DMA driver, we show how there is no hardware performance penalty when using the framework and tools. The tools do not add any additional infrastructure overhead to the Xilinx interconnect (AXI and AXI-Stream) IP cores and as such performance is only limited by the these cores. We provide a benchmark of the userspace DMA driver transfers compared against Xilinx's own driver running under their PYNQ platform. This demonstration is performed across varying sized payloads, where the PL is clocked at 200 MHz using a 32-bit AXI-Stream bus, with maximum burst size set to 256 bits, where the theoretical maximum throughput is expected to be 800 MB/s (approx 763 MiB/s). The drop in throughput for smaller bitstreams (less than 1 MiB) is amortised in larger bitstream as the DMA stream saturates.

Across all the demonstrated transfers, the driver used within ZyCAP2 consistently performs at higher throughput than the equivalent transfer in PYNQ, as shown in fig. 6.4, approaching the theoretical maximum throughput while providing a level of abstraction for controlling accelerators without compromising on performance. We assume that this discrepancy is due to the Python function calls adding a non-negligible overhead to the performance of the DMA driver.

6.7.1.1 Runtime Latency

We evaluate the trigger latency for each stage of the reconfiguration runtime called from within Linux. Trigger latency is defined as the time taken for an API call to the CM and may be cumulative if called multiple times for a complex

Table 6.2: Runtime Latency Breakdown Average Across 25 Runs.

Software Layer	Latency (ms)
Set up CSU ●	5.58
Initialise drivers ●	4.87
Allocate buffers ◐	3.18
Parse JSON ◐	2.26
Load config (bitstream only) ◐	0.21
Load config (bitstream + MMIO) ◐	1.23

● : Performed once ◐ : May be performed multiple (per config).

configuration, such as with a mode that contains multiple MMIO reads/writes. Table 6.2 provides a breakdown for the software overhead (measured as latency) required by the runtime in order to load and trigger configurations and modes. This specifically highlight loading bitstreams (as well as a combined MMIO write) as this suffers the greatest impact on overhead from the API. It is important to evaluate this as these software calls might be expected to be performed during an asynchronous transfer of PR bitstream to the PL and thus should be minimal such as not to impact the total time to perform a provision of a configuration. Notable the parsing of configuration JSON files adds non-significant latency to the actual configuration of the bitstreams, given that they can be performed asynchronously and at higher throughput that other tools. We argue that given the significant increase in time to load bitstreams versus FPGA Manager, that this abstraction is justified to reduce complexity for tracing bitstreams and managing configurations. Certain aspects of the tool may be performed at the boot time of the device, such as initialising generic userspace drivers, however these are included in the table as it could be assumed that they are not loaded until the runtime starts. Buffer generation is measured with a 5 MiB contiguous block of memory allocated for the user's accelerator in the PL. This buffer generation is used for both accelerator and PR loading. It's important to note that time to parse JSON objects scales depending on the complexity of the configurations and modes.

6.7.2 Partial Reconfiguration Performance

We demonstrate the PR controller on the FPGA by clocking it at 200 MHz and provide a benchmark comparing the runtime loading bitstreams into the PL against Xilinx's FPGA manager runtime. Theoretically while the ICAP may be clocked at higher frequencies [66], beyond the recommended specification, we demonstrate it in the context of this case study, where all the IP cores are also clocked at 200 MHz and share the same DMA controller as the ZyCAP PR controller.

6.7.2.1 Comparison to FPGA Manager

We evaluate the performance against Xilinx's provided FPGA manager tool as shown in 6.5. As previously discussed in [14], the FPGA manager tool is verbose so both a default (verbose) and silent version are compared against the runtime. In [14] we showed the performance of the ICAPE3 running at 100 MHz; we are now able to show the ICAPE3 macro at an increased clock of 200 MHz [128], where performance is significantly improved against the traditional PCAP reconfiguration flow. The runtime is benchmarked at 100 MHz and 200 MHz against Xilinx's FPGA manager in default and silent modes, using 3 varying sized bitstreams (5.430 MiB, 2.565 MiB and 1.330 MiB). The bitstreams were generated by varying the size of the assigned RP. Timings for the asynchronous calls to the DMA engine from the ZyCAP runtime are provided, which while not a true measure of the performance, give insight into the earliest availability of the processor after reconfiguration is triggered. This is a fixed triggering overhead of approximately **33 us** for a bitstream of any size, where an interrupt handler will fire when the DMA transaction completes. During this this time the processor is free to begin applying the device tree fragments, establishing any accelerator specific buffers, etc. The results demonstrate that the runtime has a significant advantage over the FPGA Manager, increasingly so as the sizes of the bitstreams increase and the time to trigger the DMA transaction is amortised in the time for transfer. Increasing the frequency of the ICAP from a base clock of 100 MHz to 200 MHz resulted in an increased throughput of **94.8%** (from 388.7 MiB/s to 757.3 MiB/s) demonstrated when compared against the PL clocked at 100 MHz (measured using the 5.430 MiB bitstream).

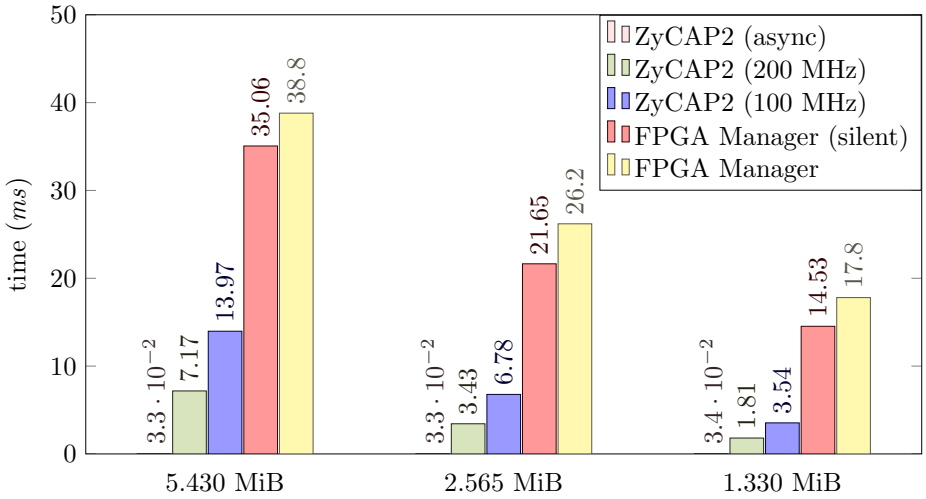


Figure 6.5: PR Runtime Performance (time to load bitstream)

6.8 Case Study

We demonstrate the build tool and runtime using an HLS Vitis Vision accelerated image processing application case study that uses a USB webcam (Logitech C920) attached to the Ultra96v2's processing system. We show how the simple runtime C++ API can be used to control the FPGA, through loading configurations and updating modes, demonstrating how PR and MMIO are managed.

The example design uses 3 PRRs with 3 independent configurations; an initial histogram computation followed by two image manipulations accelerators, which can be either pass-through, a gaussian filter, chroma key filter, gamma correction and/or histogram equalisation. These are connected to the PS using a combination of AXI Lite and AXI-Stream interfaces. The acceleration modules are generated from Xilinx's Vitis Vision HLS libraries [129] which offer OpenCV function acceleration for FPGAs. Using static reconfiguration and the FPGA Manager (limited to 256 MB/s), bitstream loading would occupy significant percentage of time sampling from the camera interface, resulting in dropped frames. The case study provides a demonstration of how accelerator chaining is relevant to image/video processing applications.

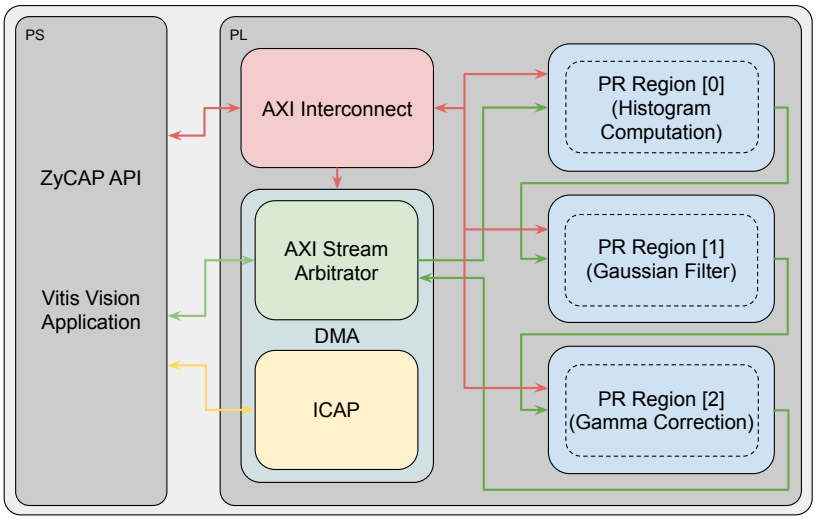


Figure 6.6: Overview of HLS Vitis Vision chained accelerator demo

Interfaces extracted during the build flow are used to instruct the tools which interfaces on the accelerators should be connected. Fig. 6.6 highlights chaining accelerators using the tools. Using a traditional shell based PRR, data must first be sent to the PS to be redirected back into another shell containing the next function, unless complex bus logic for interconnection has been implemented. This case study also highlights the significance of high performance PR, allowing us to rapidly modify regions 0, 1 and 2 without loss

of frames.

A histogram computation is performed in the first accelerator core, and sent to the software application (via MMIO AXI) to then sequence the chained configurations such that the image passing through each accelerator is processed to improve the quality of the output image. Reconfiguring rapidly during the image stream to provide real time improvements without dropping frames. The thresholds for the histogram computation can be adjusted in the software application to be more or less aggressive with attempts to improve image quality. While the demonstration shows the use of just a few acceleration cores, additional regions could be used to chain further image manipulation such as resizing or scaling.

6.8.1 PR Region Data Chaining

To configure chaining, the designer can set the following parameters as shown in listing 6.1.

Listing 6.1 shows how the PR chain is configured with a JSON array. The order in which the regions are referenced, refers to how they will be connected, where `region_a` is the first region of the chain (connected to the output of the PS DMA controller) and `region_c` is the last region (connected to the input of the PS DMA controller). Fig. 6.7 shows the process of the PS application using the PL accelerated histogram to make config/mode decisions based upon the current image in the accelerator chain. The PS application is shown to apply a gaussian filter to region 1 of the PL and then a stream pass through for region 2, upon deciding the histogram data is acceptable.

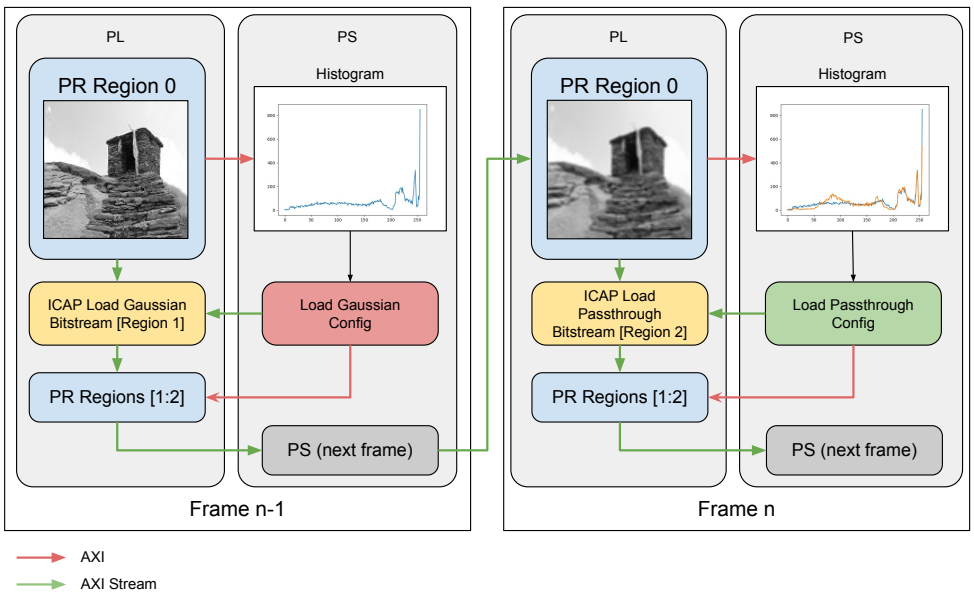


Figure 6.7: PS uses histogram to determine accelerators to apply. Blue graph indicates original histogram, Orange indicates the new histogram after configuration.

```
{
    "pr_regions": {
        "region_a": { ... },
        "region_b": { ... },
        "region_c": { ... }
    },
    "pr_chains": [
        "region_a",
        "region_b",
        "region_c"
    ]
}
```

Listing 6.1: Enable PRR chaining

Given this can be used with both configurations and modes, the software can intelligently determine to load new bitstreams or make adjustments to the currently loaded accelerators (such as applying a mode).

6.8.2 Design Process

In order to manually develop the application described in the case study, the designer would have to undertake a series of manual steps, highlighted in. The designing the accelerators themselves (using RTL, HLS or otherwise) is beyond the scope of this framework as plenty of academic and commercial work already provide tools to do this. High level synthesis tools are used to reduce the complexity of RTL design but without aid from automation, the complexity of the vendor tooling provides a high barrier to entry for novice designers. Figure 5.4 describes the workflow that is automated by the tools, where the nodes in the flow diagram, represent each of the steps that would need to be manually performed by a designer. This diagram assumes there are no mistakes or errors on the designer's behalf.

Figure 5.4 is a high level overview of the complexity; in reality there are significantly more steps involved in the development process such as tweaking board settings, correcting for mistakes and manually verifying compatibility across modules and regions. Some of the underlying steps such as parsing the interfaces from modules can be quantified, however the time taken to extract interfaces and generate custom build infrastructure is amortised by the time taken for the tools to perform synthesis and place and route. For the case study, the tools added **37.94** seconds of overhead to extract interfaces compared to the **2427** minute implementation run within Vivado. It is difficult to more generally quantify the impact of the tools on reducing design time complexity, however given that the user is only required to define a specification file, to generate a ready-to-go Linux image, this can be considered a significant advantage.

6.8.3 Comparison to Existing Tools

The case study demonstration shows the webcam running at a resolution of 1080 x 1920 pixels at 30 frames per second. For a pBlock sized to support the largest of configurations, the chroma key function, the runtime is able to perform partial reconfiguration in 1.808 ms for a 1.330 MiB bitstream. For a camera producing a new frame every 33.3 ms, reconfiguration must be performed at least within this time frame and should factor additional software overhead to ensure frames are not dropped. Comparing this to FPGA manager, the same bitstream was loaded in 17.8 ms. While this is acceptable, given the low framerate of the USB camera only one or two frames might be dropped, higher performance systems such as those used in critical safety systems for autonomous vehicles [130] or with complex PL logic demanding larger pBlocks (thus larger bitstreams), begin to become constrained by time to reconfigure under FPGA Manager. This scenario could be envisioned with the use of a high data-rate MIPI-based camera connected directly to the PL. At present this is not achievable with the target Ultra96v2 board as there were no available MIPI camera modules but could be demonstrated on another device in future work.

6.8.4 Runtime Application

The code in listing 6.2 demonstrates the abstraction used to load a configuration and then apply a subsequent mode. The JSON config files are generated from the build process and can be later modified by the user to add additional modes, as shown by the `full_hd` mode in listing 6.3. In the case study, the mode `full_hd` is used to set MMIO registers in the accelerator module corresponding to the resolution of the frames being transferred between the PS and PL.

```
#include <zycap.h>

void main()
{
   string hardware = "/lib/firmware/";
   string configs = "/lib/configs/";

   Zycap z(hardware, configs);

   /* `load` writes to the AXI-Stream Switch to set the
    multiplexed outputs as well as writes the default MMIO
    register values */
   cout << z.configs() << endl;
   Config s = z.config("gaussian");

   /* `mode` writes to MMIO registers with any custom
    settings specified by the designer in the generated JSON
    objects. */
   cout << s.modes() << endl;
   s.mode("full_hd");
}
```

Listing 6.2: C++ API

```
 "slug": "gaussian",
 "type": "partial",
 "regions": [
     {
         "name": "gaussian_a",
         "bitstream": "gaussian_a.bin",
         "overlay": "gaussian.dtbo",
         "interfaces": {
             "axi_stream": "0x1",
             "axi_mmio": "0xa0050000"
         },
         "modes": {
             "default": {
                 "0x40": "0x438",
                 "0x44": "0x780"
             },
             "full_hd": {
                 "0x40": "0x1E0",
                 "0x44": "0x280"
             }
         }
     }
 ]
```

Listing 6.3: JSON Config & Mode

The `full_hd` mode used in this snippet was added post-build by the designer,

the default configuration JSON object was produced by the tools and extendable by the user before the Linux image is compiled. Typically configuration files injected into the Linux filesystem along with PR bitstreams at compilation time but can also be added at runtime, where the runtime manager can be instructed to locate and detect new configuration files. This is configuration is made available to the runtime C++ API, requiring minimal understanding of the mechanisms required for provisioning configurations and modes.

6.9 Summary

This Chapter, presented a custom high performance runtime configuration manager that utilised the hardened ICAP macro on Zynq and Zynq Ultrascale+ architectures for near theoretical throughput provisioning of PR bitstreams into the PL (388 MiB/s and 757 MiB/s for the Zynq and ZynqMP respectively). The runtime managed both PR and hardware control under abstractions for configurations and modes. The runtime's performance was compared against Xilinx's own FPGA Manager driver as used by other academic PR runtimes. It was shown how benefits included performance gains as well as application advantages including the ability to trigger a non-blocking DMA PR event that frees the processor to perform device tree overlay provisioning and memory buffer initialisation/de-initialisation while awaiting a PR completion interrupt. It was demonstrated how the runtime could be used to accelerate a vision processing application that used the build tools (in Chapter 5) to generate a design for HLS-based accelerators chained together for tuning image quality. This case study highlighted unique architectural features including the ability to chain PRMs together (discussed in Chapter 5) for increased operation during streaming events. The following Chapter 7 builds upon the runtime by extending the abstraction to autonomous and cyber physical systems, introducing a runtime for popular autonomous system platforms.

The PR runtime is available at `http://github.com/warclab/zycap2` as open source repository to encourage wider adoption and contribution by the community.

Chapter 7

Autonomous Adaptive Systems Framework using Partial Reconfiguration

7.1 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 [131], communication systems [132] and self-driving vehicles [133, 134] 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 [135]. 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 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 [136], fixed ASIC accelerators become problematic due to their lack of flexibility. FPGAs offer application specific hardware acceleration while maintaining computational flexibility through reconfiguration.

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 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 [137]. While extensive contributions have been made in autonomous software frameworks for CPSs [138], limited research has considered their coupling with hardware acceleration. In this Chapter, FPGA SoCs, such as the Xilinx Zynq and Zynq Ultrascale+ SoCs, are considered as platforms for implementing such hybrid autonomous systems along with high level software abstractions to manage hardware.

The work presented in this Chapter has also been discussed in:

- Alex R. Bucknall and Suhaib A. Fahmy. Runtime abstraction for autonomous adaptive systems on reconfigurable hardware. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 1616–1621, 2021. doi: 10.23919/DATE51398.2021.9474199 [17]

7.2 Contributions

The key contributions of this Chapter are:

- An abstracted CPS configuration manager, built around an adaptive systems model to automate reconfigurable hardware management and allow control from a PubSub architecture.
- An extension to partial reconfiguration design tools to generate PR bitstreams, software drivers and abstract symbols based on CPS specifications.
- A ZeroMQ middleware (written in C++) wrapper for the configuration manager for the Robot Operating System (ROS)
- A case study using ROS that demonstrates using a PubSub architecture to control FPGA configurations within image processing applications.

7.3 Related Work

CPSs have gained much interest in literature over the years, with various definitions of hierarchy and structure; this Section defines relevant concepts and terminology. [139] provides a recent survey of FPGA-base computing within robotic, citing a number of publications across sensing, mapping and machine learning. Within Networking & Mapping, [140] shows a colour tracking demonstration on a Xilinx FPGA and [141] provides a case study of an application that generates a map and executes an extended A* algorithm to plan the path. The device can then localizes itself in the environment containing multiple obstacles and a target destination.

7.3.1 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* [142]. Observation is defined as the (post) processing of sensor data, such as feature extraction from an image processing event and can be considered 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. 7.1. The model in [143] 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 [144].

In [138], 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 [145], which provides a useful model for conceptualising how such nodes interact. Frameworks such as ROS [146] build upon these concepts to provide a structured communication layer that supports heterogeneous clusters. Inspiration from the ROS operating model was taken for the configuration manager.

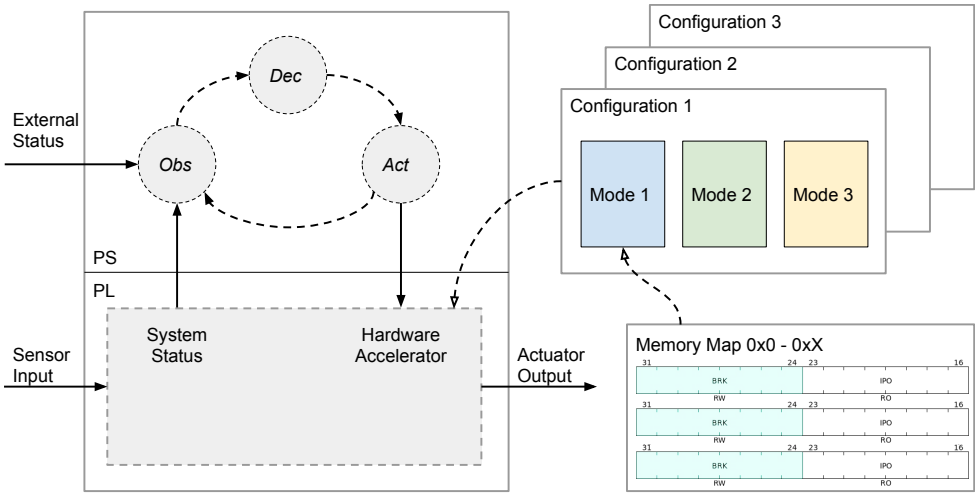


Figure 7.1: Visualisation of hardware abstraction and definition.

7.3.2 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 [147], which introduced the Data Distribution Service (DDS) and added features such as quality of service, security improvements, flexibility, and robustness.

ROS2 is designed to offer a standard software platform to enable developers to rapidly prototype and then deploy their robot applications against a common set of libraries and communication protocols. ROS2 provides a number of advantages including flexibility to be used alongside other application due to inclusivity to run as an application on top of an operating system rather than as the entire operating system, unlike some RTOS platforms. It is open source and highly documented along with a large user base of existing tools and libraries making it easy to design and implement on top of. ROS2 has been used in scientific robotics projects including NASA's Viper Rover [148] as well as in consumer products such as iRobot's autonomous vacuum cleaners [149].

ROS2 is based around a publisher/subscriber (PubSub) protocol where a number of entities exist:

- *Nodes*: A node is a ROS entity that participates in the network either by creating or consuming data. Nodes can also provide or use services and actions and may be configured via a set of parameters.
- *Messages*: ROS data types used when subscribing or publishing data to a topic.

- *Topics*: Topics are used a common channels that Nodes can use to publish or subscribe to messages.
- *Discovery*: Discovery is a mechanism that automates process through which Nodes can discover each other and determine how to communicate.

More recently works such as [150] have provided augmentation to ROS2 that enables control over FPGAs that are encapsulated by Nodes. They present a custom architecture that allows for high level algorithms to be implemented as hardware modules within the PL. Their modulate design is intended to ease adaptability of such changes within the system; they use a generic interfaces for all modules for fully customizable messages. This allows for data exchange internally as well as with external parts of a distributed system.

Within the context of a CPS, decision and action could be viewed as software driven events given the complexity with considering the evaluation of the observation state. The observation may rely on stimulus from external sources or data produced by sensors and/or peripherals, attached directly to the PS or the PL and further accelerated in the PL.

7.3.3 Configuration Terminology

Terminology is defined 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). These are referred to 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).

7.3.4 FPGA Acceleration of Adaptive Systems

Formal adaptive system frameworks are defined in the literature [151] 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 [144]. 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 [10] and increasing performance [108], rather than control abstractions. Work such as [96] 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. The Xilinx toolchain is considered 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.

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

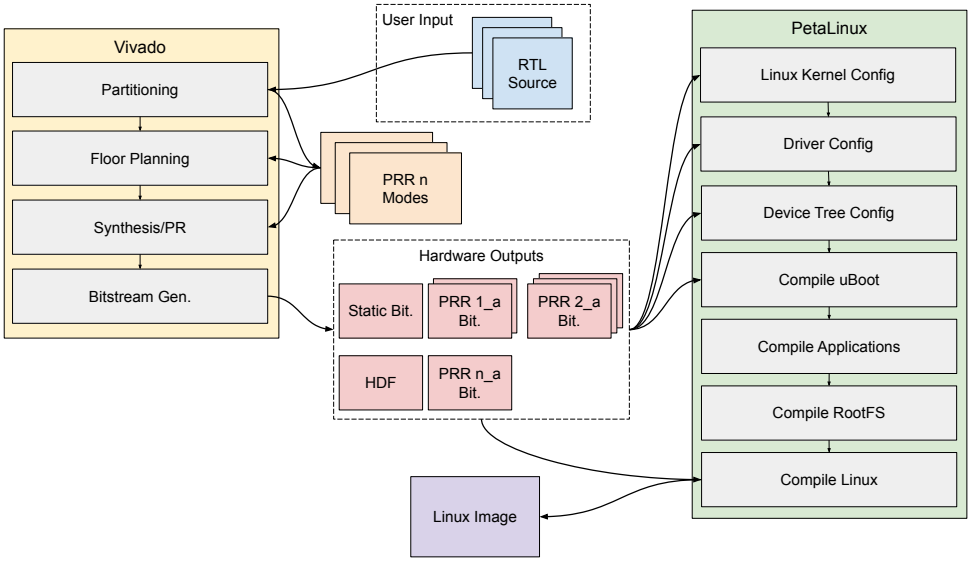


Figure 7.2: An outline of the Xilinx PR build flow, from generating hardware within Vivado, to exporting hardware data into PetaLinux.

toolflow, outlined in Fig. 7.2, 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 [103]. 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 [58] 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 [61] 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 [50] 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 [10] 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.

7.4 Architecture

An abstracted configuration management runtime was designed, 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 configuration management runtime is deployable against a Linux operating system on Xilinx Zynq and ZynqMP device types. This runtime provides a generic API, over a network capable protocol, for software frameworks to interact with hardware acceleration available to the processing system.

Incrementing on the structure of the build JSON, user is presented with a template for each viable configuration, this can then be used to design the user loadable configuration, specifying underlying PR modes, memory maps and registers that can be called from the runtime. This allows the user to deploy configured instances of hardware without concern for the method of deploying to the hardware. The user is insulated from needing to trigger and track PR bitstreams, load the corresponding drivers, manage the addresses of memory maps as well as write to specific memory maps for actions such as configuring PL IP cores. A collection of existing build tools [14] were used, which extend the vendor PR build flow with the inclusion of user defined build schemas, representing the user's desired PR modes and configurations.

7.4.1 Adaptive Hardware Design Tooling

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. The outputs produced by the build tools were used to generate higher level abstraction schemas that are used to load the state of hardware and software according to the configurations applied.

7.4.2 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. A template config schema was generated 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 7.1 shows an example output file from the build tool with two template modes for the camera, default and edge detection. The YAML format was selected as a 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
config:
  - name: camera
  - modules: [default,edge,colorise]
  - type: partial
  - modes:
      edge_detection:
        - modules: [edge]
        - mmio:
          - base_address: 0x2000
          - registers:
            - focus:
                - offset: 0x1010
                - default: 0x01
        - overlay:
          - camera
      default:
        - modules: [default]
        - overlay:
          - camera
```

Listing 7.1: 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.

7.4.3 Configuration Manager

The CM is designed to be available within the Linux userspace, regardless of the programming interface/framework attempting to consume/control it. The ZeroMQ [152] protocol was chosen, a lightweight asynchronous messaging library, used for both internal and external networking. Alternative communication libraries such as OpenDDS [153] could be substituted for ZeroMQ, which was selected for its lighter-weight implementation, support for multiple transport methods and programming languages as well as lower latency.

Shared memory via Linux's inter-process communication (IPC) transport was used for its low latency, however the CM can also use external IP transport methods, such as TCP. IPC and TCP layers are later compared in Subsection 7.5.2. ZeroMQ is a 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 [154] as the 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.

A userspace-to-kernel drivers is used 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 the Userspace IO (UIO) kernel module is used, which allows for the mapping of configuration memory addresses and interrupts within the PL up to the PS. For streaming data between hardware, a zero-copy kernel driver and userspace wrapper for the Xilinx AXI DMA interface [126] (AXIDMA) is used. 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, the 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 a custom PR manager from [14] was used, 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.

7.4.4 Configuration API

An API is exposed on a selection of topics, available over ZeroMQ. As ZeroMQ is language agnostic, this could be written in a number of languages such as Python or C++. A sample list of topics used in the framework is shown below, along with a Python example of how it can be used in Listing 7.2.

- `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.

Listing 7.2 provides a reduced and simplified version of the fundamental model of autonomous adaptive system loop [143]. The listing assumes that the internal networking between the CM and the described Python model has already been established. Line 1. describes the action function which takes the decision result and triggers a ZeroMQ publish event based upon the contents of the payload, which would be received by the CM and load an edge detection module into the FPGA. Within this function, the CM can trigger reconfiguration (partial reconfiguration), write to an internal register (parametric reconfiguration) or do nothing. Line 13. is a generalised loop for calling the observation and decision functions, where the logic of these functions can be assumed as taking place elsewhere within the application.

```
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':0x2}")
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 7.2: Pseudo example of congitive engine controlling the CM.

7.5 Demonstration

A demonstration of the CM was provided by implementing it alongside the ROS2 framework to highlight the minimal effect on latency and performance introduced with the abstraction. This demonstrates streaming image/video data to/from hardware to a lightweight software cognitive engine running upon a CPS. A series of experiments were performed to quantify the overhead of moving data between an ROS2 node and a hardware accelerator as well as examining the ZeroMQ protocol.

7.5.1 ROS2 Architecture

In ROS2, networking is built on top the pubsub model akin to ZeroMQ, as used in the CM, shown in Fig. 7.3.

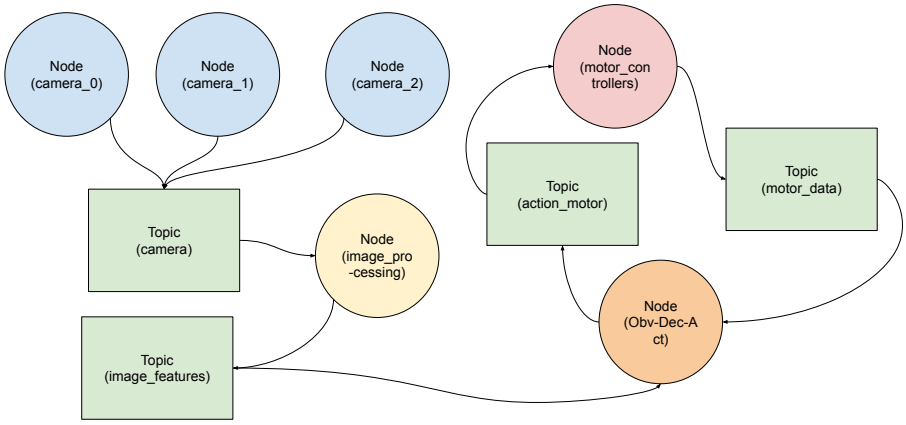


Figure 7.3: Simplified example of a potential unmanned aerial vehicle ROS2 application, where camera data could be used for object avoidance

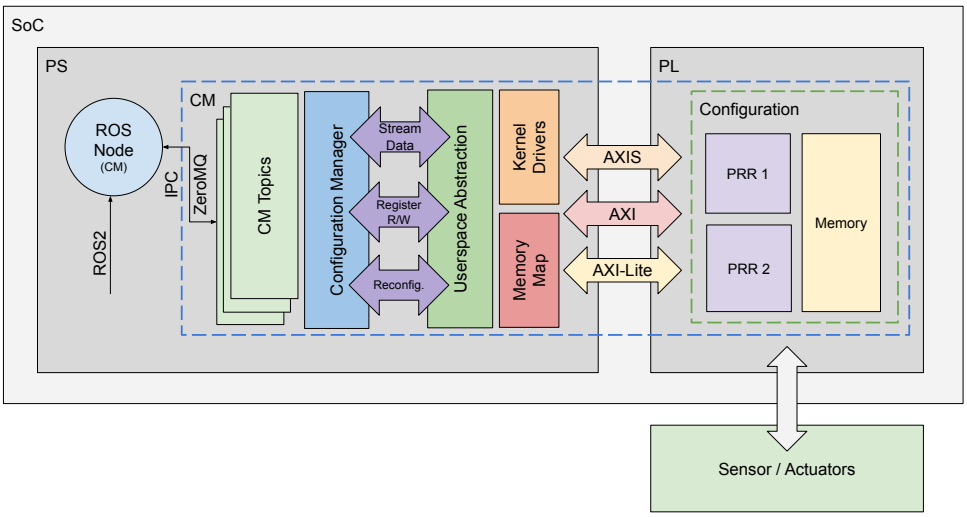


Figure 7.4: Architecture of the ROS2 wrapper using the CM and ZeroMQ.

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.

Messages can contain a wide variety of data structures, which are built by composing multiple primitive data types. In the example from Figure 7.3, the image data from a camera is published to the *Image Topic*, and received by

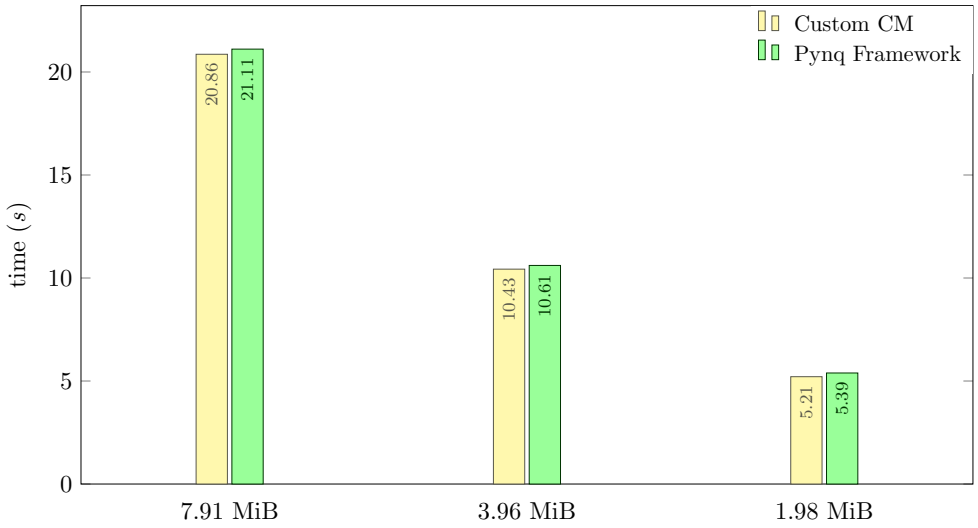


Figure 7.5: Round trip time to transfer 1000 images between PS and PL.

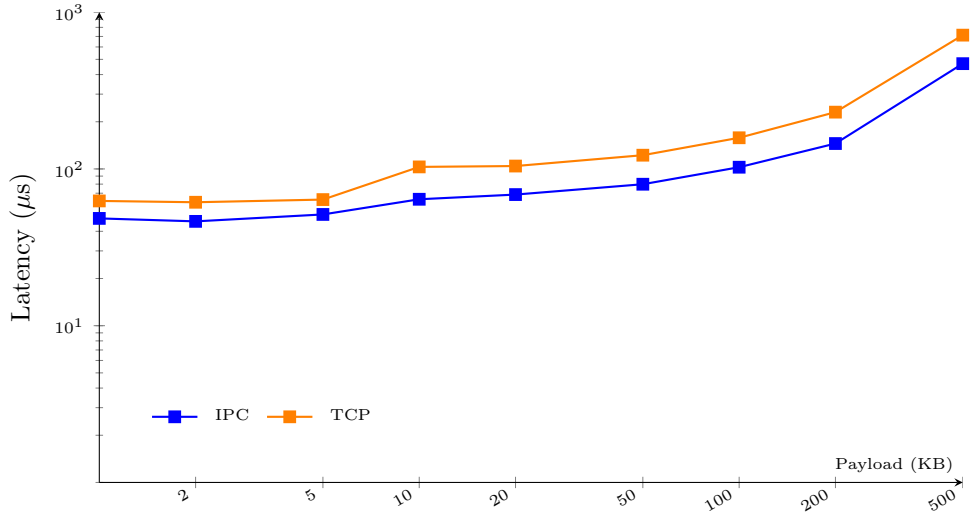


Figure 7.6: ZeroMQ latency measured with varying sized payloads.

three subscribers, which perform image recognition for separate features and publish the outputs to the *Road Features Topic*. The output is then used by a control system node to send appropriate signals to the car's motor system. This model provides modularity by treating each node as a separate entity and is suitable for distributed systems.

7.5.2 Experiment

The ZeroMQ-ROS2 wrapper was used, which maps the CM's ZeroMQ endpoints to a ROS2 node, seen in Fig 7.4. This turns the ROS2 node into a wrapper for the CM's ZeroMQ publishing/subscribing topics. The userspace abstractions provided by the CM were used to expose the PL to the ROS2 node.

This focuses on two main overheads; at the ZeroMQ interface and moving data between the PS and the PL. A PS-PL-PS DMA transaction to quantify the transfer performance between the CM and PL is benchmarked 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 example images (7.91, 3.96 and 1.98 MiB). The same image was used with differing levels of compression to generate the desired payload size. Considering the 3.96 MiB image, max. approximate throughput of 380.04 MiB/s (per direction) is measured, 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 [155]. This is compared to the PYNQ framework, where the same image was transferred in an average of 10.61ms, highlighting how the 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 [14], where the maximum configuration throughput is 398.6 MB/s with an average trigger latency of 7μs.

The ZeroMQ layer was tested under the same experiment using IPC, to simulate data moving between the CM memory buffers and the ROS node. The latency and throughput were measured across 1000 trips with the 3.96 MiB image. An average of 5.36 ms latency and approximate throughput of 1939.5 MiB/s were observed, indicating it does not provide a bottleneck. Differing payloads using ZeroMQ IPC and TCP modes were also compared, to represent local API calls, as seen in Fig. 7.6. These results were evaluated against a comprehensive benchmark of ROS2 overheads, performed on a desktop class machine [156]. 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 the results for an equivalent payload. It can be concluded 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 further reductions could be seen, such as with the use of component latency reduction techniques presented in [157].

7.6 Summary

In this Chapter, an abstracted configuration manager for managing hardware acceleration within an autonomous adaptive system implemented on an FPGA

SoC were been demonstrated. It was shown 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. The CM can be used with frameworks such as ROS2, where the hardware can be abstracted into a networked node and communicated to using a publisher-/subscriber protocol. Within the case study this was demonstrated using the ZeroMQ protocol but could easily be interchanged with other PubSub protocols such as OpenDDS or MQTT. Work was performed to expand 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 proposed future work, there is an intention to release this work as an extension to the ZyCAP2 open source framework for autonomous adaptive systems and evaluate networking the CM across a distributed cluster.

Chapter 8

Conclusion and Future Work

This thesis introduced a build framework and runtime tools to enable higher level abstractions for designing and implementing FPGA-based partially reconfigurable applications on adaptive systems. It has demonstrated how and why FPGAs are suitable for adaptive system applications acceleration, where both high performance and flexibility are required to meet the demands of real world tasks. The work demonstrated across this thesis has provided an alternative mechanism for rapid reconfiguration over the Zynq's networking interface, a fully automated framework for building Linux-based FPGA applications on modern Xilinx architectures (Zynq & ZynqMP) as well a high performance runtime manager to abstract the cross-domain management of hardware and software. The collection of tools reduce the barriers to entry for software-hardware acceleration while maintaining the high performance requirements demanded from the real time applications this abstraction was designed for.

This Chapter concludes the thesis, highlights its contributions, and provides suggestions for areas of further research.

8.1 Summary of Contributions

Across the thesis, we introduced runtime tools for abstracting the application of hardware configurations, provided an over-the-network mechanism for the Zynq-7000 that enabled a low latency trigger for initiating PR as well as developed PubSub configuration manager for popular autonomous systems platforms. Our work also targeted the development and build processes relating to PR applications, where we provided a number of tools to handle the challenges associated with the complex hardware to software abstraction under a Linux operating system. We offered benchmarks, case studies and comparisons of our tools, contrasted against other available academic and vendor alternatives. We showed how we were able to provide abstractions while maintaining the high

level of performance associated with FPGA SoCs.

8.1.1 Network-Enabled FPGA Reconfiguration

Chapter 4 introduced a smart networking approach designed to trigger PR via a network interface on the Xilinx Zynq SoC. This mechanism enables bypassing the traditional processing system based ethernet driver and Linux scheduler to load ethernet frames directly into the programmable logic. This reduces the non-deterministic latency experienced using the networking stack from the processing system, particularly with the system under load. We utilised a method of DMA proxying ethernet packets into the PL by remapping the PS ethernet RX frame buffer of the into a ping-pong FIFO within the PL. Packet headers are extracted using an arbiter architecture that redirects the packets to their final destination; either used to initiate PR from a ZyCAP controller or back to the PS for use within general purpose processing. A case study demonstrated loading cryptographic cores for network enabled acceleration. We benchmark the case study and highlight the latency reduction experienced in our alternative mechanism. We provided further demonstrations for loading the bitstream directly from the network interface by transferring the complete bitstream via ethernet as well as the frame header.

8.1.2 Python Library for SoC Interfaces Extraction and Generation

In Chapter 5, we introduced a tool for connecting user hardware sources files, including HDL, IP cores and HLS, within our PR build tool for compile-time generation of hardware infrastructure. This functionality was expanded into a standalone Python library that enabled the extraction of interfaces and ports from modules that could be grouped as buses, such as AXI (Lite) & Stream Master and Slave ports. This has wider uses as standard buses such as AXI are commonly used with SoC designs to move data between the PS and PL. Automatically extracting interfaces provides build tools with the information required to abstract the infrastructure need to communicate with the user's hardware. In addition to extracting interfaces the tool also generates Verilog wrappers for the underlying hardware, pass-through hardware that are used by our tools to set the static region and create blackboxes for the contained logic, needed for PR. This tool was extracted to function as a standalone library to encourage use with other open source projects such as the FPGA SoC building tool Litex [158]. This library allows designers to not be concerned about the interfaces of a module and simply import it into the PR workflow.

8.1.3 Automated End-to-End PR Development Flow

The performance and flexibility of FPGA SoC platforms has limited wider adoption, in particular advanced features such as PR due to the complexity and existing limitations of vendor PR tool's and their ability to serve non-experts. In Chapter 5, we discussed and explained our end-to-end toolchain designed for automating the process of developing Linux-based PR applications for independent designers, without the need for pre-requisite knowledge, typically required to across the domain complexities. This tool takes the user's hardware sources (HDL, HLS, etc.) along with a specification file provided by the designer then generates PRMs, allocate PRRs and configure the required Vivado runs for generating the appropriate partial and static bitstreams. Further advanced design functionality such as the chaining of AXI Stream interfaces between PRMs is discussed and presented. Build metadata (PR configurations, memory maps, etc.) is then extracted from the FPGA workflow and handled over to a Linux workflow (based upon PetaLinux) where various components required to build the Linux image are stored. Firmware for uboot, including custom PMU firmware, are assembled as required. Generic userspace drivers are included into the design flow and a custom device tree and device tree fragments are created from the Vivado hardware export. The PR runtime manager is included into the Linux builder along with the configuration files and locations needed to manage PR. These are then built as part of the PetaLinux build process and a compressed OS image, including PR bitstreams, PR configurations files, ready-to-go Linux image and runtime management software, and bundled for deployment.

8.1.4 High Performance Runtime PR Manager

Chapter 6 discussed the high performance runtime manager designed to provide near theoretical throughput reconfiguration for the Zynq and Zynq Ultrascale+ device types within Linux. This runtime manager provides a non-blocking DMA interface for loading PR bitstreams into PL as well as moving data to/from hardware acceleration functions. Additionally, this runtime also handles applying configurations and modes as discussed in Chapter 5, where the triggering of the state abstractions may require software operations such as MMIO reads/writes and dynamic loading/unloading of device tree overlays. The runtime is demonstrated with a case study for an image processing pipeline, where frames are streamed in from the PS over DMA (AXI Stream) and a histogram array is sent back over AXI Lite. The processing system then uses this data to determine which accelerator it should load into the next PRR slot of PL in order to best improve the image quality. PR chaining and the rapid reconfiguration of the PL are demonstrated within this application.

Experiments are provided to show how performance compares against the standard vendor tool, FPGA Manager for loading a selection of bitstreams that reflect the case study application.

8.1.5 Abstracted Configuration Manager for CPSs

In Chapter 7, we discussed an abstract model for adaptive cyber physical systems, where we extend existing research models to support hardware acceleration. An configuration manager is introduced that utilises a publisher/subscriber (constructed on top of the ZeroMQ protocol) architecture to abstractly map hardware across a system. The configuration manager is demonstrated with a case study for the Robot Operating System 2, where an image processing application is evaluated for latency, overhead and throughput for both the transferring of an image as well as the time to initiate/trigger a PR event. This demonstrated how the high level of abstraction provided by the configuration manager can be integrated with popular frameworks like ROS2.

8.2 Future Work

The research discussed in this thesis covers automated design flow and an abstracted runtime for PR. Our research has helped to advance the state of abstraction and automation for these systems but we are aware that there are still additional challenges that should be addressed. We have identify a number of areas of interest, where the time limitations of this thesis have pushed these topics to be explored and discussed as future work.

8.2.1 Containerization of vendor tooling within ZyCAP2

Due to the release strategy of Xilinx's Vitis, Vivado and PetaLinux tooling, it can be difficult to develop custom build tools that are able to continue to extend functionality as breaking changes are often introduce with new versions Xilinx's tools. During the undertaking of this thesis, multiple updates of Xilinx's tools caused problems at various stages of the ZyCAP2 building process. Currently, we deal with some of these dependency issues by abstracting the building of Linux images into a PetaLinux Docker container as these tools are open source and can easily be installed within a virtual environment. In order to better isolate dependencies and links within a user's workspace, containers ensure that when the vendor tools are accessed/requested that there are no unexpected setup issues, that may be dependent on the environment of the designer. While this doesn't resolve TCL-based breaking changes introduced by Xilinx, it would give a better mechanism for isolating the version specific problems and allow us to better easily manage our own tools against Xilinx's for compatibility.

8.2.2 Integration of FuseSoC into ZyCAP2 tools

At present the work to extend the FuseSoC library manage to support PR is standalone from the ZyCAP2 toolchain. We would like to integrate this directly with ZyCAP2 to enable building designs directly from FuseSoC cores. While this does not reduce the complexity required to write or design hardware accelerators, it would allow for increased reusability across device types, users environments and within libraries. A built-in library management platform would enable designers to find, acquire and deploy readily available IP cores, similar to selecting a software library.

8.2.3 FuseSoC Vitis HLS Support

Increasing the abstraction and lowering the complexity for developing hardware accelerators is an important aspect to improving the viability of FPGAs as adaptive edge platforms. Our current workflow enables building HLS-based designs from Xilinx's Vitis HLS suite but expects these to be managed, tested and exported manually by the designer. A number of acceleration functions written in HLS already exist as the Vitis Suites for Vision, Graphing, Deep Learning as well as others and already provide examples for creating further more complex functions using these libraries. Given that FuseSoC provides a command line interface for easily creating, testing and building HDL projects and as such would be well suited to manage HLS libraries. Our work to enable PR-based FuseSoC HLS cores would allow this to feed directly into our existing build tools and even further reduce the complexity for non-experts to accelerate their software applications with hardware.

8.2.4 Xilinx DFX Abstract Shell Workflow

Adding support for Xilinx's DFX Abstract Shell [8] workflow would enable an increased modularisation for implementation process of PRMs within Vivado. At present, design iteration with our tools is slowed down due to the requirement to place and route for each of the configurations of PRMs within the locked static regions. This is the same design time requirement as Xilinx's standard DFX workflow where multiple passes with the implementation tools are needed and makes the generation of new PRMs tedious and memory intensive as the static region must be kept in memory. The DFX Abstract Shell process allows the implementation requirement of the static region to be separated from the PRM using a trimmed down version of the static design based upon a given RP. If added to our workflow, this would mean that storing the complete static logic would not be required to build new partial bitstreams. Support for the Abstract Shell workflow would also drastically reduce implementation

time across our tools and make it easier implement PRMs after initial build compilation.

8.3 Summary

This thesis has contributed towards reducing the barrier to entry for designing and developing partially reconfigurable applications on edge FPGA SoCs. We have identified and demonstrated resolutions for a selection of issues that introduce complexity, performance and abstraction limitations as well as automation of existing tools We have placed particular focus on ensuring that the abstractions we introduce do not impact performance, where our runtime minimizes latency while approaching the maximum theoretical reconfiguration throughput for Xilinx Zynq and ZynqMP devices. We have released a number of open source tools to enable the FPGA community to better leverage the contributions described in this thesis as well as offered contributions towards already popular open source projects to encourage adoption.

Appendix A

Code Snippets

```
name: fusesoc:examples:blinky:1.0.0
filesets:
  rtl:
    files:
      - rtl/blinky.v
      - rtl/macros.v:
          is_include_file: true
    file_type: VerilogSource
  tb:
    files:
      - tb/blinky_tb.v
    file_type: VerilogSource
  zynqmp:
    files:
      - data/zynqmp.xdc: {file_type: xdc}
targets:
  default: &default
    filesets:
      - rtl
    toplevel: blinky
    parameters:
      - clk_freq_hz
  sim:
    <<: *default
    default_tool: icarus
    filesets_append:
      - tb
    toplevel: blinky_tb
    tools:
      - icarus
      - modelsim
    parameters:
      - pulses=10
  build:
    <<: *default
    default_tool: vivado
    filesets_append:
      - zynqmp
    tools:
      vivado:
        part: xczu3eg-sbva484-1-i
    parameters:
      - clk_freq_hz=100000000
```

Listing A.1: An example YAML file for a blinky FuseSoC core on the Ultra96v2

Bibliography

[1] Ronak Bajaj and Suhaib Fahmy. Mapping for maximum performance on FPGA DSP blocks. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 35:1–1, 01 2015. doi: 10.1109/TCAD.2015.2474363.

[2] *UG574: UltraScale Architecture Configurable Logic Block*. Xilinx Inc., Nov. 2017. v1.5.

[3] Khoa Dang Pham, Edson Horta, and Dirk Koch. BITMAN: A tool and API for FPGA bitstream manipulations. In *Design, Automation & Test in Europe Conf. & Exhibition (DATE), 2017*, pages 894–897. IEEE, 2017.

[4] *UG470: 7 Series FPGAs Configuration User Guide*. Xilinx Inc., Aug. 2018. v1.13.1.

[5] *UG953: 7 Series FPGA and Zynq-7000 SoC Libraries Guide*. Xilinx Inc., Oct. 2021. v2021.2.

[6] Achronix. Embedded FPGA – the ultimate accelerator, 2021. URL `https://www.achronix.com/blog/embedded-fpga-ultimate-accelerator`. Accessed: 2021-11-10.

[7] Xilinx. Zynq-7000 SoC, 2021. URL `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`. Accessed: 2021-11-10.

[8] *WP533: Solution Efficiencies for Dynamic Function eXchange Using Abstract Shells*. Xilinx Inc., Jun. 2021. v1.0.

[9] Alfonso Rodríguez, Juan Valverde, Jorge Portilla, Andrés Otero, Teresa Riesgo, and Eduardo de la Torre. FPGA-Based High-Performance Embedded Systems for Adaptive Edge Computing in Cyber-Physical Systems: The ARTICo^3 Framework. *Sensors*, 18(6), 2018. ISSN 1424-8220. doi: 10.3390/s18061877. URL `https://www.mdpi.com/1424-8220/18/6/1877`.

[10] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. FOS: A modular FPGA operating system for dynamic workloads. *ACM Tran. Reconfigurable Technol. Syst.*, 13(4), September 2020. ISSN 1936-7406. doi: 10.1145/3405794.

[11] Kizheppatt Vipin and Suhaib A Fahmy. ZyCAP: Efficient partial reconfiguration management on the xilinx zynq. *IEEE Embedded Systems Letters*, 6(3):41–44, 2014.

[12] Shanker Shreejith, Bhaskar Banarjee, Kizheppatt Vipin, and Suhaib A Fahmy. Dynamic cognitive radios on the Xilinx Zynq hybrid FPGA. In *Int. Conf. on Cognitive Radio Oriented Wireless Networks*, pages 427–437. Springer, 2015.

[13] *WP470: Unleash the Unparalleled Power and Flexibility of Zynq UltraScale+ MPSoCs*. Xilinx Inc., Jun. 2016. v1.1.

[14] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq UltraScale+. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 215–220, 2020. doi: 10.1109/ICFPT51103.2020.00037.

[15] *UG1137: Zynq UltraScale+ MPSoC: Software Developers Guide*. Xilinx Inc., Jul. 2020. v12.0.

[16] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Network enabled partial reconfiguration for distributed FPGA edge acceleration. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 259–262, 2019. doi: 10.1109/ICFPT47387.2019.00042.

[17] Alex R. Bucknall and Suhaib A. Fahmy. Runtime abstraction for autonomous adaptive systems on reconfigurable hardware. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 1616–1621, 2021. doi: 10.23919/DATE51398.2021.9474199.

[18] Alex R. Bucknall and Suhaib A. Fahmy. ZyCAP2: End-to-end build tool and runtime manager for partial reconfiguration of FPGA SoCs at the edge. In *submitted to: TRETS*, 2021.

[19] Suhaib A. Fahmy. Design abstraction for autonomous adaptive hardware systems on fpgas. In *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, pages 142–147, 2018. doi: 10.1109/AHS.2018.8541489.

[20] Intel. Microcontrollers for intelligent things. Online, 2015. URL `https://www.intel.co.uk/content/www/uk/en/embedded/products/quark/overview.html`.

[21] Edson Luiz Padoin, Laércio Lima Pilla, Márcio Castro, Francieli Z Boito, Philippe Olivier Alexandre Navaux, and Jean-François Méhaut. Performance/energy trade-off in scientific computing: the case of ARM big.

LITTLE and intel sandy bridge. *IET Computers & Digital Techniques*, 9(1):27–35, 2015.

[22] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous CPU/GPU spark platform for machine learning algorithms. In *IEEE Int. Conf. on Networking, Architecture and Storage (NAS)*, pages 347–348. IEEE, 2015.

[23] Marcos Amaris, Raphael Y de Camargo, Mohamed Dyab, Alfredo Goldman, and Denis Trystram. A comparison of GPU execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th Int. Sym. on Network Computing and Applications (NCA)*, pages 326–333. IEEE, 2016.

[24] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Machine learning at the edge: Efficient utilization of limited CPU/GPU resources by multiplexing. In *IEEE 28th Int. Conf. on Network Protocols (ICNP)*, pages 1–6. IEEE, 2020.

[25] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21 (8):613–641, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359579. URL `https://doi.org/10.1145/359576.359579`.

[26] Google. Cloud tpu. Online, 2019. URL `https://cloud.google.com/tpu`.

[27] Vijeta Sharma, Gaurav Kumar Gupta, and Manjari Gupta. Performance benchmarking of GPU and TPU on google colaboratory for convolutional neural network. In *Applications of Artificial Intelligence in Engineering*, pages 639–646. Springer, 2021.

[28] Jurgen Vandendriessche, Nick Wouters, Bruno da Silva, Mimoun Lamrini, Mohamed Yassin Chkouri, and Abdellah Touhafi. Environmental sound recognition on embedded systems: From FPGAs to TPUs. *Electronics*, 10(21):2622, 2021.

[29] Xilinx. Deep learning processing unit. Online, 2019. URL `https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/dpu_over.html`.

[30] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[31] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. CNP: An fpga-based processor for convolutional networks. In *Int. Conf. on Field Programmable Logic and Applications*, pages 32–37. IEEE, 2009.

[32] Shady Soliman, Mohammed A Jaela, Abdelrhman M Abotaleb, Youssef Hassan, Mohamed A Abdelghany, Amr T Abdel-Hamid, Khaled N Salama, and Hassan Mostafa. FPGA implementation of dynamically reconfigurable IoT security module using algorithm hopping. *Integration*, 68:108–121, 2019.

[33] Cesar Da Costa, Mauro Hugo Mathias, and Masamori Kashiwagi. Development of an instrumentation system embedded on FPGA for real time measurement of mechanical vibrations in rotating machinery. In *Int. Sym. on Instrumentation & Measurement, Sensor Network and Automation (IMSNA)*, volume 1, pages 60–64. IEEE, 2012.

[34] Andres Upegui, Carlos Andrés Pena-Reyes, and Eduardo Sanchez. An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and microsystems*, 29(5):211–223, 2005.

[35] Amor Nafkha and Yves Louet. Accurate measurement of power consumption overhead during FPGA dynamic partial reconfiguration. In *Int. Sym. on Wireless Communication Systems (ISWCS)*, pages 586–591. IEEE, 2016.

[36] Erwan Moréac, El Mehdi Abdali, Francois Berry, Dominique Heller, and Jean-Philippe Diguet. Hardware-in-the-loop simulation with dynamic partial FPGA reconfiguration applied to computer vision in ROS-based UAV. In *Int. Workshop on Rapid System Prototyping (RSP)*, pages 1–7, 2020. doi: 10.1109/RSP51120.2020.9244863.

[37] Ratheesh Kalarot and John Morris. Comparison of FPGA and GPU implementations of real-time stereo vision. In *IEEE Computer Society Conf. on Computer Vision and Pattern Recognition-Workshops*, pages 9–15. IEEE, 2010.

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

[39] Khronos. Open standard for parallel programming of heterogeneous systems. Online, 2009. URL `https://www.khronos.org/opencl/`.

[40] Nvidia. Cuda. Online, 2019. URL https://developer.nvidia.com/cuda-zone.

[41] Xilinx. Vitis hls. Online, 2021. URL https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/introductionvitishls.html.

[42] Chisel. Chisel compiler framework. Online, 2021. URL https://www.chisel-lang.org/.

[43] nmigen. A refreshed python toolbox for building complex digital hardware. Online, 2021. URL https://nmigen.info/nmigen/.

[44] Xilinx. Zynq ultrascale+ MPSoC, 2021. URL https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html. Accessed: 2021-11-10.

[45] Xilinx. Vitis model composer. Online, 2021. URL https://www.xilinx.com/products/design-tools/vitis/vitis-model-composer.html.

[46] MATLAB. Hdl coder. Online, 2021. URL https://uk.mathworks.com/products/hdl-coder.html.

[47] Diana Gohringer, Michael Hubner, Volker Schatz, and Jurgen Becker. Runtime adaptive multi-processor system-on-chip: RAMPSoC. In *IEEE Int. Sym. on Parallel and Distributed Processing*, pages 1–7, 2008. doi: 10.1109/IPDPS.2008.4536503.

[48] Dirk Koch, Christian Beckhoff, and Jurgen Teich. ReCoBus-builder—a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs. In *Int. Conf. on field programmable logic and applications*, pages 119–124. IEEE, 2008.

[49] Christian Beckhoff, Dirk Koch, and Jim Torresen. Go ahead: A partial reconfiguration framework. In *IEEE 20th Int. Sym. on Field-Programmable Custom Computing Machines*, pages 37–44. IEEE, 2012.

[50] Kizheppatt Vipin and Suhaib A Fahmy. Mapping adaptive hardware systems with partial reconfiguration using CoPR for Zynq. In *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, 2015.

[51] Rafael Zamacola, Alberto Garcia Martinez, Javier Mora, Andres Otero, and Eduardo de La Torre. IMPRESS: Automated tool for the implementation of highly flexible partial reconfigurable systems with xilinx vivado. In *2018 Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2018. doi: 10.1109/RECONFIG.2018.8641703.

[52] Michael Xi Yue, Dirk Koch, and Guy GF Lemieux. Rapid overlay builder for xilinx FPGAs. In *IEEE 23rd Annual Int. Sym. on Field-Programmable Custom Computing Machines*, pages 17–20. IEEE, 2015.

[53] C. Lavin and A. Kaviani. Rapidwright: Enabling custom crafted implementations for FPGAs. In *2018 IEEE 26th Annual Int. Sym. on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140, April 2018. doi: 10.1109/FCCM.2018.00030.

[54] Joel Mandebi Mbongue, Danielle Tchuinkou Kwadjo, and Christophe Bobda. Automatic generation of application-specific FPGA overlays with rapidwright. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 303–306, 2019. doi: 10.1109/ICFPT47387.2019.00053.

[55] Nadir Khan, Jorge Castro-Godínez, Shixiang Xue, Jörg Henkel, and Jürgen Becker. Automatic floorplanning and standalone generation of bitstream-level IP cores. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 29(1):38–50, 2021. doi: 10.1109/TVLSI.2020.3023548.

[56] Enno Lubbers and Marco Platzner. ReconOS: An RTOS supporting hard-and software threads. In *IEEE Int. Conf. on Field Programmable Logic and Applications*, pages 441–446, 2007.

[57] Enno Lübbers and Marco Platzner. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):1–33, 2009.

[58] Aws Ismail and Lesley Shannon. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In *IEEE Int. Sym. on Field-Programmable Custom Computing Machines*, pages 170–177, 2011.

[59] Khoa Dang Pham, Anuj Vaishnav, Malte Vesper, and Dirk Koch. ZUCL: a ZYNQ ultrascale+ framework for OpenCL HLS applications. In *FSP Workshop 2018; Fifth Int. Workshop on FPGAs for Software Programmers*, pages 1–9. VDE, 2018.

[60] Khoa Dang Pham, Kyriakos Paraskevas, Anuj Vaishnav, Andrew Attwood, Malte Vesper, and Dirk Koch. ZUCL 2.0: Virtualised memory and communication for ZYNQ ultrascale+ FPGAs. In *FSP Workshop 2019; Sixth Int. Workshop on FPGAs for Software Programmers*, pages 1–9. VDE, 2019.

[61] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. ReconOS: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71, 2013.

[62] Diana Göhringer, Michael Hübner, Etienne Nguepi Zeutebouo, and Jürgen Becker. CAP-OS: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures. In *IEEE Int. Sym. on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1–8, 2010. doi: 10.1109/IPDPSW.2010.5470732.

[63] Jens Rettkowski, Philipp Wehner, Evgheni Cutiscev, and Diana Göhringer. LinROS: A linux-based runtime system for reconfigurable MPSoCs. In *IEEE Int. Parallel and Distributed Processing Sym. Workshops (IPDPSW)*, pages 208–216, 2016. doi: 10.1109/IPDPSW.2016.156.

[64] Xabier Iturbe, Khaled Benkrid, Chuan Hong, Ali Ebrahim, Raul Torrego, Imanol Martinez, Tughrul Arslan, and Jon Perez. R3TOS: a novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on FPGAs. *IEEE Trans. on computers*, 62 (8):1542–1556, 2013.

[65] Adewale Adetomi, Godwin Enemali, Xabier Iturbe, Tughrul Arslan, and Didier Keymeulen. R3TOS-based integrated modular space avionics for on-board real-time data processing. In *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, pages 1–8. IEEE, 2018.

[66] Simen Gimle Hansen, Dirk Koch, and Jim Torresen. High speed partial run-time reconfiguration using enhanced ICAP hard macro. In *IEEE Int. Sym. on Parallel and Distributed Processing Workshops and PhD Forum*, pages 174–180. IEEE, 2011.

[67] Luca Pezzarossa, Martin Schoeberl, and Jens Sparsø. A controller for dynamic partial reconfiguration in FPGA-based real-time systems. In *IEEE 20th Int. Sym. on Real-Time Distributed Computing (ISORC)*, pages 92–100. IEEE, 2017.

[68] Amit Kulkarni, Vipin Kizheppatt, and Dirk Stroobandt. MiCAP: a custom reconfiguration controller for dynamic circuit specialization. In *Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[69] Amit Kulkarni and Dirk Stroobandt. MiCAP-Pro: a high speed custom reconfiguration controller for dynamic circuit specialization. *Design Automation for Embedded Systems*, 20(4):341–359, 2016.

[70] Kizheppatt Vipin and Suhaib A Fahmy. DyRACT: A partial reconfiguration enabled accelerator and test platform. In *24th Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–7, 2014.

[71] Ahmad Sadek, Hassan Mostafa, Amin Nassar, and Yehea Ismail. Towards the implementation of multi-band multi-standard software-defined radio using dynamic partial reconfiguration. *Int. Journal of Comms. Systems*, 30(17):e3342, 2017.

[72] Pedro Reviriego, Anees Ullah, and Salvatore Pontarelli. PR-TCAM: Efficient TCAM emulation on xilinx FPGAs using partial reconfiguration. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 27(8): 1952–1956, 2019. doi: 10.1109/TVLSI.2019.2903980.

[73] Xiaoyao Li, Xiuxiu Wang, Fangming Liu, and Hong Xu. DHL: Enabling flexible software network functions with FPGA acceleration. In *IEEE 38th Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 1–11. IEEE, 2018.

[74] Milica Orlandić and Kjetil Svarstad. An adaptive high-throughput edge detection filtering system using dynamic partial reconfiguration. *Journal of Real-Time Image Processing*, 16(6):2409–2424, 2019.

[75] Afandi Ahmad, Abbes Amira, Paul Nicholl, and Benjamin Krill. FPGA-based IP cores implementation for face recognition using dynamic partial reconfiguration. *Journal of real-time image processing*, 8(3):327–340, 2013.

[76] Sheetal Bhandari, Shaila Subbaraman, Shashank Pujari, Fabio Cancare, Francesco Bruschi, Marco D Santambrogio, and Paolo Roberto Grassi. High speed dynamic partial reconfiguration for real time multimedia signal processing. In *15th Euromicro Conf. on Digital System Design*, pages 319–326. IEEE, 2012.

[77] Hanaa M Hussain, Khaled Benkrid, and Huseyin Seker. Dynamic partial reconfiguration implementation of the SVM/KNN multi-classifier on FPGA for bioinformatics application. In *37th Annual Int. Conf. of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 7667–7670. IEEE, 2015.

[78] Heba Elhosary, Michael H. Zakhari, Mohamed A. Elgammal, Khaled A. Helal Kelany, Mohamed A. Abd El Ghany, Khaled N. Salama, and Hassan Mostafa. Hardware acceleration of high sensitivity power-aware epileptic seizure detection system using dynamic partial reconfiguration. *IEEE Access*, 9:75071–75081, 2021. doi: 10.1109/ACCESS.2021.3079155.

[79] Hasan Irmak, Daniel Ziener, and Nikolaos Alachiotis. Increasing flexibility of FPGA-based CNN accelerators with dynamic partial reconfiguration.

In *31st Int. Conf. on Field-Programmable Logic and Applications (FPL)*, pages 306–311, 2021. doi: 10.1109/FPL53798.2021.00061.

[80] Shanker Shreejith, Suhaib A Fahmy, and Martin Lukasiewycz. Reconfigurable computing in next-generation automotive networks. *IEEE embedded systems letters*, 5(1):12–15, 2013.

[81] Bikash Poudel and Arslan Munir. Design and evaluation of a reconfigurable ECU architecture for secure and dependable automotive CPS. *IEEE Trans. on Dependable and Secure Computing*, 2018.

[82] Shanker Shreejith, Kizhepatt Vipin, Suhaib A Fahmy, and Martin Lukasiewycz. An approach for redundancy in FlexRay networks using FPGA partial reconfiguration. In *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 721–724. IEEE, 2013.

[83] M Ceschia, M Violante, M Sonza Reorda, A Paccagnella, P Bernardi, M Rebaudengo, D Bortolato, M Bellato, P Zambolin, and A Candelori. Identification and classification of single-event upsets in the configuration memory of SRAM-based FPGAs. *IEEE Trans. on Nuclear Science*, 50 (6):2088–2094, 2003.

[84] Khoa Pham, Edson Horta, Dirk Koch, Anuj Vaishnav, and Thomas Kuhn. IPRDF: An isolated partial reconfiguration design flow for xilinx FPGAs. In *IEEE 12th Int. Sym. on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 36–43. IEEE, 2018.

[85] Björn Osterloh, Harald Michalik, Sandi Alexander Habinc, and Björn Fiethe. Dynamic partial reconfiguration in space applications. In *NASA/ESA Conf. on Adaptive Hardware and Systems*, pages 336–343. IEEE, 2009.

[86] Xabier Iturbe, Khaled Benkrid, Tughrul Arslan, Chuan Hong, Ahmet T Erdogan, and Imanol Martinez. Enabling FPGAs for future deep space exploration missions: improving fault-tolerance and computation density with R3TOS. In *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, pages 104–112. IEEE, 2011.

[87] Maryam Hemmati, Morteza Biglari-Abhari, and Smail Niar. Adaptive vehicle detection for real-time autonomous driving system. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1034–1039, 2019. doi: 10.23919/DATE.2019.8714818.

[88] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Building the computing system for autonomous micromobility vehicles:

Design constraints and architectural optimizations. In *53rd Annual IEEE/ACM Int. Sym. on Microarchitecture (MICRO)*, pages 1067–1081. IEEE, 2020.

[89] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.

[90] Li Du, Yuan Du, Yilei Li, Junjie Su, Yen-Cheng Kuan, Chun-Chen Liu, and Mau-Chung Frank Chang. A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things. *IEEE Trans. on Circuits and Systems I: Regular Papers*, 65(1):198–208, 2018.

[91] Naoya Chujo. Fail-safe ECU system using dynamic reconfiguration of FPGA. *R & D Review of Toyota CRDL*, 37(2):54–60, 2002.

[92] Alex R Bucknall, Shanker Shreejith, and Suhaib A Fahmy. Network enabled partial reconfiguration for distributed FPGA edge acceleration. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 259–262. IEEE, 2019.

[93] *DS020: Xilinx Prototype Platforms User Guide for Virtex and Virtex-E Series FPGA*. Xilinx Inc., Dec. 1999. v1.1.

[94] Ken Eguro. SIRC: An extensible reconfigurable computing communication API. In *18th IEEE Annual Int. Sym. on Field-Programmable Custom Computing Machines*, pages 135–138. IEEE, 2010.

[95] Ming Liu, Wolfgang Kuehn, Zhonghai Lu, and Axel Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Int. Conf. on Field Programmable Logic and Applications*, pages 498–502. IEEE, 2009.

[96] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *IEEE 22nd Annual Int. Sym. on Field-Programmable Custom Computing Machines*, pages 109–116. IEEE, 2014.

[97] Martin Geier, Florian Pitzl, and Samarjit Chakraborty. GigE Vision data acquisition for visual servoing using SG/DMA proxying. In *14th ACM/IEEE Sym. on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–10. IEEE, 2016.

[98] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte

Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Int. Workshop on Cryptographic Hardware and Embedded Systems*, pages 450–466. Springer, 2007.

[99] Nagham Samir, Yousef Gamal, Ahmed N El-Zeiny, Omar Mahmoud, Ahmed Shawky, AbdelRahman Saeed, and Hassan Mostafa. Energy-Adaptive Lightweight Hardware Security Module using Partial Dynamic Reconfiguration for Energy Limited Internet of Things Applications. In *IEEE Int. Sym. on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2019.

[100] Secworks. secworks/aes256, Feb 2014. URL `https://github.com/secworks/aes`.

[101] Shadi Al-Sarawi, Mohammed Anbar, Kamal Alieyan, and Mahmood Alzubaidi. Internet of Things (IoT) communication protocols. In *8th Int. Conf. on information technology (ICIT)*, pages 685–690. IEEE, 2017.

[102] *UG585: Zynq-7000 SoC All Programmable SoC Technical Reference-Manual*. Xilinx Inc., Jul. 2018. v1.12.2.

[103] *Python productivity for Zynq*. Xilinx Inc. URL `http://pynq.io/`.

[104] Nadir Khan, Jorge Castro-Godínez, Shixiang Xue, Jörg Henkel, and Jürgen Becker. Automatic floorplanning and standalone generation of bitstream-level ip cores. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 29(1):38–50, 2020.

[105] Zongwei Zhu, Junneng Zhang, Jinjin Zhao, Jing Cao, Duan Zhao, Gangyong Jia, and Qingyong Meng. A hardware and software task-scheduling framework based on cpu+fpga heterogeneous architecture in edge computing. *IEEE Access*, 7:148975–148988, 2019. doi: 10.1109/ACCESS.2019.2943179.

[106] Khoa Pham, Dirk Koch, Anuj Vaishnav, Konstantinos Georgopoulos, Pavlos Malakonakis, Aggelos Ioannou, and Iakovos Mavroidis. Moving compute towards data in heterogeneous multi-FPGA clusters using partial reconfiguration and I/O virtualisation. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 221–226. IEEE, 2020.

[107] Christian Lienen, Marco Platzner, and Bernhard Rinner. Reconros: Flexible hardware acceleration for ros2 applications. In *Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 268–276. IEEE, 2020.

[108] Kizheppatt Vipin and Suhaib A Fahmy. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys*, 51(4):1–39, 2018.

[109] Olof Kindgren. Edalize: Python library for interacting with eda tools. `https://github.com/olofk/edalize`, 2018.

[110] Whitequark. nmigen. `https://github.com/nmigen/nmigen`, 2018.

[111] Kevin E Murray, Mohamed A Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. Symbiflow and VPR: An open-source design flow for commercial and novel FPGAs. *IEEE Micro*, 40(4):49–57, 2020.

[112] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog HDL. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer Int. Publishing, Apr 2015. doi: 10.1007/978-3-319-16214-0_42. URL `http://dx.doi.org/10.1007/978-3-319-16214-0_42`.

[113] Kizheppatt Vipin and Suhaib A Fahmy. Automated partial reconfiguration design for adaptive systems with CoPR for Zynq. In *22nd Annual Int. Sym. on Field-Programmable Custom Computing Machines*, pages 202–205. IEEE, 2014.

[114] Marco Rabozzi, Gianluca Carlo Durelli, Antonio Miele, John Lillis, and Marco Domenico Santambrogio. Floorplanning automation for partial-reconfigurable FPGAs via feasible placements generation. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 25(1):151–164, 2017. doi: 10.1109/TVLSI.2016.2562361.

[115] Christian Beckhoff, Dirk Koch, and Jim Torreson. Automatic floorplanning and interface synthesis of island style reconfigurable systems with GOAHEAD. In *Int. Conf. on Architecture of Computing Systems*, pages 303–316. Springer, 2013.

[116] Kizheppatt Vipin and Suhaib A Fahmy. Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In *Int. Sym. on Applied Reconfigurable Computing*, pages 13–25. Springer, 2012.

[117] Norbert Deak, Octavian Cret, and Horia Hedesiu. Efficient FPGA floorplanning for partial reconfiguration-based applications. In *IEEE 27th Annual Int. Sym. on Field-Programmable Custom Computing Machines (FCCM)*, pages 309–309, 2019. doi: 10.1109/FCCM.2019.00050.

[118] *PG116: MicroBlaze Micro Controller System v3.0.* Xilinx Inc., Jul. 2021. v3.0.

[119] Xilinx. Ultra96v2 development kit. Online, 2021. URL `https://www.xilinx.com/products/boards-and-kits/1-vad4rl.html`.

[120] *PG085: AXI4-Stream Infrastructure IP Suite v3.0.* Xilinx Inc., Dec. 2018. v3.0.

[121] *PG059: AXI Interconnect v2.1.* Xilinx Inc., Dec. 2017. v2.1.

[122] Olof Kindgren. FuseSoC: Package manager and build abstraction tool for fpga/asic development. `https://github.com/olofk/fusesoc`, 2018.

[123] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. Minimizing partial reconfiguration overhead with fully streaming dma engines and intelligent icap controller. In *FPGA*, page 292. Citeseer, 2010.

[124] François Duhem, Fabrice Muller, and Philippe Lorenzini. Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA. In *Int. Sym. on Applied Reconfigurable Computing*, pages 253–260. Springer, 2011.

[125] Kawazome Ichiro. u-dma-buf. `https://github.com/ikwzm/udmabuf`, 2015.

[126] Brendan Perez and Jared Choi. Xilinx AXI DMA linux driver. `https://github.com/bperez77/xilinx_axidma`, 2015.

[127] 46cv8. Xilinx AXI DMA linux driver - modified. `https://github.com/46cv8/xilinx_axidma/tree/axidma_loopback/master`, 2021.

[128] *UG909: Dynamic Function eXchange v2019.2.* Xilinx Inc., Jan. 2020. v2019.2.

[129] Xilinx, 2021. URL `https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-vision.html`.

[130] Shanker Shreejith, Kizhepatt Vipin, Suhaib A. Fahmy, and Martin Lukasiewycz. An approach for redundancy in flexray networks using fpga partial reconfiguration. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 721–724, 2013. doi: 10.7873/DATE.2013.155.

[131] Mustapha Bouhali, Farid Shamani, Zine Elabadine Dahmane, Abdelkader Belaidi, and Jari Nurmi. FPGA applications in unmanned aerial vehicles-a review. In *Int. Sym. on Applied Reconfigurable Computing*, pages 217–228, 2017.

[132] Simon Haykin. Cognitive radio: brain-empowered wireless communications. *IEEE Journal on Selected Areas in Communications*, 23(2): 201–220, 2005.

[133] Cong Hao et al. A hybrid GPU+ FPGA system design for autonomous driving cars. In *IEEE Int. Workshop on Signal Processing Systems (SiPS)*, pages 121–126, 2019.

[134] Jonathan Kok, Luis Felipe Gonzalez, and Neil Kelson. FPGA implementation of an evolutionary algorithm for autonomous unmanned aerial vehicle on-board path planning. *IEEE Trans. on Evolutionary Computation*, 17 (2):272–281, 2012.

[135] Yecheng Lyu, Lin Bai, and Xinming Huang. Real-time road segmentation using LiDAR data processing on an FPGA. In *IEEE Int. Sym. on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[136] Pooyan Jamshidi, Javier Cámara, Bradley Schmerl, Christian Käestner, and David 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)*, pages 39–50, 2019.

[137] Blake Fuller, Jonathan Kok, Neil A Kelson, and Luis 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.

[138] Claudio Savaglio and Giancarlo Fortino. Autonomic and cognitive architectures for the Internet of Things. In *Int. Conf. on Internet and Distributed Computing Systems*, pages 39–47, 2015.

[139] Zishen Wan, Bo Yu, Thomas Yuang Li, Jie Tang, Yuhao Zhu, Yu Wang, Arijit Raychowdhury, and Shaoshan Liu. A survey of FPGA-based robotic computing. *IEEE Circuits and Systems Magazine*, 21(2):48–74, 2021.

[140] Nur Alisa Ali, Sani Irwan Md Salim, Rosman Abd Rahim, Siti Aisyah Anas, Zarina Mohd Noh, and Sharatul Izah Samsudin. PWM controller design of a hexapod robot using FPGA. In *IEEE Int. Conf. on Control System, Computing and Engineering*, pages 310–314. IEEE, 2013.

[141] Donald Bailey, Miguel Contreras, and Gourab Sen Gupta. Towards automatic colour segmentation for robot soccer. In *6th Int. Conf. on Automation, Robotics and Applications (ICARA)*, pages 478–483. IEEE, 2015.

[142] Henry Hoffmann. *SEEC: A framework for self-aware management of goals and constraints in computing systems*. PhD thesis, Massachusetts Institute of Technology, 2013.

[143] John Strassner, Nazim Agoulmine, and Elyes Lehtihet. FOCALE: A novel autonomic networking architecture. *ITSSA Journal*, 3(1):64–79, 2006.

[144] Suhaib A Fahmy. Design abstraction for autonomous adaptive hardware systems on FPGAs. In *NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, pages 142–147, 2018.

[145] Stuart Clayman and Alex Galis. INOX: a managed service platform for inter-connected smart objects. In *Workshop on Internet of Things and Service Platforms*, 2011.

[146] Morgan Quigley et al. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, number 3.2, page 5, 2009.

[147] Vincenzo DiLuoffo, William R Michalson, and Berk Sunar. Robot operating system 2: The need for a holistic security approach to robotic architectures. *Int. Journal of Advanced Robotic Systems*, 15(3), 2018.

[148] Anthony Colaprete, Daniel Andrews, William Bluethmann, Richard C Elphic, Ben Bussey, Jay Trimble, Kris Zacny, and Janine E Captain. An overview of the volatiles investigating polar exploration rover (viper) mission. In *AGU Fall Meeting Abstracts*, pages P34B–03, 2019.

[149] iRobot. irobot launches create 3, with ros 2 built in. Online, 2022. URL `https://spectrum.ieee.org/irobot-create-3`.

[150] Ariel Podlubne, Julian Haase, Lester Kalms, Gökhan Akgün, Muhammad Ali, Habib Ulhasan Khar, Ahmed Kamal, and Diana Göhringer. Low power image processing applications on fpgas using dynamic voltage scaling and partial reconfiguration. In *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, pages 64–69. IEEE, 2018.

[151] Rogério De Lemos et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. 2013.

[152] ZeroMQ, 2021. URL `https://zeromq.org/`.

[153] OpenDDS. URL `https://opendds.org/`.

[154] Protocol buffers. URL `https://developers.google.com/protocol-buffers/docs/overview`.