ZyPR: End-to-end Build Tool and Runtime Manager for Partial Reconfiguration of FPGA SoCs at the Edge

ALEX R. BUCKNALL, University of Warwick, United Kingdom
SUHAIB A. FAHMY, King Abdullah University of Science and Technology (KAUST), Saudi Arabia

Partial reconfiguration (PR) is a key enabler to the design and development of adaptive systems on modern Field Programmable Gate Array (FPGA) Systems-on-Chip (SoCs), allowing hardware to be adapted dynamically at runtime. Vendor-supported PR infrastructure is performance-limited and blocking, drivers entail complex memory management, and software/hardware design requires bespoke knowledge of the underlying hardware. This article presents ZyPR: a complete end-to-end framework that provides high-performance reconfiguration of hardware from within a software abstraction in the Linux userspace, automating the process of building PR applications with support for the Xilinx Zynq and Zynq UltraScale+ architectures, aimed at enabling non-expert application designers to leverage PR for edge applications. We compare ZyPR against traditional vendor tooling for PR management as well as recent open source tools that support PR under Linux. The framework provides a high-performance runtime along with low overhead for its provided abstractions. We introduce improvements to our previous work, increasing the provisioning throughput for PR bitstreams on the Zynq Ultrascale+ by 2× and 5.4× compared to Xilinx's FPGA Manager.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; *System on a chip*; • **Software and its engineering** → *Development frameworks and environments;*

Additional Key Words and Phrases: Field programmable gate arrays, partial reconfiguration, adaptive systems

ACM Reference format:
Alex R. Bucknall and Suhaib A. Fahmy. 2023. ZyPR: End-to-end Build Tool and Runtime Manager for Partial Reconfiguration of FPGA SoCs at the Edge. *ACM Trans. Reconfig. Technol. Syst.* 16, 3, Article 34 (June 2023), 33 pages.
https://doi.org/10.1145/3585521

1 INTRODUCTION

Field Programmable Gate Arrays (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. FPGAs in edge computing have been used to provide acceleration for applications such as machine learning [37], image processing [47], and in-network processing [29]. Datacenters have found use for FPGAs in the acceleration of complex computing tasks [9]. While FPGAs excel at accelerating specific computations, system tasks such as hosting an **operating**

This work was supported by the UK Engineering and Physical Sciences Research Council, grant EP/N509796/1.
Authors' addresses: A. R. Bucknall, University of Warwick, Gibbet Hill Road, Coventry, Warwickshire, United Kingdom; email: a.r.bucknall@warwick.ac.uk; S. A. Fahmy, Computer, Electrical and Mathematical Sciences and Engineering, King Abdul-lah University of Science and Technology (KAUST), Thuwal, 23955, Saudi Arabia; email: suhaib.fahmy@kaust.edu.sa.

1936-7406/2023/06-ART34 $15.00
https://doi.org/10.1145/3585521

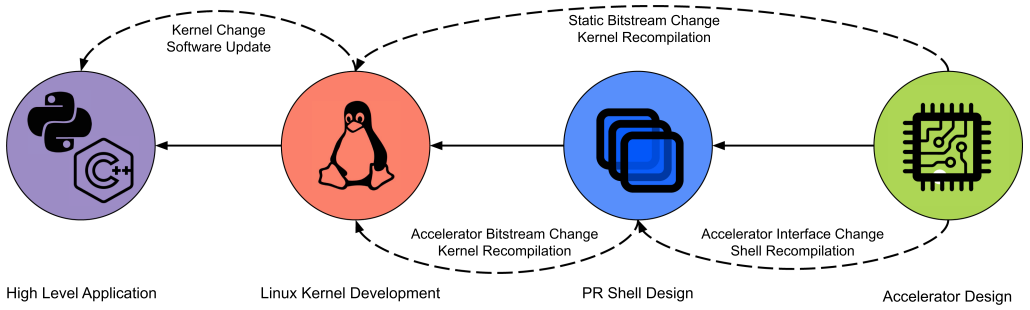

Fig. 1. Example Linux PR workflow. Designers are required to propagate their changes up from the accelerator, through to the shell, the Linux kernel, as well as track PL changes from their high-level applications.

system (OS) and managing high-level networking are better suited to general purpose processors such as CPUs. Numerous academic works have demonstrated performance characteristics and specialisation of discrete computing devices such as FPGAs, GPUs, and CPUs, defining their comparative strengths [3]. The combination of FPGA hardware for high-performance accelerators and general purpose **processing systems (PS)** or CPUs has led to popular heterogeneous **System on Chip (SoC)** platforms from leading manufacturers, such as the Zynq and Zynq UltraScale+ (ZynqMP) 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 (PL)** 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 (PR)**, which allows the FPGA to modify/update specific regions while still processing data. There are examples of PR being used in a variety of domains, such as space applications [1, 28], in datacenters for cloud computing [12, 21], as well as image-processing systems [25]. Given the rise in popularity of FPGAs as platforms for neural networks [15], generalised support for runtime PR is becoming increasingly important to enable the performance of specialised hardware to be coupled with some of the flexibility of software. Managing the state of the FPGA hardware logic from software running on a CPU is challenging, making designing and deploying such systems extremely complex.

For PR to finally become feasible in mainstream applications, a number of challenges must be addressed: (1) designing and building PR systems should be possible by non-experts; (2) abstractions between hardware and software must be managed such that both PR and accelerator performance are 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 detailed platform 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 [32]. Figure 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 [42] platform do provide some hardware abstraction under a Python SDK and allow for independent **register transfer level (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 ZyPR 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.

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. In an earlier publication [8], we introduced an early set of abstractions that we have built upon to enable an **end-to-end (E2E),** from FPGA design to Linux image build, tool that allows for complete abstracted PR application development. The following contributions were introduced in Reference [8]:

- A high-performance PR controller and Linux runtime service (limited to approx. 380 MiB/s)
- PR build tool (only Vivado 2018.3, non-extensible, and no support for end-to-end designs)
- Generated PR infrastructure using shared DMA resources (between accelerators and ICAP) with no support for PR chaining

Our complete framework, ZyPR, extends the previous work on ZyCAP and fulfils these criteria by automatically constructing a ready-to-go Linux image for the designer to implement their higher-level software applications, without needing to concern themselves with complexities of hardware management from the Linux userspace. We abstract hardware and driver deployment, generating the required infrastructure and kernel modules at build time and exporting human readable JSON objects that may by consumed by our runtime tool as well as be updatable/modified by the designer to tweak behaviour. The key contributions of our framework are as follows:

- ***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 (up to Vivado 2020.3)
- ***A comprehensive hardware-software build abstraction*** that simplifies 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
- ***Improved high-performance asynchronous PR controller*** using the ZynqMP ICAP interface at near theoretical throughput (increased to 757 MiB/s)
- ***A runtime PR configuration API*** for PS-PL management that enables simple software abstraction of memory-mapped I/O, DMA streaming, as well as loading/unloading partial and complete bitstreams as part of our described mode and configuration abstractions

- ***A Vitis HLS case study using OpenCV edge accelerators in a PR application*** that demonstrates our software abstraction and benchmarks performance and resources usage

2 CONCEPTS

We define a number of important concepts in the context of our custom tools and specify their relevance to this work; 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 that 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 an initial setup file, not to be confused with the definition of configurations; file is referred to as the `spec.json`.

2.1 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 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 this time, Xilinx offers two device families for Heterogeneous SoCs, the Zynq and Zynq Ultrascale+; our framework supports both devices.

2.2 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, and so on. 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.

2.3 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. Reference [36] 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 (DPR)** describes a feature of the FPGA's to 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. In the context of this article, we use the term PR to describe dynamic PR.

2.4 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 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 article presents abstractions for the reduction of complexity in the build times and runtimes specifically for PR applications. Reducing the development complexity of the hardware accelerators themselves remains 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.

2.5 PR Runtime (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. To load the PCAP, FPGA Manager must perform a sequence of register reads and writes to the **Configuration Security Unit (CSU)** registers that are managed by the **ARM Trusted Firmware (ATF)** and then the **Platform Management Unit (PMU)**. 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 Figure 2.

We argue against using FPGA Manager for PR management for its limited performance (due to the use of PCAP), blocking software paradigm (which halts the software flow until PR is complete) and inability to apply design tree fragments while performing PR (required to alert the Linux kernel to hardware changes). FPGA Manager also does not support caching of bitstreams and loads bitstreams from the Linux filesystem, rather than higher-performance physical memory-mapped locations.

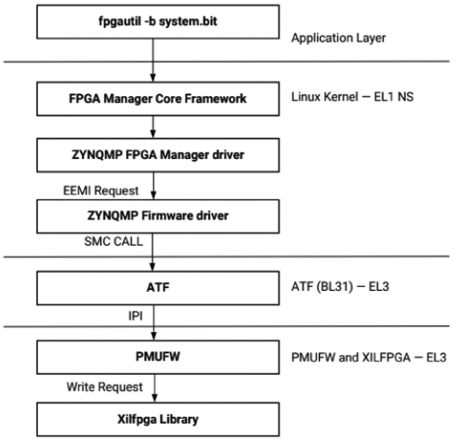

Fig. 2. Loading of the PCAP from FPGA Manager (ZynqMP)[43].

3 BACKGROUND

To effectively explain the current state of research for PR design flows, both vendor tooling and academic works are evaluated.

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 in future work.

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 Dynamic Function Exchange 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. We specifically examined Vivado from version 2019.2 onwards, with the introduction of the Dynamic Function Exchange tools.

3.1.2 Dynamic Function Exchange. **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 is a collection of tools provided by Xilinx to reduce some of the complexities associated with designing partially reconfigurable applications. 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 **Direct Memory Access (DMA)** from the PS.

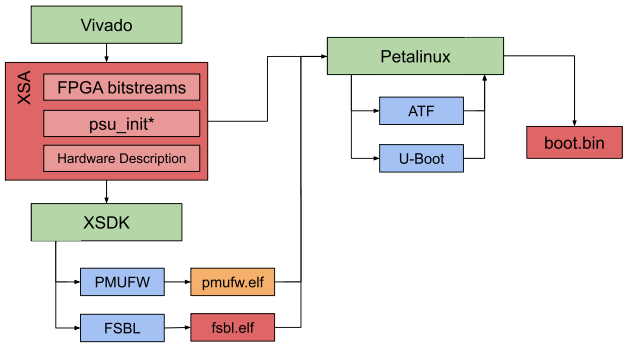

Fig. 3. The Xilinx Linux build flow.

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.

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. Figure 3 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.

Figure 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.

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 [4] 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. Reference [46] 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

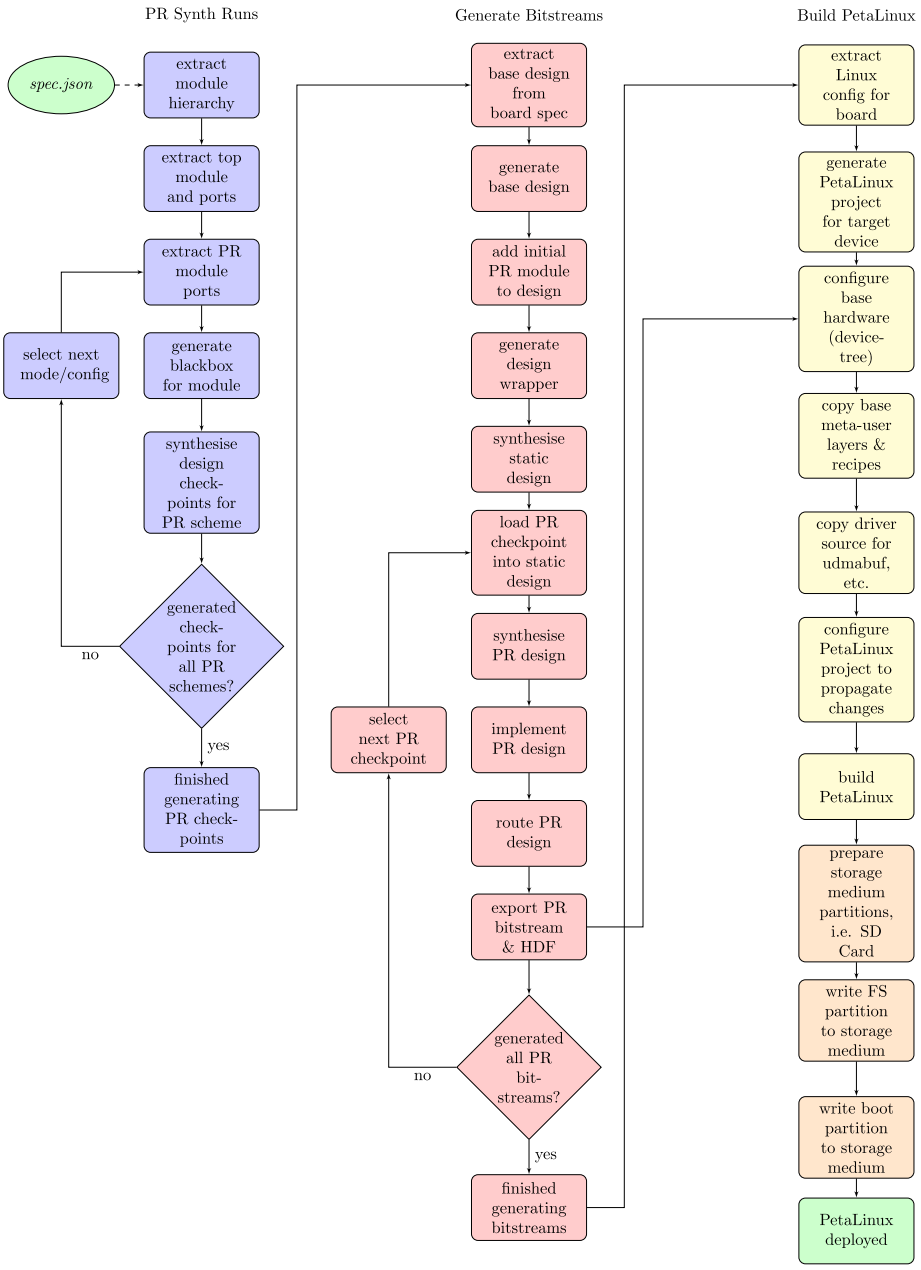

Fig. 4. Stages of the PR build flow [8].

the partial bitstreams and thus further increases the time taken to load over the PCAP interface. More recently, Reference [16] 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 Ultra-scale+ devices. The work in Reference [48] 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 [32] are designed to isolate each element of the workflow such that designers with expertise can independently develop these components, while others [2] offer highly integrated workflows that tightly integrate the movement of data between PS and PL using custom APIs.

3.2.1 FPGA Operating System. FOS [32] 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 [24], 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, and so on, 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 that 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 more suited for centralised offloaded compute applications.

Other tools [48] have also focused on scheduling of PR systems, aiming to provide a framework for efficient task switching and provisioning of the PL within heterogeneous systems. As there is a significant body of work in this domain, our tools predominately focus on the abstraction as well as the build process for the designer, rather than the well-established domain of PR scheduling. 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 Reference [23] suggests extensions to FOS that provide plugin support for the ICAP. However, the driver and HDL code for this extension not been incorporated into the open source release of 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 Reference [7]. Table 1 shows a comparison of current competing tools that target relevant devices and current vendor place and route tooling. A comparison of runtime managers is shown in Table 2.

3.2.2 ReConOS. ReConOS [2] 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,

Table 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	●	●	◐	○	○	○	○	○	○	○	○	○
ZyPR	●	◐	◐	○	●	●	●	●	●	●	●	●

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

Table 2. 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	○	●	●	○	●	○	◐	○	○
ZyPR	●	●	●	○	●	●	◐	●	●

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

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 [19]. 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.

3.2.3 ARTICo³. Reference [27] is another automated toolchain designed for PR application deployment. They describe their tool as being able to dynamically adapt between tradeoffs with computing performance, energy consumption, and fault tolerance, meeting the demands of a cyber physical system. ARTICo³ 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 Reference [27]) 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, and 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 ARTICo³ 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.

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 are common across the different tools, pushing designers to learn custom workflows for designing/wrapping their acceleration functions as well as writing software with consideration for framework specific APIs. 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.

3.3 PR Managers

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 2, 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 Section 2.5. We 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 are comparatively low compared to other methods, later highlighted in Section 6.2. 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 Reference [20] 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 800 MB/s [11]. 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 [35] 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 382 MB/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 Reference [6] showed how the ICAP could be programmed over DMA on the Zynq Ultrascale+ devices, and the authors of Reference [23] increased the clock frequency of the ICAP to 200 MHz, further increasing performance to close to the theoretical 800 MB/s.

4 BUILD TOOLFLOW

Our workflow differs from current tools with its aims and implementation details, generating PR infrastructure specific for serving edge applications. The ZyPR 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 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. Our tools extend recent works by:

- 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. The build tool utilises the *Edalize* Python library for interacting with EDA tools programmatically [17]. *Edalize* is used to inject template TCL scripts that control the Vivado workflow at build time.

4.1 Hardware Abstraction

Under the framework, states, modes, and configurations are defined to abstract hardware control from the designer's software application [6]. 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*, which 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)**.

4.2 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. Figure 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 [31], 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 (HLS)**-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 ZyPR runtime, which will set the AXI-Stream arbiter master and slave addresses according to the applied configurations. In future work, we aim to improve port extraction by providing full support for VHDL and SystemVerilog.

4.2.1 Compile-time Generated Interfacing. To allow for a variety of PRRs, the build tool generates infrastructure at compile-time to support the accelerator interfaces. Module interfaces are determined using a custom Python library, interfacer, which is able to extract supported protocol interfaces, which the build tool matches and generates infrastructure for, including AXI interconnects and AXI-Stream arbitrators (multiplexer/demultiplexers). Currently this supports AXI Standard/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 determines the infrastructure to generate, such as setting default widths for interconnects as well as bus-width converters, if required. At present, the Interfacer library supports Verilog modules, however, due to the nature of this Python library, adding support for additional languages only requires a port/interface parser for the target language. In future work, we intend to support SystemVerilog and VHDL. The library does not currently handle differing/crossing clock domains between PRRs, as this requires specific attention when floor

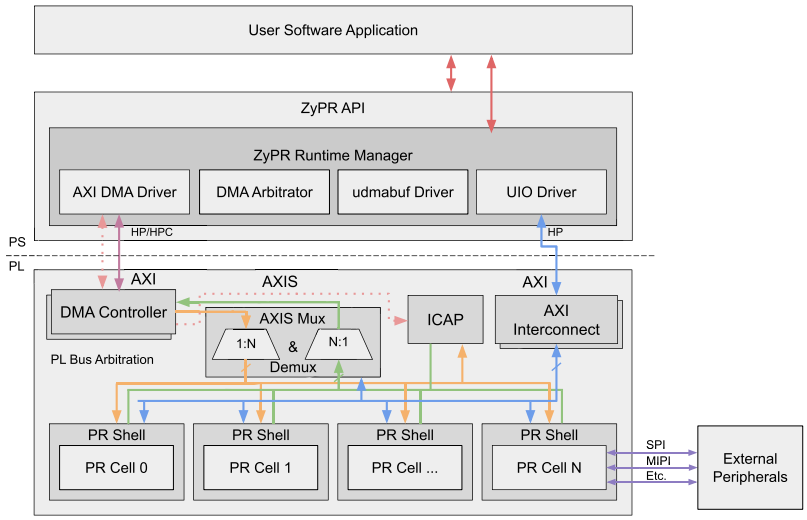

Fig. 5. PL architecture generated using the ZyPR build tooling.

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.

Listing 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 Figure 5.

```
"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 1. AXI-Stream Master Interface.

4.2.2 Automatic PR Region Generation. The tools will search the user's *specification* file for PRRs, assign the specified *pBlocks*, and build the PR logic accordingly. Listing 2 shows a *specification* file with a two-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 that resource requirements exceed the PRR specified (specification shown in Listing 2).

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

Listing 2. Multi-region specification with pblock definition.

Figure 6 shows an example of the post-build generated wrappers for the user's HDL modules. In this instance, the tools generated two wrappers (supporting two 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 32 bit AXI-Stream and another with a 64 bit 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 [5, 26, 34] 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 References [10, 33].

4.2.3 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

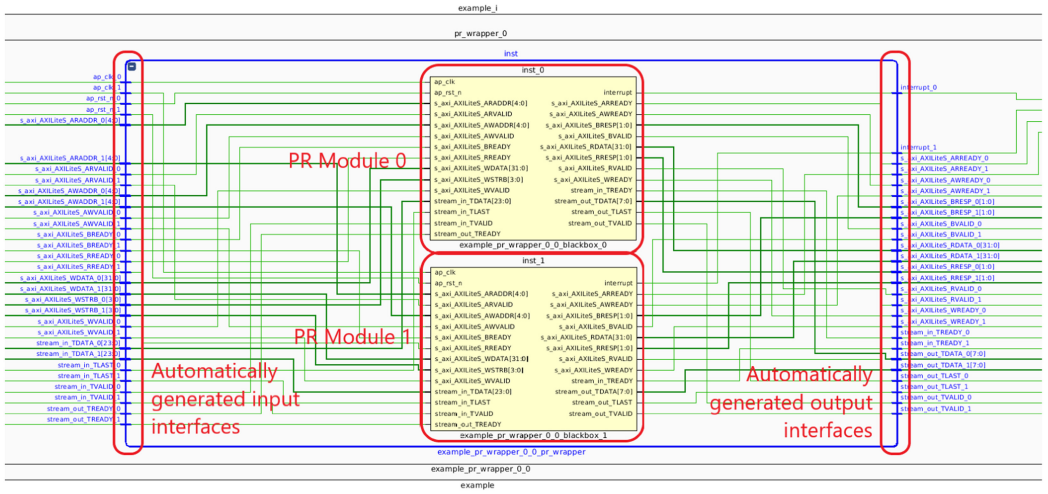

Fig. 6. Synthesis schematic after build tool generates wrappers for each PRR.

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.

4.2.4 Customised Base Design. Under the default settings, the base design will encapsulate the user's accelerator with a 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, as required for edge applications such as image processing and 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.

4.2.5 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. The interfaces for these primitives are shown in Figure 7.

The tools automatically determine the target device and deploy infrastructure accordingly. If targetting a Zynq device, then the ICAPE2 will be used at 100 MHz, and if a ZynqMP is selected,

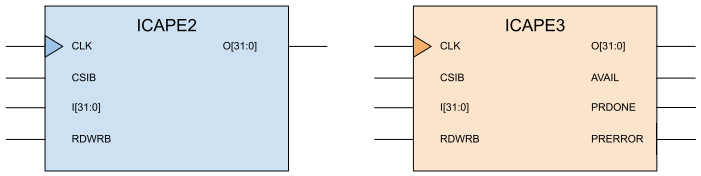

Fig. 7. ICAPE2 and ICAPE3 macros.

then the infrastructure builds for a 200 MHz clock and the ICAPE3 signalling. A unique DMA controller can be allocated for the ICAP as well as for each of the user's accelerators, at the cost of additional logical resources. When sharing a DMA controller, the tool will prioritise the accelerator clock frequency and clock the ICAP according to the slowest common clock, potentially reducing reconfiguration time if the accelerated region is running at a lower frequency than the ICAP. We use a Python toolbox for building digital hardware, nmigen [38], 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.

4.3 Linux Build Flow

We support building directly into a pre-prepared Linux image, 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.

4.3.1 PMU Firmware. 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 [45] 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 [43]. Due to this restriction, we build a modified version of the PMU firmware 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).

4.3.2 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.

The 3.18 release of the Linux kernel introduced the **device tree overlay (DTO)**, an implementation of the in-kernel device tree that can be used to modify the kernel's live tree and affect

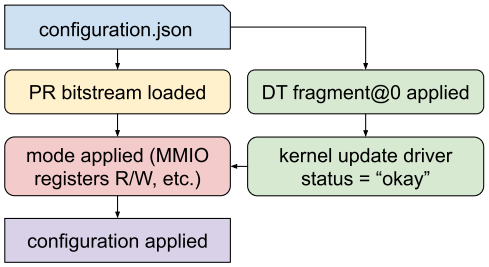

Fig. 8. Applying DT fragment via configuration.

```
1 axi_accel0: axi_accel@0 {
2     compatible = "linux,axi_accel";
3     status = "disabled";
4 };
5 fragment@0 {
6     target = <&axi_accel0>;
7     __overlay__ {
8         status = "okay";
9     };
10 };
```

Listing 3. Example of AXI DT fragement@0.

the running kernel, such as applying driver changes, registering and deregistering nodes, in turn loading/unloading modules. 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 FPGA build process or produce the overlays from the DFX workflow. The build tooling uses the data from the FPGA build process to generate custom DTOs, describing required modes and configurations that 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. DTO loading is crucial when applying configurations that require changes to nodes of the live device tree, for example, alerting a driver of the status of a hardware module, as shown in Listing 3 and Figure 8.

4.3.3 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 FPGA build process. Nodes are generated for all of the available MMIO (AXI) addresses as well as the position of any AXI-Stream interfaces, located under the DMA controller.

5 RUNTIME MANAGEMENT

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 that abstracts 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, and so on.

5.1 Configuration Abstraction

To simplify the user's perspective for controlling the state of the PL, we use the concept of modes and configurations mentioned in Section 2.

5.1.1 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.

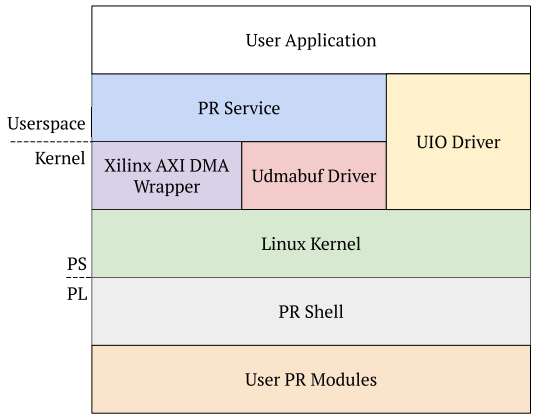

Fig. 9. ZyPR Linux Stack [8].

5.1.2 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 *configs* interface. For the PL, this can be used to dynamically load and unload hardware according to what is present 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. A benefit of using DTOs to manage system-wide configurations is that it retains its state and configuration, regardless of the application. If the user's application is paused or stopped, then the configuration should be maintained/stored, otherwise the designer must implement their own mechanisms to track memory addresses and configurations, outside of the application's runtime. Stopping the application should not unnecessarily unload or reload the FPGA; a DTO allows this to be tracked outside of the user's application.

5.1.3 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 a consistent interface that is 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. An alternative design pattern would be to build all the tooling as standalone loadable kernel modules, however, this presents the challenge of needing to support and provide compatibility for our own modules that would need to support Xilinx hardware such as the DMA controller. The Linux stack for the ZyPR runtime is shown in Figure 9.

5.1.3.1 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 ZyPR runtime gives the software developer abstracted access to these UIO registers, without requiring them to directly initialise and set up these modules, themselves. The ZyPR runtime manages the availability of these UIO addresses to prevent reading/writing to hardware, while it is in an undefined state. 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.

5.1.3.2 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 [14]. 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.

5.1.3.3 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 [22]. 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 supports transmit, receive, and 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 that 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 to 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 of the driver that is compatible with both the Zynq and ZynqMP and has been tested in PetaLinux 2019 (Linux Kernel 4.19.0). 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.

5.2 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 [35]. 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 ZyPR 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 saturation.

5.3 Configuration Manager

The runtime or **configuration manager (CM)** enables the designer to abstractly control the hardware modes and configurations and is a key feature of the framework. Figure 10 provides an

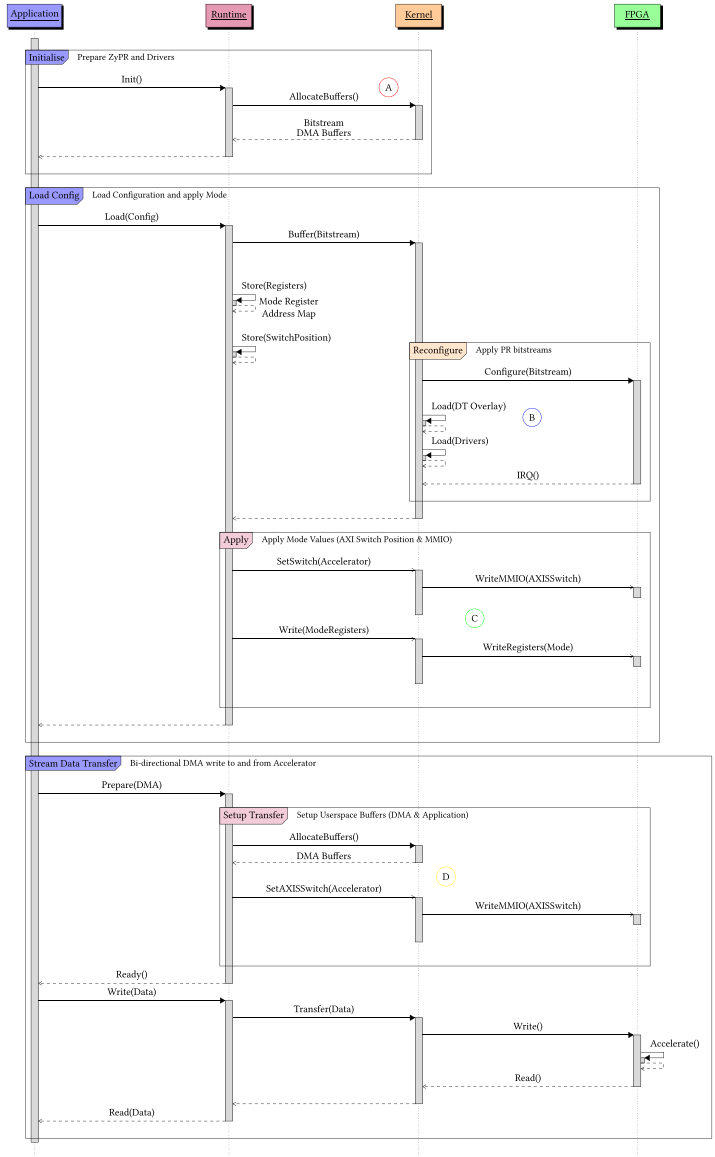

Fig. 10. Sequence diagram for the ZyPR runtime (loading and data transfer).

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 Figure 10, 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 allows 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.

5.4 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 intend to later release a Python library, enabling software designers to further abstract their applications as well as extend the Xilinx PYNQ platform to leverage abstractions. We use `z` for the ZyPR Manager and `s` for the configuration.

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

We evaluate the ZyPR 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

Table 3. 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)	7,688	5,132	5	7.27
2 AXI4-Lite (32-bit) + 0 AXI4-Stream (32-bit)	7,738	5,142	5	7.28
1 AXI4-Lite (32-bit) + 1 AXI4-Stream (32-bit)	8,817	5,619	5	7.96
2 AXI4-Lite (32-bit) + 2 AXI4-Stream (32-bit)	9,063	6,051	5	8.58

accelerator and software performance. The evaluation is performed on a ZynqMP development kit, the Ultra96v2 (Xilinx Zynq UltraScale+ MPSoC ZU3EG), 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.

6.1 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, then the tools will not generate an AXI-Stream switch.

6.1.1 Compile-time-generated Infrastructure. Here, we demonstrate a varying selection of custom accelerators with the respective interfaces that they expose and measure the resources consumed by the infrastructure generated required to support the described interfaces. Table 3 shows the resources across a selection of arrangements; the consumed resources never exceed 9% of the LUT utilisation of the PL. The infrastructure generated to house additional buses is a Xilinx AXI4 Lite multiplexer, where minimal resources are required to route additional AXI4 Lite buses. The same is true for the AXI-Stream interface, where ZyPR controller manages toggling between the source and destinations of the Read/Write transactions/streams. 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.

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) [41] and the AXI Interconnect (v2.2) [40] 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.

6.2 Accelerator Performance

To evaluate the impact of ZyPR'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 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

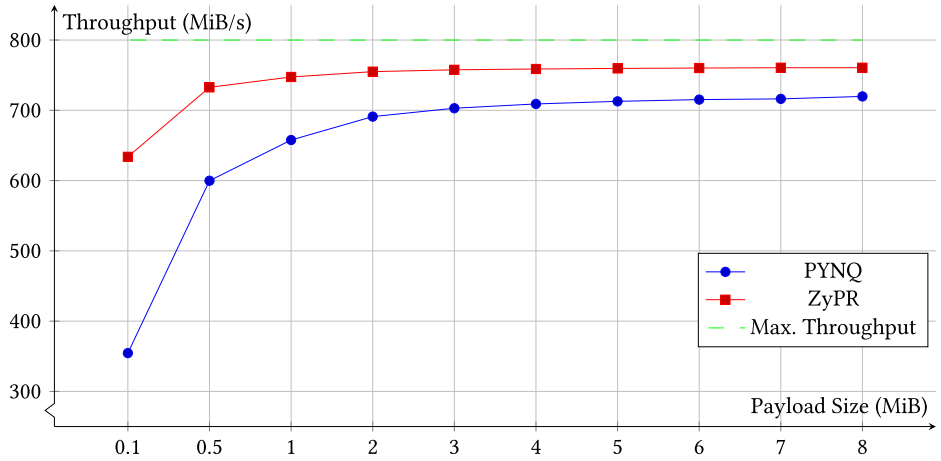

Fig. 11. DMA Driver Benchmark across 1,000 transfers (PL clocked at 200 MHz).

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 ZyPR consistently performs at higher throughput than the equivalent transfer in PYNQ, as shown in Figure 11, 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.

As our tools generate virtual memory maps and buffers for accelerators in the PL, these abstractions should have limited impact on the designer's ability to orchestrate hardware tasks from software. Unlike PYNQ, where an interpreted Python layer exists between the user's software and the hardware, we provide minimally impacted access to hardware resources from the ZyPR API. Due to the performant interface (DMA from pre-prepared contiguous memory buffers) provided between the processor and FPGA, the latency is kept low.

6.3 Software Overhead

To evaluate the performance impacts of the ZyPR runtime, including userspace abstractions, we quantify various portions of the PRR management abstraction.

6.3.1 Build Time Complexity. While it is difficult to quantify the impact of abstracting the build workflow, given the variation of efficiency due to designer's knowledge, development machine performance, among other variables, we provide a timed build run for the tools, generating four specified combinations of modules. The tools automatically assemble the build flow 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 Figure 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 three-configuration design with 1 AXI-Lite (slave) and 1 AXI-Stream (master and slave), port

Table 4. Runtime Latency Breakdown

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).

extraction takes **37.94** seconds, and the total build time is **2,427** 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, and so on. Comparatively, the overhead for the extensions to the workflow are negligible compared against the vendor-locked aspects such as synthesis and implementation.

6.3.2 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 configuration, such as with a mode that contains multiple MMIO reads/writes. Table 4 provides a breakdown for the software overhead (measured as latency) required by the runtime to load and trigger configurations and modes. This specifically highlights 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 so as not to impact the total time to perform a provision of a configuration. Notably, 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 than other tools. We argue that given the significant increase in time to load bitstreams versus FPGA Manager, 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 is important to note that time to parse JSON objects scales, depending on the complexity of the configurations and modes.

6.4 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 [13], 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 ZyPR PR controller.

6.4.1 Comparison to FPGA Manager. We evaluate the performance against Xilinx's provided FPGA manager tool, as shown in Figure 12. As previously discussed in Reference [8], the FPGA manager tool is verbose, so both a default (verbose) and silent version are compared against our runtime. In Reference [8], we showed the performance of the ICAPE3 running at 100 MHz;

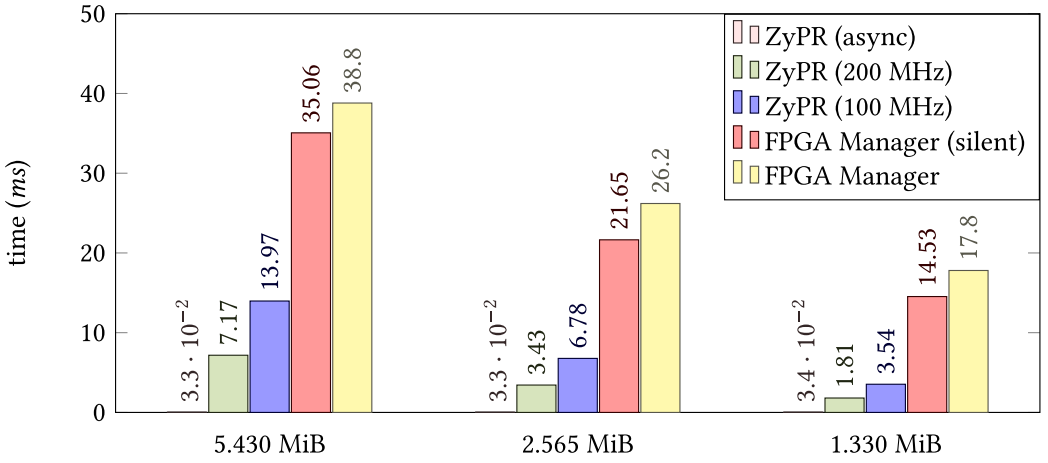

Fig. 12. PR Runtime Performance (time to load bitstream).

given that we are able to further clock the ICAPE3 macro at 200 MHz [44], performance is significantly improved against the traditional PCAP reconfiguration flow. We benchmark our runtime at 100 MHz and 200 MHz against Xilinx's FPGA manager in default and silent modes, using three 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 ZyPR 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 time the processor is free to begin applying the device tree fragments, establishing any accelerator specific buffers, and so on. The results demonstrate that the ZyPR runtime has a significant advantage over FPGA Manager, that increases with bitstream size as the time taken to trigger the DMA transaction is amortized over the whole 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).

7 CASE STUDY

We demonstrate the ZyPR 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. As our tools are aimed at users who may not be experts in RTL design, we choose to demonstrate a simple image-processing application built from HLS sources, as this is a likely choice for designers looking to generate complex accelerator logic from higher-level source code. Our tools consume the Verilog output files from generated by the HLS source, matching the ports and interfaces of the accelerator. While we do not suggest that our tools help with the design of the RTL accelerators themselves, the ability to utilise HLS enables users to more rapidly iterate on their applications.

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 three PRRs with three independent configurations; an initial histogram computation followed by two image-manipulation 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 [39], which offer OpenCV function acceleration for FPGAs. Using static reconfiguration and the FPGA Manager (limited to 256 MB/s), bitstream loading would consume significant time, resulting in dropped frames. The case study provides a demonstration of how accelerator chaining is relevant to image/video processing applications.

Interfaces extracted during the build flow are used to instruct the tools which interfaces on the accelerators should be connected. Figure 13 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) which then determines the chained configurations required to improve image quality. Fast reconfiguration during the video stream means the processing can be decided in real-time 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.

7.1 PR Region Data Chaining

To configure chaining, the designer can set the following parameters as shown in Listing 4:

```
{
    "pr_regions": {
        "region_a": {
            "pblock": [
                "SLICE_X36Y121:SLICE_X47Y155 DSP48E2_X3Y50:DSP48E2_X4Y61 ..."
            ],
            ...
        },
        "region_b": { ... },
        "region_c": { ... }
    },
    "pr_chains": [
        "region_a",
        "region_b",
        "region_c"
    ]
}
```

Listing 4. Enable PRR chaining.

Listing 4 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). Figure 14 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.

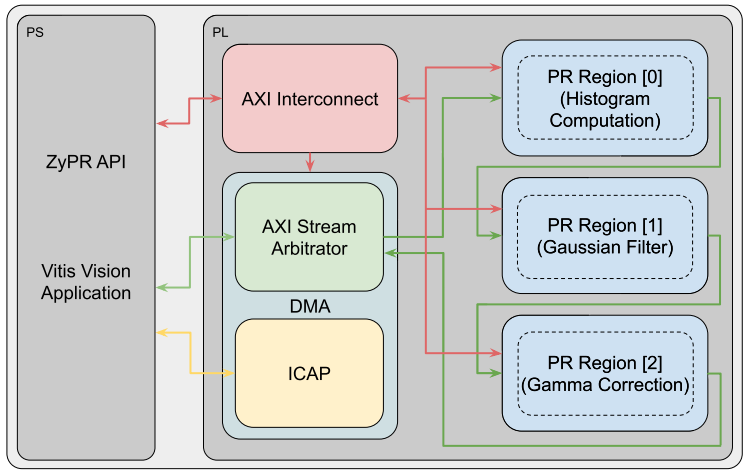

Fig. 13. Overview of HLS Vitis Vision chained accelerator demo.

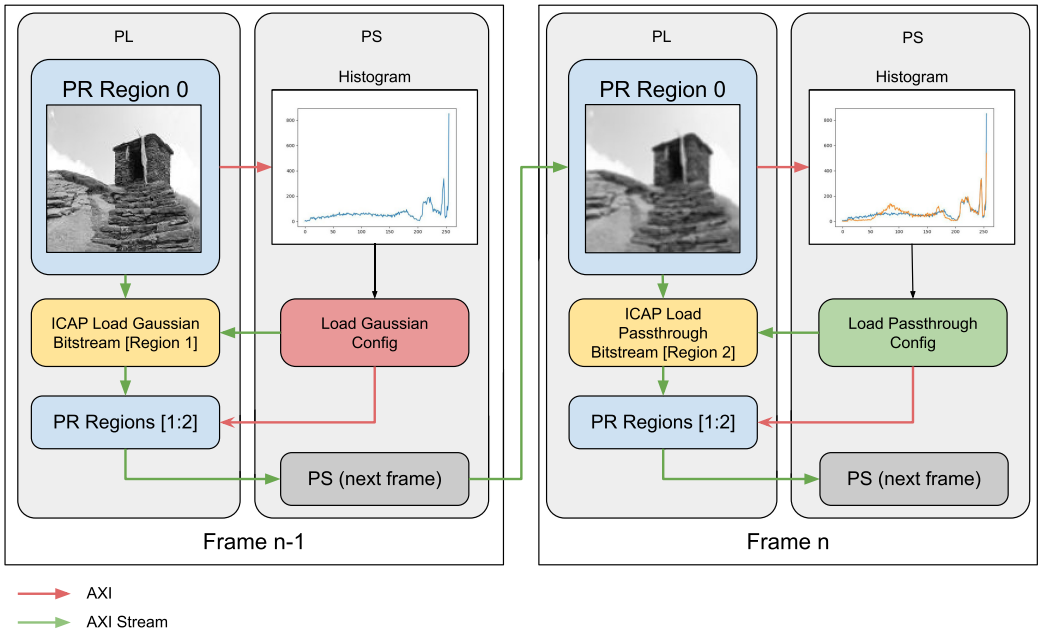

Fig. 14. PS uses histogram to determine accelerators to apply.

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).

7.2 Design Effort and Framework Impact

To manually develop the application described in the case study, the designer would have to undertake a series of manual steps, highlighted in 4. The initial stage of designing the accelerators themselves is beyond the scope of this framework, but 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 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 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 **2,427**-second implementation run within Vivado. While it is largely arbitrary to measure the impact of our tools in the scope of the entire workflow, it demonstrates that running our tools only contributes to 1.5% of the total build time. Provided that the user is only required to define a single config file, to generate a ready-to-go Linux image, this can be considered a significant advantage.

Comparing the case study project, for a user to design and develop the infrastructure to support three different PR regions with three different accelerators, the configuration file is 230 lines of JSON (including JSON syntax), where 38 lines are tooling setup. During build time, the tools generated approximately 1,706 **lines of code (LoC)**, including TCL scripts, Verilog wrappers, data stores (JSON), and memory maps (CSV). Additionally, this spans 36 different generated files and does not include the templates used by the tools to generate the project files, which are not stored in the case study build directory. While this is also a largely arbitrary measurement of complexity reduction, it demonstrates the complexity of developing an application and how much automation is performed by the tools. We expect to be able to even further reduce the LoC, as we could condense the module extraction process by pointing the tools at a directory and infer the source files to generate the accelerator cores.

7.3 Comparison to Existing Tools

The case study demonstration shows the webcam running at a resolution of 1,080 × 1,920 pixels at 30 frames per second. For a pBlock allocated 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. The pBlock is sized at 22.45%, 22.45%, 26.67% of the respective total available CLB LUTs, CLB Flip-Flops, DSPs on the ZU3EG ZynqMP device (Ultra96v2). 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 frame rate 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 [30] 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.

7.4 Runtime Application

The code in Listing 5 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. 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.

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. This is available to the runtime C++ API, requiring minimal understanding of the mechanisms required for provisioning configurations and modes.

```cpp
#include <zypr.h>

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

    Zypr 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 5. C++ API.

```json
"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. JSON Config and Mode.

8 CONCLUSION

In this article, we presented ZyPR: a toolflow for automating the build process of PR designs, from HDL through to a final ready-to-go Linux image supporting a custom runtime configuration manager. ZyPR takes the logic from user applications, extracts interfaces, and generates infrastructure at compile-time to support data transfer between the PS and PL. It hides the complexity of the PR build process and the handover to PetaLinux for Linux kernel compiling and file-system construction. ZyPR also incorporates a high-performance configuration manager that utilises the hardened ICAP 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 ZyPR runtime manages both PR and hardware control under abstractions for configurations and modes. We compared ZyPR's performance against Xilinx's own FPGA Manager driver used by other design frameworks and showed performance benefits as well as functional behaviours such as non-blocking DMA triggering, which free the processor to complete device tree overlay provisions and buffer generation while awaiting an interrupt to denote PR completion. We demonstrate how ZyPR can be used with a vision-processing case study that uses the build tools to generate a design for HLS-based accelerators chained together for tuning image quality.

Given the scope of this work and the breadth of the problem being addressed, it is useful to state what is presently unsupported

- Further simplification of writing RTL (Xilinx's Vitis HLS and others already exist)
- Support for non-Verilog modules (VHDL and SystemVerilog)
- 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 propose these features for future research as well as an intention to support building from the FuseSoC [18] IP library tool to allow for users to easily fetch packages from libraries and include 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. Furthermore, it would provide additional abstraction to integrate this workflow with Xilinx's PYNQ tools, implementing low-level optimisations under lightweight Python wrappers. Additionally, it would be advantageous to decouple the Linux build process from Xilinx's PetaLinux project, as maintaining this in an up-to-date manner is complex as development board, Vitis, PetaLinux, Yocto Layers, and Linux Kernel versions must all align to successfully build Linux images. The ZyPR framework is available at https://github.com/accl-kaust/zypr and the Python library at https://github.com/accl-kaust/interfacer as open source repositories for wider adoption and contribution by the community.

REFERENCES

[1] Adewale Adetomi, Godwin Enemali, Xabier Iturbe, Tughrul Arslan, and Didier Keymeulen. 2018. R3TOS-based integrated modular space avionics for on-board real-time data processing. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS'18)*.

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

[3] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. 2009. Performance comparison of FPGA, GPU and CPU in image processing. In *International Conference on Field Programmable Logic and Applications (FPL'09)*. 126–131.

[4] Christian Beckhoff, Dirk Koch, and Jim Torresen. 2012. Go Ahead: A partial reconfiguration framework. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*. 37–44.

[5] Christian Beckhoff, Dirk Koch, and Jim Torreson. 2013. Automatic floorplanning and interface synthesis of island style reconfigurable systems with GoAhead. In *International Conference on Architecture of Computing Systems*. Springer, 303–316.

[6] Alex R. Bucknall and Suhaib A. Fahmy. 2021. Runtime Abstraction for Autonomous Adaptive Systems on Reconfigurable Hardware. In *Design, Automation Test in Europe Conference Exhibition (DATE'21)*. 1616–1621.

[7] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. 2019. Network enabled partial reconfiguration for distributed FPGA edge acceleration. In *International Conference on Field-Programmable Technology (FPT'19)*. 259–262.

[8] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. 2020. Build automation and runtime abstraction for partial reconfiguration on Xilinx Zynq UltraScale+. In *International Conference on Field-Programmable Technology (FPT'20)*. 215–220.

[9] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. FPGAs in the cloud: Booting virtualized hardware accelerators with openstack. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'14)*. 109–116.

[10] Norbert Deak, Octavian Cret, and Horia Hedesiu. 2019. Efficient FPGA floorplanning for partial reconfiguration-based applications. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'19)*. 309–309.

[11] François Duhem, Fabrice Muller, and Philippe Lorenzini. 2011. FaRM: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA. In *International Symposium on Applied Reconfigurable Computing*. Springer, 253–260.

[12] Suhaib A. Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA accelerators for efficient cloud computing. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom'15)*. 430–435.

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

[14] Kawazome Ichiro. 2015. u-dma-buf. Retrieved from https://github.com/ikwzm/udmabuf.
[15] Florian Kästner, Benedikt Janßen, Frederik Kautz, Michael Hübner, and Giulio Corradi. 2018. Hardware/Software codesign for convolutional neural networks exploiting dynamic partial reconfiguration on PYNQ. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*. 154–161. DOI: https://doi.org/10.1109/IPDPSW.2018.00031
[16] Nadir Khan, Jorge Castro-Godínez, Shixiang Xue, Jörg Henkel, and Jürgen Becker. 2020. Automatic floorplanning and standalone generation of bitstream-level IP cores. *IEEE Trans. Very Large Scale Integ. Syst.* 29, 1 (2020), 38–50.
[17] Olof Kindgren. 2018. Edalize: Python Library for Interacting with EDA Tools. Retrieved from: https://github.com/olofk/edalize.
[18] Olof Kindgren. 2018. FuseSoC: Package Manager and Build Abstraction Tool for FPGA/ASIC Development. Retrieved from: https://github.com/olofk/fusesoc.
[19] Christian Lienen, Marco Platzner, and Bernhard Rinner. 2020. ReconROS: Flexible hardware acceleration for ROS2 applications. In *International Conference on Field-Programmable Technology (FPT'20)*. 268–276.
[20] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. 2010. *Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller*. Technical Report MSR-TR-2009- 150, Microsoft Research.
[21] David Ojika, Ann Gordon-Ross, Herman Lam, and Bhavesh Patel. 2019. FaaM: FPGA-as-a-Microservice—A case study for data compression. In *International Conference on Computing in High Energy and Nuclear Physics*.
[22] Brendan Perez and Jared Choi. 2015. Xilinx AXI DMA Linux Driver. Retrieved from: https://github.com/bperez77/xilinx_axidma.
[23] Khoa Pham, Dirk Koch, Anuj Vaishnav, Konstantinos Georgopoulos, Pavlos Malakonakis, Aggelos Ioannou, and Iakovos Mavroidis. 2020. Moving compute towards data in heterogeneous multi-FPGA clusters using partial reconfiguration and I/O virtualisation. In *International Conference on Field-Programmable Technology (FPT'20)*. 221–226.
[24] Khoa Dang Pham, Edson Horta, and Dirk Koch. 2017. BITMAN: A tool and API for FPGA bitstream manipulations. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'17)*. 894–897.
[25] Ariel Podlubne, Julian Haase, Lester Kalms, Gökhan Akgün, Muhammad Ali, Habib Ulhasan Khar, Ahmed Kamal, and Diana Göhringer. 2018. Low power image processing applications on FPGAs using dynamic voltage scaling and partial reconfiguration. In *Conference on Design and Architectures for Signal and Image Processing (DASIP'18)*. 64–69.
[26] Marco Rabozzi, Gianluca Carlo Durelli, Antonio Miele, John Lillis, and Marco Domenico Santambrogio. 2017. Floorplanning automation for partial-reconfigurable FPGAs via feasible placements generation. *IEEE Trans. Very Large Scale Integ. Syst.* 25, 1 (2017), 151–164.
[27] Alfonso Rodríguez, Juan Valverde, Jorge Portilla, Andrés Otero, Teresa Riesgo, and Eduardo de la Torre. 2018. FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The ARTICo^3 framework. *Sensors* 18, 6 (2018). Retrieved from: https://www.mdpi.com/1424-8220/18/6/1877.
[28] Siva Satyendra Sahoo, Tuan D. A. Nguyen, Bharadwaj Veeravalli, and Akash Kumar. 2019. Multi-objective design space exploration for system partitioning of FPGA-based dynamic partially reconfigurable systems. *Integration* 67 (2019), 95–107.
[29] Shanker Shreejith, Ryan A. Cooke, and Suhaib A. Fahmy. 2018. A smart network interface approach for distributed applications on Xilinx Zynq SoCs. In *International Conference on Field Programmable Logic and Applications (FPL'18)*.
[30] Shanker Shreejith, Kizhepatt Vipin, Suhaib A. Fahmy, and Martin Lukasiewycz. 2013. An approach for redundancy in FlexRay networks using FPGA partial reconfiguration. In *Design, Automation Test in Europe Conference and Exhibition (DATE'13)*. 721–724.
[31] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A Python-based hardware design processing toolkit for Verilog HDL. In *Applied Reconfigurable Computing (Lecture Notes in Computer Science)*, Vol. 9040. Springer International Publishing, 451–460.
[32] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. 2020. FOS: A modular FPGA operating system for dynamic workloads. *ACM Trans. Reconfig. Technol. Syst.* 13, 4 (Sept. 2020).
[33] Kizheppatt Vipin and Suhaib A Fahmy. 2012. Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In *International Symposium on Applied Reconfigurable Computing*. Springer, 13–25.
[34] Kizheppatt Vipin and Suhaib A. Fahmy. 2014. Automated partial reconfiguration design for adaptive systems with CoPR for Zynq. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'14)*. 202–205.
[35] Kizheppatt Vipin and Suhaib A. Fahmy. 2014. ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq. *IEEE Embed. Syst. Lett.* 6, 3 (2014), 41–44.
[36] Kizheppatt Vipin and Suhaib A. Fahmy. 2018. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *Comput. Surv.* 51, 4 (2018).
[37] Chao Wang, Lei Gong, Xi Li, and Xuehai Zhou. 2020. A Ubiquitous machine learning accelerator with automatic parallelization on FPGA. *IEEE Trans. Parallel Distrib. Syst.* 31, 10 (2020), 2346–2359.

[38] Whitequark. 2018. nmigen. Retrieved from: https://github.com/nmigen/nmigen.
[39] Xilinx. 2020. Retrieved from: https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-vision.html.
[40] Xilinx Inc. 2017. *PG059: AXI Interconnect v2.1.* Xilinx Inc. v2.1.
[41] Xilinx Inc. 2018. *PG085: AXI4-Stream Infrastructure IP Suite v3.0.* Xilinx Inc. v3.0.
[42] Xilinx Inc. 2020. *Python Productivity for Zynq.* Xilinx Inc. http://pynq.io/.
[43] Xilinx Inc. 2020. *UG1137: Zynq UltraScale+ MPSoC: Software Developers Guide.* Xilinx Inc. v12.0.
[44] Xilinx Inc. 2020. *UG909: Dynamic Function eXchange v2019.2.* Xilinx Inc. v2019.2.
[45] Xilinx Inc. 2021. *PG116: MicroBlaze Micro Controller System v3.0.* Xilinx Inc. v3.0.
[46] Rafael Zamacola, Alberto Garcia Martinez, Javier Mora, Andres Otero, and Eduardo de La Torre. 2018. IMPRESS: Automated tool for the implementation of highly flexible partial reconfigurable systems with Xilinx Vivado. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig'18).*
[47] Shuaiqing Zhi, Yani Cui, Jiaxian Deng, and Wencai Du. 2020. An FPGA-based simple RGB-HSI space conversion algorithm for hardware image processing. *IEEE Access* 8 (2020), 173838–173853. DOI: https://doi.org/10.1109/ACCESS.2020.3026189
[48] Zongwei Zhu, Junneng Zhang, Jinjin Zhao, Jing Cao, Duan Zhao, Gangyong Jia, and Qingyong Meng. 2019. A hardware and software task-scheduling framework based on CPU+FPGA heterogeneous architecture in edge computing. *IEEE Access* 7 (2019), 148975–148988. DOI: https://doi.org/10.1109/ACCESS.2019.2943179

Received 21 July 2022; revised 31 January 2023; accepted 10 February 2023