Build Automation and Runtime Abstraction for Partial Reconfiguration on Xilinx Zynq UltraScale+

Alex R. Bucknall*, Shanker Shreejith†, Suhaib A. Fahmy*
*School of Engineering, University of Warwick, Coventry, UK
†Department of Electronic and Electrical Engineering, Trinity College Dublin, Ireland

***Abstract*—Partial reconfiguration (PR) is fundamental to building adaptive systems on modern FPGA SoCs, where hardware can be adapted dynamically at runtime. Vendor supported reconfiguration is performance limited, drivers entail complex memory management, and software/hardware design requires detailed knowledge of the underlying hardware. This paper presents a collection of abstractions that provide high performance reconfiguration of hardware from within the Linux userspace, automating the process of building PR applications, and adding support for the Xilinx Zynq UltraScale+ architecture. We compare our abstractions against vendor tooling for PR management and open source tools supporting PR within Linux. Our tools provides automation and abstraction layers, from defining PR configurations through to compiling and packaging Linux with support for userspace PR control, targeted for non-experts.**

I. Introduction and Background

FPGAs enable efficient acceleration of compute-intensive and latency-sensitive tasks through custom datapaths that exploit algorithmic parallelism. Partial Reconfiguration (PR) is a well established feature of FPGAs, allowing portions of the hardware fabric to be modified at runtime, permitting different accelerators to be loaded into pre-defined regions. This allows for the implementation of adaptive systems, where accelerators are loaded into hardware on-demand at runtime. Dynamic accelerator loading reduces the area (and power) requirements of such adaptive systems as only the active accelerators consume resources. While a variety of FPGA systems using PR have been demonstrated [1], [2], these implementations are often bespoke designs, requiring significant FPGA expertise to exploit PR as well as interface to and from software. More recently, PR has found renewed use in abstracting interfaces for accelerators in cloud datecenters, allowing a host server to load accelerators onto an FPGA while keeping the PCI Express interface active. However, such a controlled environment usually has fixed hardware/software infrastructure and does not allow interfacing with peripherals. Reconfiguration performance is also usually less important. Adapting similar approaches to dynamic accelerators for edge computing is an active area of research, and presents additional challenges in terms of lightweight abstractions.

The tightly coupled Arm based processing system (PS) and FPGA programmable logic (PL) in modern FPGA SoCs represent an ideal computing platform for combining software programmability and management with accelerator hardware. These systems can exploit the standard peripherals available in the PS, such as networking and control interfaces, as well as utilise custom PL hardware. To enhance software productivity, operating systems such as Linux are often deployed on the PS, requiring suitable infrastructure to manage reconfiguration and the interfaces between the PS and PL.

While there has been a strong body of research focusing on FPGA PR to load accelerated tasks within a software system generically, and more recently for multi-user systems, the more complex challenges presented by an independent adaptive system with custom PL peripherals have yet to be explored in detail. This presents a significant barrier to entry for designing combined software/hardware systems on FPGA SoCs. The use of FPGA PR has been demonstrated in applications across automotive [3], image/signal processing [4], [5], and space applications [6] among others. However, the design approach has been ad-hoc and requires significant hardware and software expertise. While FPGA vendors provide limited support for PR, through design flows and low-level drivers, this is not sufficiently abstracted for use by non-experts and often exhibits poor performance. These problems become more pronounced when layering a software platform on top of an operating system like Linux, where layers such as the filesystem, scheduler, and kernel add further latency between application and accelerator.

To clarify the discussion of PR abstraction used in this paper, we define a set of PR *regions* as labels, without any definition of region location or size. A user then defines multiple *configurations* of their system, each representing one possible state. Within each configuration an additional abstraction of *modes* is supported to allow a differentiation between large scale structural changes and smaller differences in individual modules used, such as memory addressing. For each mode, a set of hardware *modules* is attached to each *region*.

Managing PR accelerators can be done in software or hardware. A variety of hardware controllers for PR were proposed in the literature [7]. For Xilinx devices, the Internal Configuration Access Port (ICAP) can be used to allow PR control from hardware. To control reconfiguration, an external interface can pass bitstreams to the ICAP, or alternatively, a soft processor may used from the PL to manage this [8].

With the Xilinx Zynq SoC, the Processor Configuration Access Port (PCAP) and accompanying software library were introduced. This is a PS interface for managing reconfiguration of Xilinx FPGA SoCs and can also be used to partially reconfigure. While this interface and the vendor build flow offer some abstraction, using both require the designer to have detailed understanding of the PR process and that only PCAP

is supported by the software library.

Although ICAP offers considerably higher throughput and lower latency [9] than PCAP, it requires custom control hardware and software. Some frameworks have proposed abstractions for the ICAP but these typically do not support a Linux runtime or the newer Zynq UltraScale+ (ZynqMP) [10].

Presently Xilinx offers a Linux driver called FPGA Manager, a hardware agnostic driver that manages loading of static and PR bitstreams. FPGA Manager, offers an abstraction to load bitstreams from the filesystem over PCAP but provides limited support for high performance transfers or abstracting bitstream management complexities, such as caching bitstreams in DRAM, and offers no support for the ICAP on the Zynq or ZynqMP [11]. Additionally, loading bitstreams uncached from the Linux filesystem, rather than DRAM, is inefficient and slow.

While FPGA Manager abstracts managing memory addressing of the PR bitstreams, it requires the user to manually load PR device tree overlays (DTO) and required accelerator drivers. DTOs are used to provide hardware awareness to Linux and may be loaded/unloaded at runtime to alert the kernel to a new PL accelerator provisioned with PR. High performance applications may use multiple PR bitstreams stored in memory, increasing the complexity of addressing memory from Linux's virtual memory mapping, for optimal performance.

On the ZynqMP loading of bitstreams using FPGA Manager is a blocking process, stalling the PS until direct memory access (DMA) to the Configuration and Security Unit (CSU) completes. The ZynqMP implementation of the driver performs PR by first allocating the required memory for the bitstream, issuing a request to the *Embedded Energy Management Interface* API for access to the *Arm Trusted Firmware* and upon success, passes the bitstream to the *Platform Management Unit* (PMU) which loads the PCAP using the *Xilfpga* library [12].

By default, the driver outputs to the console (kernel log), adding a magnitude of delay to the PR time. These factors all render this driver a poor choice for adaptive systems, where the PS is expected to perform computation while managing PR. Additionally, programming the PL is managed differently within the Zynq and ZynqMP architectures and this must be handled at build time.

As a PR build flow, Xilinx offers the PR Wizard (Vivado 2018.2 onwards), renamed to *Dynamic Function eXchange* (DFX) in 2019.1 [13]. This workflow imposes a sequence of complex manual steps requiring specialised knowledge to output PR bitstreams. Initially the designer must synthesize a static wrapper for each of the PR modules, instantiated as black boxes to preserve I/O buffers. Next each PR module must be synthesised with identical interfaces to the wrapper, in out-of-context mode to disable buffer insertion. Each of the PR modules is then saved as a design checkpoint (DCP) for implementation. The designer must then configure a unique build run for each combination of PR modules by linking them, loading design constraints and running the optimisation, place and route for each combination.

The designer must initially preserve a base static implementation by issuing the design run for the initial combination of modules and locking the design routing. Each PR module must then be added to the static design, assigned to a collection of logical resources, known as a *pblock*, and implemented to save a new configuration for a routed design, repeating for each PR module. This is a convoluted build process for a non-expert, even before considering the DTO, custom drivers, and application layers needed for use within Linux. Xilinx's PetaLinux tool can then be used to import hardware and build Linux images for the target platform, however it cannot manage generating device tree (DT) fragments for PR or generate PR specific drivers.

Many steps can be automated, however altering the design typically requires re-running the entire workflow. For example, in Xilinx's SDSoC framework, which provides high level synthesis (in C/C++/OpenCL) [14], changes to the design require hardware re-synthesis and recompilation of the kernel.

In this paper, we present a collection of tool abstractions to simplify the PR design process for integrating FPGA SoCs within Linux. These tools allow for various design automations by extracting details from user PR configurations, generating required wrappers and bitstreams as well as providing a runtime for software control that includes Linux DTO support for dynamic peripheral loading. The low-overhead runtime API manages non-blocking reconfiguration requests within the userspace, abstracting data movement between software and accelerator. Finally, they offer a boot-time configuration for the kernel and memory map, ensuring low latency reconfiguration can be achieved. These tools are compatible with Zynq and ZynqMP architectures, masking low-level PR implementation details. We compare these abstractions against the Xilinx Vivado PR design flow as well as other open source build and runtime tools.

II. Related Work

In this section we outline related tool and runtime abstractions for PR to differentiate our contributions.

Early work such as [15], [16], and [17] automate and abstract tasks such as floorplanning, generating board constraints, and integration of PR modules for the discontinued Xilinx ISE tool. Frameworks such as [18] automated some manual steps such as partitioning and limited floorplanning, however these frameworks do not support Vivado, the ZynqMP, or deployment on embedded Linux.

PR frameworks such as [19], use lightweight shell interfaces aimed at on-demand acceleration rather than adaptive systems, requiring that users conform their logic to the shell specification, trading off PL resources for standardized partially reconfigurable regions (PRR). Limited recent research has focused on improving the build flow; however most work targets the deployment of PR modules rather than the design process, driven by the wider adoption of PR for accelerator platforms.

FOS [20] modularises the PR design flow for the Zynq and ZynqMP. It abstracts the flow into discrete development stages to offload complexity for designers with domain expertise. Under the standard vendor flow, the designer must be aware

of non-trivial considerations for PR hardware such as PL device constraints, generating a custom Linux DT, as well as interfacing high level software applications with hardware accelerators via kernel drivers or userspace abstractions. FOS supports flexibly compiling PR modules independent of the shell interfaces, using bitstream manipulation to assign PL fabric mapping to the modules, abstracting resource allocation [21], unlike the traditional flow which requires first locking a static region. However, custom shell logic is still required to interface directly with external PL peripherals. In FOS, the scheduler and bitstream manipulation are controlled from a software daemon that abstracts single and multi-user accelerators. While FOS provides both build and runtime management abstractions, domain expertise is still required for the build process and PR latency is higher due to it building on the Xilinx drivers that use the PCAP. It utilises a runtime resource elastic scheduler designed for arbitrating accelerator resources within the PL for single and multi-tenant environments, and does not consider a peripheral rich system as might be required for adaptive systems.

To allow for hardware access from within Linux, build tools must be able to generate configurations for kernel drivers and DTOs. Frameworks such as ReConOS [22] take a bespoke approach to OS support, requiring custom kernel drivers, shell interfaces, and build scaffolding. ReConOS manages the loading and scheduling of PR bitstreams, including a custom Linux operating system specifically designed to delegate hardware as threaded resources, but is limited to the original Xilinx Zynq. ReConOS enables generation of PR bitstreams but is tightly coupled to the kernel as custom drivers and bespoke RTL infrastructure; thus any PR modules added post-build, require the kernel to be recompiled. Additionally, configuration DTOs are not generated, so modules that require specific configurations, such as unique drivers, require the user to manage their own DTOs.

In an adaptive system, the cognitive functions that control adaptation are best implemented in software, and hence runtime PR management is important. [23] is one such approach to runtime management comprising a soft coprocessor in the PL for managing PR, however it is only capable of static PR as the soft processor retrieves its state from the main processor. To abstract the complexity of software to hardware reconfiguration, PR controllers such as ZyCAP [10] have been developed for the PL along with accompanying software in the PS. ZyCAP provides a high performance configuration hardware interface and software abstraction that hides the details of provisioning from the processor. The runtime manages caching filesystem bitstreams into external memory as well as loading bitstreams through DMA. However, such approaches are implemented as bare metal APIs, currently without Linux support for the ZynqMP.

Most Zynq and all current ZynqMP PR runtimes use Xilinx's supplied FPGA Manager driver, which abstracts the PCAP interface for loading PR modules, hence suffering higher PR latency than is achievable using the ICAP.

While existing PR frameworks provide access to PR with the compromise of overhead, portability, and performance, we propose 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. Our 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 our tooling abstractions.

III. PR Abstraction and Automation

Our tools provide PR build and runtime management abstractions across both Xilinx Zynq and ZynqMP device families. They consist of various build automations for generating PR bitstreams as well as providing a runtime service for managing, caching, and controlling PR with minimal latency. The tools provide a workflow for generating static wrappers and PR bitstreams, exporting to a Linux build flow, using the generated DTOs to configure the kernel with required drivers for userspace PR. The runtime service manages high speed, non-blocking loading of PR configurations while exposing a high level userspace API to application designers. The tools integrate with Vivado 2018.3 and will be updated to support future versions.

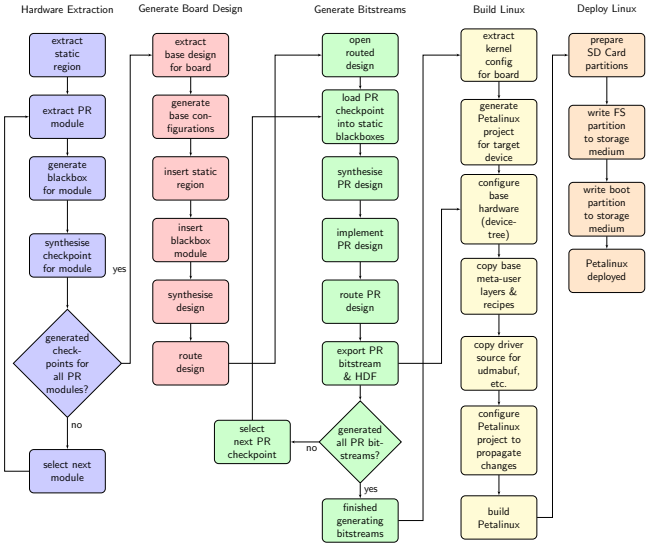

Fig. 1. Stages of the PR build flow.

A. Build Tools

The build tools take a set of user configurations and source files, specified in a config file, extract the PR module(s) and generate a collection of PR bitstreams and DTOs corresponding to each configuration. It synthesises the modules sequentially into a series of DCP files, built around a board specific base design, including any required static interfaces for peripherals such as cameras, Ethernet controllers, etc. Fig. 1 outlines the typical workflow for Linux PR images.

1) PetaLinux Support: After each PRR run, the tools produce a hardware export file *.hdf* that allows the Xilinx SDK to generate a custom DT, along with dynamically generating DTOs for PR modes. To allow for multiple PR interfaces within DTOs, such as AXI Stream and AXI Lite, we built a tool that extracts interfaces exposed to the static design and generates the DTOs accordingly. The *.hdf* file for the default configuration is used to generate a base PetaLinux device tree. At Linux compilation, bitstreams are packaged with the runtime PR manager; by default, the total size of the bitstreams is used to allocate PR memory buffers.

2) Partial Reconfiguration Hierarchy: Given resource availability, multiple PRRs may be instantiated and grouped as configurations and modes. Each hardware module's structure is parsed and parameters and ports are extracted to group interfaces (e.g. AXI and AXIS); later used to generate the structural buses between the PS and module. The user specifies the top file/module in their RTL and any desired PR modules, which are extracted and generated into PR compatible versions. Unlike other PR build abstractions, we generate shell interfaces at build time, according to the exposed interfaces in the modules, as seen in Fig. 2. This provides more flexibility for custom hardware rather than conforming to defined shell interfaces, however PR modules must use consistent interfaces across modes.

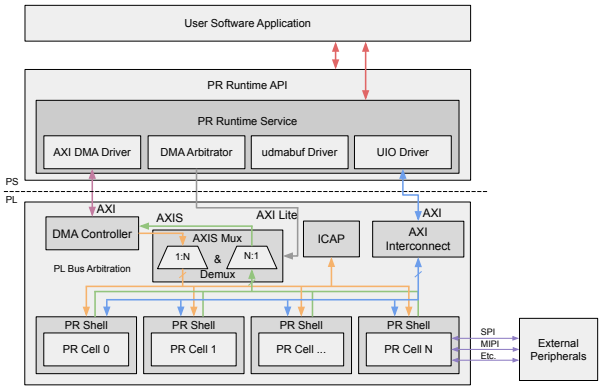

Fig. 2. Example of our hardware architecture.

3) Extensible Build Structure: The tool takes the hierarchy specified by the config file and generates the appropriate PR configurations. It allows for easily extending support to additional base designs, architectures (Zynq and ZynqMP), multiple PRRs as well as custom DTOs. Additional development boards can then be added by including the required constraint files, TCL scripts for the base design and any specific drivers/recipes for Petalinux, such as networking drivers. The hierarchical generation of PR modules enables the tool to re-use static DCPs, allowing further PR configurations to be generated, post-build. The tool uses existing static routing to ensure that changes are compatible with existing bitstreams; PR modules that do not use the maximum number of interfaces available have these interfaces tied-off and unloaded via DTO. Currently, our abstraction supports building PR modules from Verilog and Xilinx IP core sources.

4) ZynqMP PMU: Previous PR controllers have utilised DMA to load and trigger PR events, however they were constrained to only the Zynq and used bare metal APIs. Our tools support both the Zynq and ZynqMP architectures, where the tools abstract the specific target architecture, handling the differences automatically. For the ZynqMP, the PR software must request ICAP access by writing to the *pcap_ctrl* register in the CSU [13], managed by the PMU firmware. The default Xilinx PMU firmware for the ZynqMP, restricts access to the CSU and requires elevated security privileges; we bypass this with a custom PMU firmware that enables access. After setup, the CSU is no longer required to control the ICAP. The change from the ICAPE2 primitive on the Zynq to the ICAPE3 on the ZynqMP is also handled within the build process. ICAPE3 supports PR at higher frequencies and provides output status unlike the ICAPE2 [24]. We have tested our system at 100 MHz and will investigate over-clocking in future work.

B. Runtime Service

The runtime service executes entirely from userspace to utilise the multitude of software libraries available, unlike the restrictive nature of kernel drivers. This has the advantage of being independent of kernel configuration, reducing the security risks of providing direct hardware control to the user and allowing for the reliability of existing upstream vendor drivers as opposed to custom drivers. The PR manager runs as a Linux service, allowing the user to attach their application and trigger PR from an API that exposes defined configurations. The software abstraction can be seen in Fig. 3.

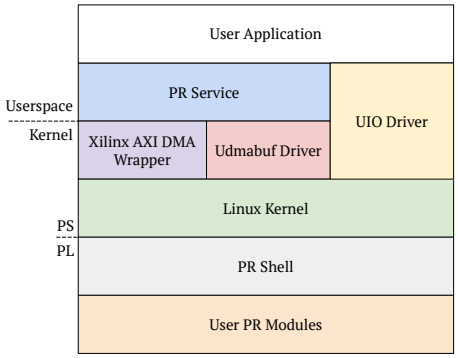

Fig. 3. Linux software abstraction.

1) Reconfiguration Manager: Our PR manager provides significant performance gains over current PR management tools. It uses a DMA controller in the PL to load PR bitstreams at near peak theoretical performance from external DRAM into the ICAP interface. Our hardware PR manager builds on the ZyCAP manager [10], using both a configuration (AXI4-Lite) and transfer (AXIS) interface to connect between the PL and the PS. The service abstracts managing bitstream and DMA memory buffers, generating and resizing as required.

We provide a C API to encapsulate the software service, which can be wrapped for languages such as Python. This API abstracts bitstream and accelerator memory addressing, allowing the user's software to call buffers at runtime. The service handles loading of DTO fragments with bitstream provisioning such that mode settings are applied concurrently. If a target bitstream is not currently cached in DRAM, e.g. it was added post-build, the user can cache/remove additional

bitstreams from the userspace using the *LoadBitstream()* API call. The runtime is non-blocking as we use DMA to load the ICAP in the PL, allowing software applications to continue, with an interrupt raised at completion. Xilinx's FPGA Manager driver forces blocking until PCAP is complete, must be polled for status, and cannot cache bitstreams from the filesystem.

2) Bitstream Caching: For high speed reconfiguration, Linux's virtual memory addressing must be considered. Virtual memory is used by the kernel as an expandable buffer for files between the disk and physical memory, typically DRAM. The kernel can allocate file buffers to a location mapped on disk known as the swap. The kernel uses swap as a means to temporarily store unused data (freeing memory) until it is required when it is reloaded back into main memory. Virtual addresses can map to physical addresses on DRAM, however this process is invisible to the user and it is significantly slower to read/write from disk than from DRAM. For high performance PR, bitstreams should be located in physical memory, yet be addressable from a virtual address map in the userspace. At initialization, our runtime prepares buffers for PR bitstreams and user accelerators, using a contiguous memory allocation driver, *udmabuf* [25]. The FPGA Manager runtime is limited in performance as bitstreams are stored on disk rather than in memory.

3) DMA Controller: A DMA controller is used to move PR bitstreams into the ICAP and may be shared by the PL accelerators, reducing resource consumption and removing the need for accelerator specific controllers. The controller's memory map to stream and stream to memory map interfaces are connected to AXIS interfaces on PR accelerators by a build time generated AXIS multiplexer/demultiplexer. The number of AXIS interfaces is extracted from the required number of streams in the user's PR modules, shown in Fig. 2. The runtime toggles between the interfaces to the ICAP and the accelerators, as needed. An AXI interconnect is generated for any accelerators that require it, along with memory maps in the DTO and abstracted by the runtime API, thus exposing read/write locations to accelerators. To manage this with existing tools, the user would have to create bespoke control hardware and software.

4) udmabuf Driver: To manage memory buffers, we use an open source kernel module known as the userspace mappable DMA buffer (*udmabuf*) [25]. This allows for the allocation of contiguous physical memory blocks within the kernel that can be used as DMA buffers, mapped to virtual addresses and available in the userspace. This permits the runtime to move data from userspace into memory and to the PR controller and/or accelerators within the PL. These DMA buffers are used in conjunction with the userspace I/O (UIO) driver for AXI memory transactions between the PS and PL.

5) Userspace IO Driver: The runtime utilises the Linux UIO driver to abstract memory transactions over AXI interfaces to the PL and is used in the DT to expose PL AXI interfaces to the userspace. It provides a generic software interface for PR accelerator memory maps and hardware interrupts. This allows the runtime to handle PR changes to the accelerators without requiring custom drivers for each configuration/mode. While Xilinx's SDK DT generator can populate the DT with UIO drivers for static designs, it cannot generate this for PR modules. Our build tools extract AXI devices from the PR configurations/modes and populate DTOs with UIO fragments.

TABLE I
EXAMPLE CUMULATIVE SETUP TIME.

Parameter	Design	Time (μs)	Size (MiB)
Configure CSU	T_{config}	5,576	N/A
Bitstream Buffer	$T_{bitstream}$	1,277	1.08
Accelerator Buffer	T_{accel}	3,785	7.63
Total	T_{total}	10,638	N/A

TABLE II
PR MANAGER STATIC PL RESOURCES.

Conf.	FFs	LUTs	BRAMs	% of PL
1 AXI4 + 1 AXI4-S	8817	5619	2	7.96
3 AXI4 + 2 AXI4-S	8887	5698	2	8.08

6) AXI DMA Driver: Xilinx's vendor DMA driver provides kernel access to the PL DMA controller but does not expose an interface that is available from the userspace. This forces the user to build their software as a kernel module and inject or compile it into the kernel. Instead, our tools use an open source userspace wrapper to control Xilinx's DMA controller [26]. This allows for compatibility with the vendor supplied DMA driver, which reduces the need for kernel maintenance unless breaking changes occur upstream from the vendor. Although this driver can use contiguous physical memory allocation, we use the *udmabuf* driver for dynamic buffer resizing. *udmabuf* is used in conjunction with the DMA driver for streaming data to/from the PL.

IV. EVALUATION

We present experiments to quantify our runtime's performance against to the Xilinx FPGA Manager driver. Experiments were performed on the Ultra96v1 (ZynqMP) board.

We benchmark loading 3 PR bitstreams (of sizes 5.67 MiB, 4.43 MiB and 0.86 MiB respectively) using the Xilinx FPGA manager and our runtime service. We evaluate two versions of the FPGA Manager; as provided by Xilinx (verbose output from kernel) and with the kernel logging silenced. FPGA Manager is verbose by default; outputting serial logs during PR. We intentionally modify and optimise its source code to disable logging to improve performance [11].

A single PR region with 3 different modes: a Xilinx AXIS FFT IP Core, a Verilog AXI AES Cryptographic Module, and an HLS generated AXIS FIR Module was defined. Given that PR modes for the same PRR are equal in size, we vary pblock size to simulate a range of bitstream sizes. Fig. 4 highlights our runtime's PR performance, showing a maximum of up to a **66.9 %** reduction in completion time over the FPGA Manager, approaching 50.3 % as bitstream size increases (at 5.67 MiB). This equates to an approximate maximum throughput of **398.6 MB/s** for our tool, compared to the maximum

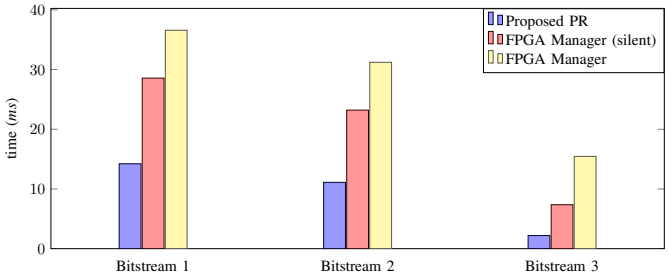

Fig. 4. PR Manager Performance.

throughput of the FPGA Manager, 190.8 MB/s, under the 5.67 MiB bitstream. Larger bitstreams offer better throughput as the fixed trigger overhead is amortised as transfer time increases. The 5.67 MiB bitstream is approximately 12.07 % of the available PL resources (logic CLBs).

These results are from the point of triggering PR as both runtimes require an initial setup time with scaling factors determined by bitstream and buffer sizes. Due to Linux's non-deterministic scheduler, we observed a maximum variation in trigger latency of 7 μs, across 20 runs. Table I shows the variables relevant to our tool's setup time; T_{config}, the time consumed writing to the CSU register, $T_{bitstream}$, and $T_{accel.}$, the cumulative time to allocate *udmabuf* buffers. We consider this relevant as buffers must be allocated initially, prior to triggering.

There are finite PL resources and it is important to minimize the resources consumed by PL infrastructure to allow for maximum user accelerator resources. In our tool, resource usage scales according to the number/type of PR interfaces in the designer's modules. The number of PR accelerator interfaces will vary the size of the AXI interconnect and the AXIS multiplexer/demultiplexer. Table II shows the resource consumption in the PR manager infrastructure, shown with a combination of AXIS and AXI Lite interfaces, where the majority of resources are consumed by the DMA controller. Given that the DMA is the primary method for PL to PS data transfer and is used by both the PR accelerators and the PR controller, we consider that less than 10 % total usage of the PL is acceptable.

V. Conclusion and Further Work

We have presented a collection of abstractions for Zynq and Zynq UltraScale+ devices, offering non-blocking PR from within the Linux userspace, bitstream build and runtime abstraction, as well as improved PR performance. These abstractions allow for extensible hardware generation, building from defined user configurations and modes and directly exporting to Linux. Our runtime service demonstrates an up to 66.9% reduction in PR load time over the Xilinx FPGA Manager driver. Additionally our abstractions prioritizes PR performance while providing simple flow control to PL accelerators.

We intend to assemble these tools into an open source end-to-end PR build and runtime framework for Zynq/MP families in future work.

Acknowledgement

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

References

[1] S. Bhandari, S. Subbaraman, S. Pujari, F. Cancare, F. Bruschi, M. D. Santambrogio, and P. R. Grassi, "High speed dynamic partial reconfiguration for real time multimedia signal processing," in *Euromicro Conf. on Digital System Design*, 2012, pp. 319–326.
[2] J. Huang, M. Parris, J. Lee, and R. F. Demara, "Scalable FPGA-based architecture for DCT computation using dynamic partial reconfiguration," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, 2009.
[3] S. Shreejith, K. Vipin, S. A. Fahmy, and M. Lukasiewycz, "An approach for redundancy in FlexRay networks using FPGA partial reconfiguration," in *Design, Automation & Test in Europe Conf. (DATE)*, 2013, pp. 721–724.
[4] S. U. Bhandari, S. Subbaraman, S. Pujari, and R. Mahajan, "Real time video processing on FPGA using on the fly partial reconfiguration," in *IEEE Int. Conf. on Signal Processing Systems*, 2009, pp. 244–247.
[5] B. Krill, A. Ahmad, A. Amira, and H. Rabah, "An efficient FPGA-based dynamic partial reconfiguration design flow and environment for image and signal processing IP cores," *Signal Processing: Image Communication*, vol. 25, no. 5, pp. 377–387, 2010.
[6] B. Osterloh, H. Michalik, S. A. Habinc, and B. Fiethe, "Dynamic partial reconfiguration in space applications," in *NASA/ESA Conf. on Adaptive Hardware and Systems*, 2009, pp. 336–343.
[7] K. Vipin and S. A. Fahmy, "A high speed open source controller for FPGA partial reconfiguration." in *Int. Conf. on Field Programmable Technology*, 2012, pp. 61–66.
[8] M. Hübner, D. Göhringer, J. Noguera, and J. Becker, "Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs," in *IEEE Int. Sym. on Parallel and Distributed Processing Workshops and PhD Forum*, 2010, pp. 1–8.
[9] S. Liu, R. N. Pittman, and A. Forin, "Minimizing partial reconfiguration overhead with fully streaming DMA engines and intelligent ICAP controller," in *MSR-TR-2009-150*. Microsoft, 2010, p. 292.
[10] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, 2014.
[11] "Solution ZynqMP PL programming." [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841847/Solution+ZynqMP+PL Programming
[12] *UG1137: Zynq UltraScale+ MPSoC Software Developer Guide*, Xilinx Inc., Dec. 2019, v11.0.
[13] *UG909: Dynamic Function eXchange*, Xilinx Inc., Jan. 2020, v2019.2.
[14] L. Wirbel, "Xilinx SDAccel: a unified development environment for tomorrow's data center," *The Linley Group Inc*, 2014.
[15] S. Yousuf and A. Gordon-Ross, "DAPR: Design automation for partially reconfigurable FPGAs," in *ERSA*, 2010, pp. 97–103.
[16] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An open-source partial-reconfiguration toolkit for Xilinx FPGAs," in *IEEE Int. Sym. on Parallel and Distributed Processing Workshops and PhD Forum*, 2011, pp. 228–235.
[17] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A partial reconfiguration framework," in *IEEE Int. Sym. on Field-Programmable Custom Computing Machines*, 2012, pp. 37–44.
[18] K. Vipin and S. A. Fahmy, "Automated partial reconfiguration design for adaptive systems with CoPR for Zynq," in *IEEE Int. Sym. on Field-Programmable Custom Computing Machines*, 2014, pp. 202–205.
[19] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *IEEE Int. Sym. on Field-Programmable Custom Computing Machines*, 2014, pp. 109–116.
[20] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "FOS: A modular FPGA operating system for dynamic workloads," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 4, Sep. 2020.
[21] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2017, pp. 894–897.
[22] E. Lubbers and M. Platzner, "ReconOS: An RTOS supporting hard- and software threads," in *Int. Conf. on Field Programmable Logic and Applications*, 2007, pp. 441–446.
[23] T. de Albuquerque Reis and A. A. Fröhlich, "Operating system support for difference-based partial hardware reconfiguration," in *IEEE/IFIP Int. Sym. on Rapid System Prototyping*, 2009, pp. 75–80.
[24] *UG570: UltraScale Architecture Config.*, Xilinx Inc., Sep. 2019, v1.11.
[25] K. Ichiro, "udmabuf driver," https://github.com/ikwzm/udmabuf, 2015.
[26] B. Perez and J. Choi, "Xilinx AXI DMA Linux driver," https://github.com/bperez77/xilinx_axidma, 2015.