Dynamic and Partial Reconfiguration of FPGAs

15

Suhaib A. Fahmy and Krishnan B. Iyer

Contents

Abstract

The reconfigurability of FPGAs is a unique capability that can be exploited beyond just repurposing or modifying hardware designs. Static reconfiguration, where a single monolithic hardware design is replaced by another, allows for in-field upgradability and enhancements, de-risks hardware deployment, and enables their function as off-the-shelf programmable devices. However, this

S. A. Fahmy (✉)
King Abdullah University of Science and Technology (KAUST), Department of Computer, Electrical and Mathematical Sciences and Engineering, Thuwal, Saudi Arabia
e-mail: suhaib.fahmy@kaust.edu.sa

K. B. Iyer
Computer Science, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
e-mail: krishnan.iyer@kaust.edu.sa

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_51

configurability, through modification of configuration memory contents, also opens the door to dynamic reconfiguration, where hardware designs are changed at runtime to serve different purposes. More advanced still is the ability to modify only portions of the hardware architecture, while the rest remains functional. By closing the loop, wherein the static part of the hardware is responsible for controlling the reconfiguration of the dynamic part, self-reconfiguring systems are possible. This chapter explores the dynamic and partial reconfiguration capabilities of FPGAs from the perspectives of the architecture, the programming model, and the applications that can leverage these unique capabilities.

Keywords

Reconfigurable computing · Field programmable gate arrays · Partial reconfiguration · Self-adaptive hardware

Introduction

The flexibility of field-programmable gate arrays (FPGAs) is afforded by the various programmable elements discussed in Boutros and Betz (2023). The configuration memory is tightly coupled with these elements, with individual bits of the configuration memory controlling features of the various elements to implement supported functions. These can be the contents of the look-up tables (LUTs) to implement logic functions, whether logic element flip-flops are enabled or not and how they are connected, whether the LUT implements a fractured function, which input and output ports of a switch box to connect, single-ended vs. differential I/O configurations, and the operating modes of hardened macros like DSP Blocks and Block Memories. Figures 1 and 2 show a representation of how configuration affects logic elements and routing.

The contents of the whole configuration memory represent the hardware context of the FPGA, that is, the way that every programmable element is configured to implement the required circuit. This binary data is referred to as the *bitstream*. Loading a bitstream into the FPGA sets up the FPGA as determined by the way the contents configure each of the millions of programmable elements. It is worth noting that it might be possible to load a bitstream into the FPGA that configures a circuit that does not function correctly or even worse causes physical damage to the FPGA, hence the need for approaches to ensure only valid bitstreams are loaded, as will be discussed later.

The most widely used FPGAs today have a volatile SRAM-based configuration memory, thereby enabling reconfiguration: the rewriting of the configuration bitstream. While early devices required this to be done while in a reset state—a *static* reconfiguration—many devices today allow *dynamic* reconfiguration, that is, the rewriting of the bitstream while the device is active, effectively a hardware context switch. A further enhancement that significantly increases the utility of this feature is *partial* reconfiguration, wherein only portions of the configuration memory are replaced, thereby modifying the context of only the corresponding areas of the

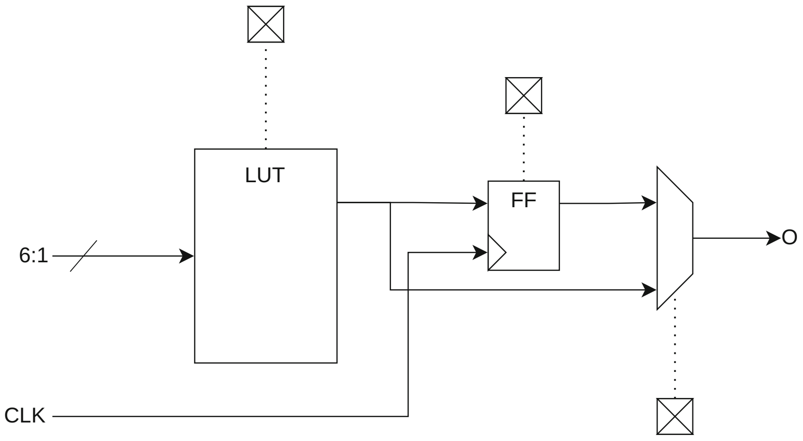
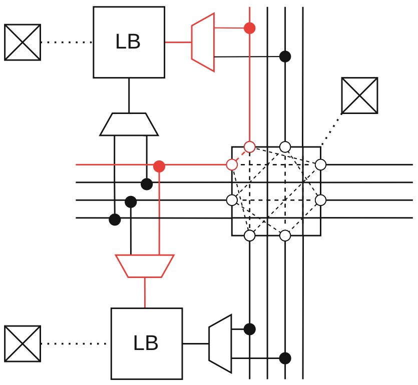


Fig. 1 Configuration of a simplified logic element showing 64-bit LUT6 contents determining logic function, flip-flop as latch with initial state, and multiplexer select pins set by the values in the configuration memory

Fig. 2 Configuration of a route between two logic blocks through configuring the connection boxes and the switch box to connect them



device. This opens the door to hardware designs where parts can be modified at runtime based on functional needs. A *static region* contains the parts of the hardware design that remain active throughout operation, while one or more *partial regions* contain functionality which can be swapped out during operation.

This capability has found use in a variety of applications, such as cognitive radio systems that modify their hardware baseband processing depending on operating requirements, computer vision systems that adapt the type of image processing depending on the properties of the scene, virtualization of FPGA interfaces for generalized accelerator deployments, and resilient designs that are robust to radiation effects in space.

This chapter discusses the mechanisms that enable dynamic and partial reconfiguration and how to design such systems and gives examples of applications that exploit these capabilities. Future challenges in this research area are also identified. Various existing surveys delve into these aspects in more detail and the authors are referred to these (Compton and Hauck (2002), Koch (2012), Vipin and Fahmy (2018), and Vaishnav et al. (2018)).

FPGA Configuration

The configuration memory is what enables the flexibility of FPGAs to be exploited. Different memory technologies are used to implement FPGA configuration memories (Kuon et al. 2008); among them Flash, Antifuse, and SRAM are prevalent. Antifuse memory is one-time writable and hence FPGAs that use it do not provide features like in-system programming and reconfiguration. SRAM memory is typically latch-based, which provides stability against minor noise or glitches, and offers write stability due to the feedback in the cells. However, due to volatility, SRAM-based FPGAs must be configured each time they are powered on. Flash-based configuration memory is non-volatile, thereby offering instant power-up, but suffers from limited write cycles, and is therefore not ideally suited for reconfigurable applications. Hence, SRAM-based FPGAs typically offer the reconfiguration capability that is discussed in this chapter. However, due to their volatility, bitstreams must be stored in non-volatile memory, e.g., Flash, to be loaded onto the FPGA after any power reset.

One complication with storing bitstreams in external non-volatile memory is the potential for a bitstream to be read out by a malicious attacker who can then use it on their own (identical) device to implement the same circuit. It is worth clarifying that while this is problematic in the general sense, reverse engineering the bitstream to obtain the original hardware description language source code for design remains as complex as reverse engineering a software binary to obtain its source code. Rather the bitstream can be used to extract a complex low-level netlist of device primitives that requires significant effort to analyze (Benz et al. 2012).

However, considering these designs represent the value offered by FPGA system designers, there are various mechanisms by which bitstreams are secured. Both Intel/Altera and AMD/Xilinx support on-chip decryption of encrypted bitstreams (Xilinx 2024; Karen Horovitz 2024; Intel 2021), with a dedicated memory for storing the encryption key. At present, they both use AES encryption, while AMD/Xilinx uses RSA and Intel/Altera uses ECDSA for authentication. This allows bitstreams to be locked to writing a specific device. To make encryption effective, vendors disable readback of the keys from memory. Some low-end FPGAs do not include these features, instead relying on obfuscation and the proprietary nature of the bitstream format. Various academic projects have tried to address this challenge through custom encryption and authentication schemes (Vliegen et al. 2013; Duncan et al. 2019). A thorough review of the security implications of FPGA reconfigurability is presented in Proulx et al. (2023).

Bitstreams can also be compressed to allow faster reconfiguration. Custom schemes would still require decompression before being passed to the configuration interface, so they only offer the benefit of reduced bitstream size in memory.

AMD/Xilinx supports a proprietary compression technique enabled by setting a special flag during bitstream generation and this scheme is natively supported in the hardware. This offers high bitstream compression when there is lower device utilization (Xilinx 2023). A custom scheme that exploits similarity in bitstream data based on a custom relocation scheme is presented in Beckhoff et al. (2014).

The bitstream is not itself a one-to-one mapping to the contents of the configuration memory but rather a microcode for the FPGA configuration controller to encode and direct data to specific locations in the configuration memory. Being proprietary, it requires significant effort to fully decode this relationship, though some understanding of the frame structure and bitstream format can be gained from the AMD/Xilinx official configuration guides (Xilinx 2023). Various academic efforts have successfully reverse engineered these bitstream formats for some devices to enable open-source tools to target them (Soni et al. 2013).

FPGAs can be configured by writing bitstreams over a variety of interfaces. The most basic approach is over the JTAG debugging interface. This is slow as it is not optimized for throughput and bitstreams are sent as individual memory writes. External configuration interfaces are faster but must be managed by a distinct processor. FPGA SoCs have internal configuration interfaces that the processors can manage to configure the FPGA portions on the same die. Interface selection determines where the bitstreams can be stored and how long the reconfiguration process takes. This is discussed in more detail in Section "Managing Partial Reconfiguration".

Generally, reconfiguring an FPGA required a reset to be applied first, to protect against stray values which might cause unexpected behavior after configuration. More recent FPGAs allow dynamic reconfiguration, allowing a new bitstream to be written while the device is active. This enables a new form of reconfigurability: partial reconfiguration (PR).

PR allows reconfiguration of a specific region of the FPGA while the rest remains operational. To exploit PR, a design must be partitioned into a static region (SR) and one or more partially reconfigurable regions (PRRs). The static region maintains its hardware context throughout the operational lifetime of the device (until power off for devices with volatile configuration memory). It houses any fixed functionality required throughout operation, such as external interface logic, a soft processor that manages the system, memory connections, and other peripherals. Partially reconfigurable regions have multiple contexts that can be swapped at runtime. These must be arranged on the device according the various constraints to be valid. To swap contexts, a partial bitstream must be loaded to reconfigure the relevant region.

PR presents some design challenges discussed in Section "Designing Partially Reconfigurable Systems" such as the coarse granularity of PR, relocation of bitstreams, the consistency model for reconfiguration, decoupling of signals during reconfiguration, high-level design support, and simulation support. The consideration impacts build and runtime development flow significantly. For example, simulation support is lacking in vendor tools, and hence, potential pitfalls caused by dynamic reconfiguration can be missed.

Designing Partially Reconfigurable Systems

One of the obstacles to wider adoption of PR has been design complexity. Most FPGA designers are content with writing RTL code and letting FPGA tools manage the complexity of mapping to the target architecture. Indeed, in such cases, the designer need not know much about the types of resources on the FPGA and their arrangement in silicon. Optimization for high frequency tends to require both some understanding of the underlying hardware primitives and, potentially, some spatial *floorplanning*, wherein the location of hardware primitives is fixed. However, this remains a specialist skill. Designing a PR system on FPGA requires awareness of this physical arrangement and consideration of its features at a finer level than many designers consider. FPGA tool vendors have been improving their tools recently, but this additional complexity remains a barrier to entry. The PR development flows from both AMD/Xilinx and Intel/Altera follow a similar structure.

For consistency, a common terminology is defined first. PR development requires the area of the FPGA to be divided into a *static region* (SR) and one or more *partially reconfigurable regions* (PRRs). The static region hosts the static module, which is initialized once during the configuration lifetime (until power-off). The static module can include logic to manage partial reconfiguration, such as communication with the ICAP or decoupling signal circuitry.

The partially reconfiguration regions (PRRs) can be reconfigured multiple times during system lifetime. Reconfigurable modules (RMs) are the designs configured into the PRRs. A *configuration* is a valid combination of the SR and a set of RMs allocated to the PRRs in the design. A *full* bitstream includes the static module and an RM instance for each PRR. A *partial* bitstream only contains an instance of an RM for a particular PRR. Figure 3 shows these regions and the associated bitstreams. In generating partial bitstreams for RMs, these are validated against the static module to ensure correctness and compatibility. Subsequently, all bitstreams, including static and partial, must be generated prior to configuration. Since both static and partial bitstreams must be generated through the full hardware implementation flow, which is time-consuming, their functionality, arrangement, and design must be determined up front.

Signals that interface the SR and PRRs must be locked in place (which is now automated by the tools), so that the multiple RMs all route their matching interfaces to these signals. RMs in the same PRR can have different interface signals, but the SR must implement all the required signals to interface with them.

The PR design tools start by synthesizing all RMs separately. The SR is also synthesized with the RMs replaced by black boxes, i.e., empty modules. Only during the implementation stage are the synthesized netlists of the RMs populated into the black boxes.

PRRs must be drawn manually in the implementation stage, ensuring they align with clock region boundaries. It's preferred that PRRs are rectangularly shaped in order to have uniformity in resources; any other shape is likely to cause routing congestion (Xilinx 2023).

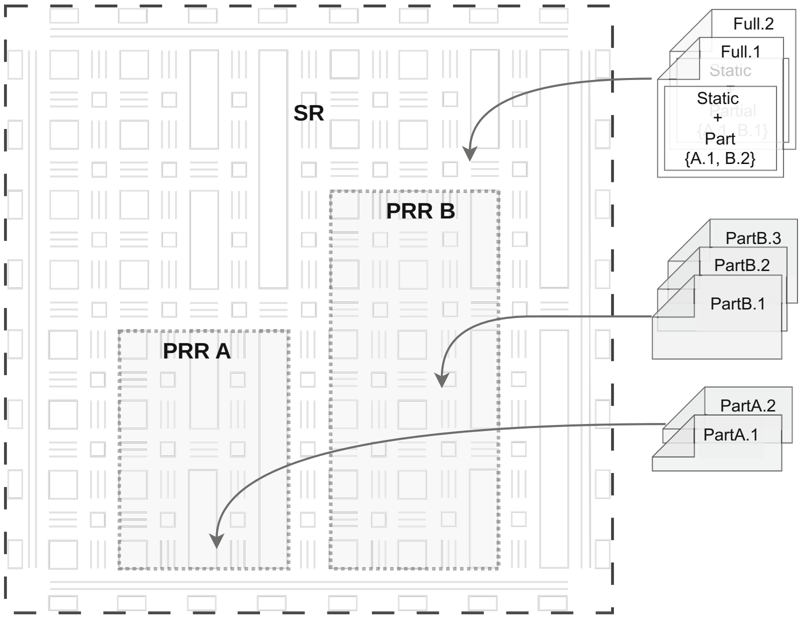


Fig. 3 A partially reconfigurable design comparing a static region (SR) and two partially reconfigurable regions (PRRs), A and B. Full bitstreams can be used to configure the whole FPGA, while partial bitstreams allow individual PRRs to be configured to a specific function. The granularity of regions is limited to device-specific constraints based on clock regions and resource arrangements

Additionally, not all resources are reconfigurable. LUTs, FFs, DSPs, CMACs, PCIe interfaces, clocks, and clocks modifying logic such as BUFG, PLL, etc., are reconfigurable. Similarly, I/O and I/O-related components such as drive strength, driver output impedance, SERDES, etc., are also reconfigurable. However, components such as BSCAN, JTAG, and ICAP are not reconfigurable.

One RM for each PRR is incorporated and a complete netlist (comprising the SR and an RM for each PRR) is generated. At this point, the static design is locked, meaning that its physical use of resources is fixed. Subsequent RMs are swapped into each PRR, and the same process repeated to determine their own implementation.

Netlists are generated for all configurations defined by the designer, resulting in a full bitstream for every configuration and partial bitstreams as needed for the PRRs to implement all configurations. Figure 4 shows an overview of the stages of designing a PR system. There are a number of academic tools that seek to further enhance the PR build flow.

Determining how many PRRs to use and which RMs to allocate to each PRR is non-trivial. Fewer PRRs containing multiple larger RMs mean that each time any part of an RM needs to be modified, the whole PRR must be reconfigured,

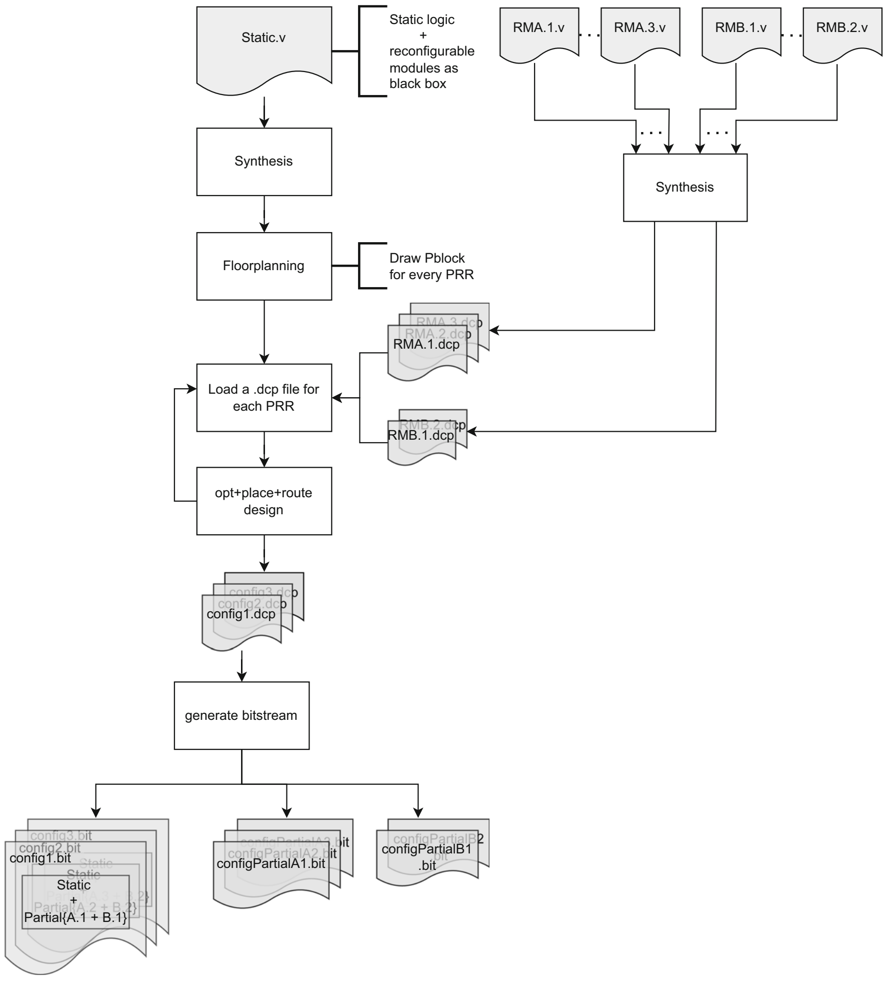


Fig. 4 The partial reconfiguration design flow. Reconfigurable modules are separately synthesized. Static logic is first passed to the design flow with reconfigurable modules as a black box. A floorplan is created for the PRRs, each of which is populated by the synthesized modules. Placement and routing are carried out on a valid configuration with PRRs populated with the corresponding RMs. This process is repeated for all valid configurations. Finally, bitstreams are generated for valid configurations

potentially resulting in some non-functional time and also slower reconfiguration (since reconfiguration time is proportional to bitstream/PRR size). Using more PRRs with smaller RMs can improve reconfiguration time, but can result in more overhead for creating suitably shaped and sized PRRs for diverse RMs. The work in Montone et al. (2010) and Vipin and Fahmy (2013) attempts to automate this process by considering these aspects.

Floorplanning PRRs is also non-trivial as the columnar arrangement of resources in the FPGAs, the need to respect clock domain boundaries, and the need for some overhead to ease placement and routing all complicate the required arrangements. Various approaches to automated floorplanning for PR systems have been proposed. These include mixed integer linear programming formulations (Rabozzi et al. 2016), kernel tessellation (Vipin and Fahmy 2012), and evolving tool placements (Beckhoff et al. 2013).

GoAhead (Beckhoff et al. 2012) automates aspects of bitstream generation including allowing relocation of RMs between PRRs, though it does require user intervention for floorplanning. It automates the insertion of blocker macros to prevent wires from the SR crossing through the PRRs.

CoPR (Vipin and Fahmy 2014) takes a description of valid system configurations and automates the process of determining PRRs, allocating RMs to PRRs, and floorplanning, followed by automation of the build process through scripting of the vendor tools.

BITMAN (Pham et al. 2017) is an open-source tool for generating and manipulating bitstreams. These manipulations can include relocation of bitstreams to different PRRs and modifying individual primitives in the FPGA.

Recently, both AMD/Xilinx and Intel/Altera have added support for nested PRRs. In earlier tools, the designer would need to determine the arrangement of PRRs in advance of design, and these could then not be changed. In the newer flows, PRRs can themselves contain PRRs, allowing the arrangement of lower-level PRRs (*child* partition) to be modified at runtime through reconfiguration of the higher-level PRRs (*parent* partition). For a child partition, the rest of the parent partition is treated like a static partition, even if the parent partition is a PRR. A child RM can be swapped out without affecting the rest of the parent partition. Figure 5 shows a representation of this concept.

The tool flow for hierarchical PR requires another level of floorplanning, netlist, and bitstream generation, including verification of all the possible configurations. RM implementations can be run in parallel once the corresponding static design and floorplan are in place.

AMD/Xilinx also introduced the abstract shell concept, allowing new RMs to be compiled for a PRR without having the full SR available. The abstract shell encapsulates all the required information about the surroundings of the PRR to enable the implementation tools to implement an RM into the PRR. This is shown in Fig. 6.

Despite these improvements, the tool flows still entail some complexities. AMD/Xilinx tools can route wires for the SR through PRRs. As a result, any change in the SR requires all RMs to be re-implemented. Additionally, these reserved wires in the PRRs can make some RMs difficult to route. This is referred to as *bleeding* over the dynamic region (Xilinx 2023). This can be overcome by setting the `CONTAIN_ROUTING` constraint to force routing to stay within the bounding box of the SR. It is also possible to avoid this by placing blockers around the PRR, which prevent wires from entering it (Beckhoff et al. 2012). However, this can cause timing-related issues as wire routes might be longer.

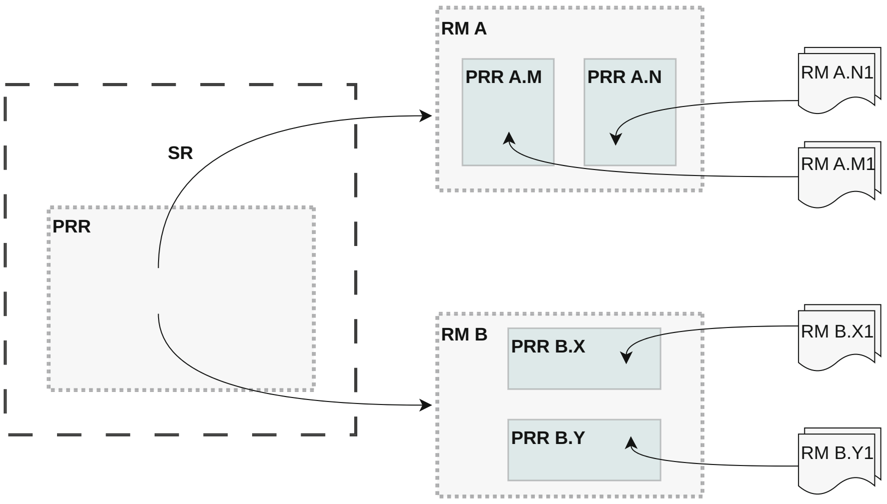


Fig. 5 The nested PR flow with two different nested PRR allocations allowing the location and size of these PRRs to be modified at runtime, thereby enhancing flexibility

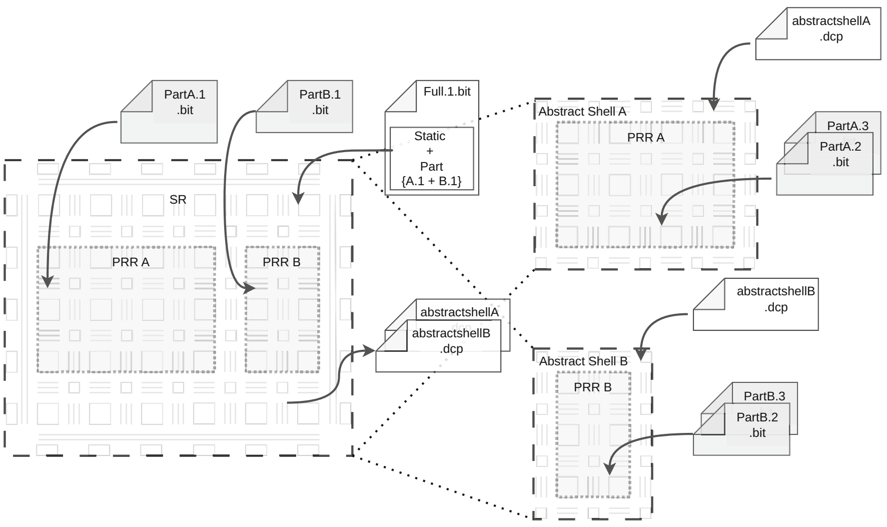


Fig. 6 The abstract shell flow enhances the traditional PR design flow by generating abstract shell checkpoints for each PRR that enable partial bitstreams to be generated without needing the full bitstream

Another key consideration is decoupling signals from/to PRRs and resetting to a known state after reconfiguration. Decoupling helps SR and PRRs to isolate stray signals that might emerge during reconfiguration. This is more complicated for hierarchical PR.

There are a few more considerations specifically targeted at hierarchical PR. Child RMs should only be reconfigured with the corresponding parent RM as there's no verification support built into a configuration controller; the configuration initiator's responsibility is to verify that a child RM corresponds to the current instance of parent RM prior to reconfiguration. Currently, PR and hierarchical PR design tools lack support when it comes to encryption and authentication of bitstreams. Authentication is only supported for full bitstreams, i.e., loading the static region and not for partial bitstreams. Partial bitstreams can be encrypted, but must be with the same key as the static bitstream.

Managing Partial Reconfiguration

Once a design has been compiled and all the full and partial bitstreams generated, the system can use these to switch between hardware configurations. A full bitstream is first loaded, instantiating the SR and potentially an RM in each PRR. (It is possible to create a configuration with an empty RM in each PRR for the initial state.) Partial bitstreams are similar in format to full bitstreams, except they contain only the configuration data associated with their PRRs and are hence smaller. To switch configurations, a partial bitstream must be loaded into the configuration memory. This can be done over any of the suitable configuration interfaces (Table 1). External JTAG is the slowest, averaging 10s of Mbps, while SelectMap (for AMD/Xilinx) and Active Serial (for Intel/Altera) allow for 100s of Mbps depending on the bus width. These interfaces must be controlled by external hardware, such as a distinct processor or externally connected computer through a dedicated programming cable.

For PCIe-hosted FPGAs, AMD/Xilinx provides the MCAP interface, while Intel/Altera provides CvP. These allow a host machine to transfer bitstreams over the PCIe interface. For MPSoC designs, the embedded processor can manage configuration of the programmable FPGA fabric over a controller in the SoC portion of the chip; in the case of AMD/Xilinx, this is called the PCAP.

Table 1 Configuration interfaces and their range of supported bandwidths and resulting approximate configuration times for averagely sized bitstreams. Times marked * are negatively impacted by the overhead of individual word writes. Numbers assume interfaces operate in insecure mode, meaning data is transferred at max bandwidth. Some interfaces support various clock rates

Type	AMD/Xilinx	Intel/Altera	Bandwidth (Gbps)	Reconfig. time (ms)
External	JTAG	JTAG	0.030–0.066	≈10000*
	SelectMap	Active Serial (AS)/FPP	0.66–3.2	≈100*
Internal	ICAP	PR Controller	2–6.4	≈10
PCIe	MCAP	CvP	6.4	≈100*
Processor subsystem	PCAP	FPGA Manager	3.2–6.4	≈30*

AMD/Xilinx offers a primitive called the internal configuration access port (ICAP) which can be instantiated in the SR of a design and which allows bitstream data to be written to the configuration memory. In this way, the device can reconfigure itself. However, the designer must build the required circuitry to move bitstream data, potentially from memory, to this interface.

Configuration controllers expect one data word at a time. Hence, there is a requirement to manage the movement of data into these interfaces, either from host or external processor software, or through a design within the SR. Since the configuration memory is written to through a single interface, multiple PRRs can only be configured sequentially.

Reconfiguration time is a key metric for PR-based designs. Many systems must be able to change modes within a constrained time. Reconfiguration time is determined by the size of the bitstream being configured and the effective throughput of the programming interface. The first depends on the size of the PRR being configured. The second depends on the throughput of the interface and the efficiency of data transfer over the interface (due to the potential control overhead of word-level transfers). ICAP has the highest maximum data throughput of any reconfiguration interface and hence can offer the fastest reconfiguration. However, to achieve this, it is necessary to be able to stream a bitstream to the ICAP at its capable data rate, which requires the design of a custom configuration controller in the SR. It is usually impractical to store even partial bitstreams in on-chip memory due to their size and the limited capacity available. Instead, external DRAM is the most common place to store bitstreams for fast transfer via DMA.

One complication to consider is that partial bitstreams are associated with a specific PRR. Hence, even if the same RM is placed in two different PRRs, that will require two different partial bitstreams to be stored in memory. With multiple RMs mixed into multiple PRRs, the number of different partial bitstreams, and hence the storage capacity required, can increase significantly. There have been efforts to allow bitstreams to be *relocated*. This requires the use of blockers to prevent SR signals to be routed through PRRs or allocation of dedicated SR routing resources through PRRs (Beckhoff et al. 2012). Furthermore, relocation requires low-level knowledge of the FPGA architecture to determine spatial correspondence of different compatible PRRs and modifying the addresses of written configuration words to allow a partial bitstream created for one PRR to be written to another. However, this is unsupported in vendor flows and cannot be applied when bitstream encryption is enabled.

There has been ample work building hardware controllers to manage reconfiguration over the ICAP in AMD/Xilinx devices with the aim of maximizing reconfiguration throughput. The first demonstration of DMA into the ICAP was in Liu et al. (2009), but from on-chip memories, so with very limited storage capacity for bitstreams. This was extended to DMA for off-chip DRAM in Vipin and Fahmy (2012) and then over the PCIe host interface in Vipin and Fahmy (2014a).

Software-managed reconfiguration is the preferred method for FPGA SoCs through the provided PCAP interface on AMD/Xilinx devices for which an API is provided which can be integrated into user bare metal software running on the processors. However, these deal with passing raw bitstream data to the configuration controller. Reconfiguration using these APIs is a blocking operation, requiring custom designs to overcome this, as was done in Vipin and Fahmy (2014b) where custom interrupts are exploited to allow bitstream DMAs to complete while the processor is busy with other tasks.

Managing PR within an OS running on the host processor requires additional integration. The generally supported approach is a fixed shell into which RMs can be compiled to generate partial bitstreams, which can then be loaded by making calls to an API that manages the transfer of bitstream data. ARTICO3 (Rodríguez et al. 2018) is a framework for the AMD/Xilinx Zynq and ZynqMP MPSoCs that allows custom kernels to be integrated into a software managed platform with automation of access to the memory hierarchy and abstracted PR. Heterogeneous tasks can be exploited to enhance runtime and energy efficiency.

ReConOS (Agne et al. 2014) is a more flexible fully developed OS abstraction for managing PR systems. It tries to unify the management of software and hardware tasks through a unified thread abstraction. Hardware threads are managed through PR and different threads can communicate with each other through various mechanisms. However, the OS kernel must be recompiled to support new hardware threads.

FPGA OS (Vaishnav et al. 2020) decomposes the development of hardware and software for heterogeneous embedded systems while supporting multi-tenancy and abstracted loading of partial bitstreams. However, similar to the above frameworks, this requires all RMs to abide by the uniform shell interface specification. This abstraction works well for pure compute acceleration, but not for some embedded systems where the required interfacing of modules can vary.

ZyPR (Bucknall and Fahmy 2023) extends abstractions further by allowing Linux Device Tree Overlays (DTOs) to be updated dynamically based on the loaded RMs with these being enumerated during the build process. This allows for RMs that access different external I/O to be supported. Additionally, ZyPR presents an abstracted configuration view to the user through its API, with its middleware translating these configuration changes into the required bitstream loading.

Coyote (Korolija et al. 2020) applies various OS abstractions to management of FPGAs, including a process model, scheduling, virtual memory, and I/O virtualization. It also supports partial reconfiguration of FPGA tasks using similar approaches to those described in Section "FPGA Configuration".

Applications of Dynamic Partial Reconfiguration

Having discussed the mechanisms and design approach for dynamic partially reconfigurable systems, how this capability can be exploited in a variety of applications is now discussed in some detail.

Computing Infrastructure and Virtualization

One of the key benefits of FPGAs as a computation platform is that their highly flexible I/O FPGAs can implement a wide range of interfaces that allow them to be integrated into a variety of deployment scenarios. For generalized compute acceleration, this has often been as PCIe accelerators much like GPUs. In such a deployment, a fixed set of FPGA pins is dedicated to implementing the PCIe interface, along with the use of soft or hard PCI interfacing logic in the FPGA. The required accelerator is integrated with this interface logic, and the FPGA can be addressed much like a GPU, as an accelerator for offloading from a host processor. Frameworks like RIFFA (Jacobsen et al. 2015) enabled accelerator designers to automate the process of building the PCIe interface logic and integrating it with their accelerator design, as well as offering software drivers to manage offload. However, each different accelerator would require a full FPGA design build with the integrated PCIe interface logic, and a static reconfiguration of the FPGA, and often, a system reboot, when changing functions.

The DyRACT framework (Vipin and Fahmy 2014a) was the first to use partial reconfiguration to keep the PCIe interface logic active in between reconfigurations of the accelerator logic and to load these bitstreams over PCIe. This concept of an interface shell, containing the required external interfaces to the FPGA and the hardware required to manage reconfiguration in static logic, and the accelerator function in dynamic logic, applied using partial reconfiguration, is now widespread. Microsoft's Catapult project (Caulfield et al. 2016) refers to the static portion as the Blue Bitstream and the user's function as the Green Bitstream. Amazon's AWS F1 (Inc. 2024a) refers to the Shell and Custom Logic. This approach is essential to productive use of FPGAs as accelerators as it means the FPGA's state does not adversely affect the host machine, and that the FPGA can be put to use for diverse application needs as those needs change over time. AMD/Xilinx's Xilinx Runtime Library (XRT) (Inc. 2024b) integrates a host-based runtime with a hardware *platform* composed of the static *Shell* and *User* portions as above. In some of these frameworks, multiple concurrent accelerators are supported, thereby enabling multiple applications to share the FPGA resources and interface access, often referred to as multi-tenancy (Nguyen and Kumar 2020).

In recent years, data center use of FPGAs has increased, and the ability of FPGAs to directly process network packets has become important. FPGA Smart NICs have thus become more commonplace. These are FPGA accelerator cards with PCIe interfaces as above, but with high-speed network interfaces now also serving to interface with the datacenter network. Frameworks like Corundum (Forencich et al. 2020) and AMD OpenNIC (Inc. 2024) offer analogous solutions to RIFFA that extend to the network interface. As of now, these functions are usually integrated in static bitstreams, but the role of reconfiguration to enhance flexibility is expected to see these frameworks extended to support partial reconfiguration soon. Serving as a "bump in the wire" NIC, i.e., passing all packets to a host, while potentially processing a subset of them, requires that any reconfiguration of function does not disrupt network to host connectivity. Partial reconfiguration is a way to achieve this (Anup Agarwal and Seshan 2023). In Bucknall et al. (2019), the authors demonstrate reconfiguration over a network interface on the AMD/Xilinx Zynq by routing configuration packets directly to the ICAP for high-speed reconfiguration.

Within embedded systems, computing resources are constrained and hardware acceleration becomes an important factor to enable computationally complex applications. However, hardware resources such as the programmable logic(PL) in an FPGA SoC cannot be committed to just individual functions. Instead, in many cases, hardware accelerators should be loaded as needed based on application requirements. Since FPGA SoCs include a software-programmable processor that can even run an operating system, integrating PR becomes important. There are a variety of frameworks that address swapping accelerators, considering the properties of PR and their impact on execution time for complex applications (Steiger et al. 2004). Approaches to support real-time schedules have also been explored (Rossi et al. 2018).

PYNQ (Inc. 2024) from AMD is a framework that provides Python APIs to manage and interact with hardware in the programmable logic (PL) of an FPGA SoC. They refer to the PL configuration as an overlay, and it is loaded via a Python function which loads the corresponding full bitstream into the PL, to instantiate required interfaces and accelerators. Custom drivers are required to be able to access accelerators from Python. PYNQ had some initial experimental approaches for integrating partial reconfiguration (Janßen et al. 2017), which has now been added to the release, necessitating a hierarchical block design to partially reconfigure any region on the FPGA, which are referred to as PYNQ Composable Overlays.

It should be clarified that the use of the term *overlays* in this context simply refers to the different hardware configurations used in the PL of an FPGA SoC. More general compute overlays are usually coarse-grained architectures implemented atop the FPGA (Capalija and Abdelrahman 2013; Jain et al. 2021). These are configured using full reconfiguration then the overlays themselves are configured using a custom interface provided by the overlay. Since the configuration space is so much more constrained, these configuration "bitstreams" are orders of magnitude smaller, do not use the FPGA configuration infrastructure, and hence can be configured orders of magnitude faster than traditional partial reconfiguration (Stitt and Coole 2011).

Design Compilation

A major challenge with FPGA design is the compilation time from synthesis to bitstream generation. This can be hours, or even days, long for complex designs. This is problematic in the development cycle, where even a small change can add many hours to compilation time.

SPADES (Nguyen et al. 2023) demonstrates the effective use of the hard Network on Chip (NoC) in AMD Versal devices, leading to significant gains in compilation time by separating different parts of the compilation to target independent regions of the FPGA.

HiPR (Xiao et al. 2019) extends this idea using partial reconfiguration. A large design is divided into multiple blocks, where each block is contained in a PRR with minimal overhead. This enables incremental compilation, where only a single PRR needs to be recompiled when a change is made, and parallel compilation, where all modules can be separately compiled in parallel, both of which dramatically reduce design iteration time. They use a packet-switched overlay network for communication between these independent blocks. Placing this network in a PRR eliminates the complex time-consuming routing between different blocks.

In further work Xiao et al. (2022), the authors try to build PR abstractions for high-level synthesis (HLS) to automatically generate static and PR regions. They incorporate the abstract shell design flow for PR functions to be placed and routed in parallel. They show that the potential loss of design optimizations across a large compile is minimal when compared with the productivity benefits.

Adaptive Systems

FPGAs have found widespread use in embedded systems where they can often absorb all the required computing capability to implement complex systems that interact with their environment. This is even more prevalent now with the wide availability of FPGA SoCs that include processor cores on the same fabric, providing software programmability alongside tightly coupled hardware acceleration. Many of these applications in communications, automotive, industrial control, etc. require some level of adaptability to evolving conditions and PR allows these systems to modify their accelerated computing functions as needed at runtime. However, as alluded to in Section "Designing Partially Reconfigurable Systems," the design process can become an obstacle to adoption by domain experts.

Cognitive radio systems have been widely implemented on FPGAs using PR. This application requires a radio system to modify its baseband radio processing algorithms based on adapting requirements. Baseband processing chains are usually implemented in hardware to enable high throughput and low latency. Since the required hardware can be significantly different for different operating modes, multiplexing the various modes can be costly in terms of area. PR allows such radio systems to switch between different modes, e.g., sensing and multiple baseband modes, dynamically at runtime. This allows the radio systems to consume less area and power while still supporting dynamic operation. An example design in Sadek et al. (2017) shows a multi-mode 3G, LTE, and WiFi transceiver using PR to switch modes that reduces power consumption by 66% compared to a multiplexed design while only requiring less than 1.5 ms to switch modes. A similar approach combining PR with module parameterization improves reconfiguration time (Pham et al. 2017).

In vision systems, there can be various forms of data-dependent processing that require different accelerators. Rather than implement all modes and select between them at runtime, it can be more efficient to dynamically load accelerators. Since the inter-frame interval can be in the order of 10s of milliseconds, it is possible to reconfigure and complete processing within this interval. Nguyen and Hoe (2018) demonstrate this capability by reconfiguring multiple PRRs with different image processing blocks within the time take to process a single frame, achieving 60fps at 720p and 30fps at 1080p resolutions. Nava et al. (2011) demonstrate a similar approach in the vision system of a robot, where, depending on the number of colors detected, different processing modes are enabled using PR. Figure 7 shows an example setup of a partially reconfigurable video processing system.

FPGA PR has also been exploited in networking applications. A Ternary Content-Addressable Memory (TCAM) architecture using PR was presented in Reviriego et al. (2019). TCAMs are used for complex pattern matching and lookups in software-defined networking applications but must be emulated on the FPGA architecture components. The authors showed that PR could be used to reduce the area consumption of TCAM designs by over 35% while still supporting update rates of several hundred or thousand rules per second. The authors in Li et al. (2018) use PR to enable network function virtualization (NFV) using a framework called DHL that they propose. It allows virtual network functions to run in software while offloading computationally intensive portions of code to an FPGA, using PR to support multiple functions.

Adaptive systems must be agile as they are deployed in unknown environments sometimes requiring a change in processing hardware. The Observe-Decide-Act

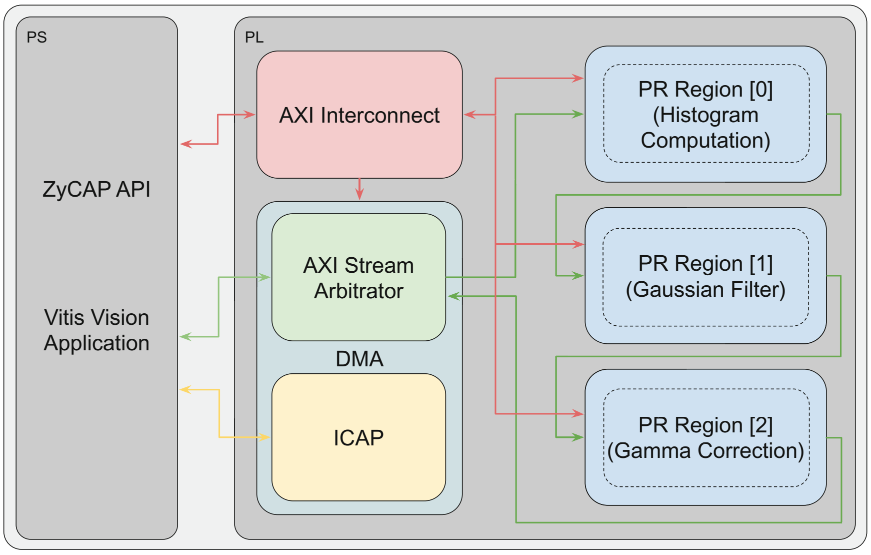

PS
ZyCAP API
Vitis Vision Application
PL
AXI Interconnect
AXI Stream Arbitrator
DMA
ICAP
PR Region [0] (Histogram Computation)
PR Region [1] (Gaussian Filter)
PR Region [2] (Gamma Correction)

Fig. 7 A vision application on an AMD/Xilinx Zynq UltraScale+ MPSoc device, showing the PS running software that adapts the types of vision filters executed in the PL using PR (Bucknall 2022)

loop typical in adaptive systems can be mapped well to FPGA SoCs where context switches can be applied through PR (Fahmy 2018). PR can also support various robustness and failure recovery modes in such systems (Paulsson et al. 2006).

Machine Learning

Significant work has been conducted on optimizing deep neural network accelerator hardware. These optimizations often require mixed or custom numerical precision support, and FPGAs are ideally suited to such hardware designs. FPGAs have been used to implement accelerators for models with precision ranging from 32-bit floating point, through various fixed point representations, down to binary neural networks, being able to trade off area against a tolerable accuracy loss.

In Hussain et al. (2015), the authors build a multi-classifier system that can switch between support vector machine (SVM) and K nearest neighbor (KNN) classifiers dynamically at runtime and demonstrate that this is up to $8\times$ faster than traditional full bitstream reconfiguration.

In Irmak et al. (2021), the authors demonstrate how PR can be used to dynamically switch convolutional neural network (CNN) architectures for multiple applications. They show that having optimized CNNs for different applications and swapping them using PR is preferable in terms of accuracy to training a hybrid CNN while still offering low latency.

In Venieris and Bouganis (2017), the authors build a CNN design framework that allows multiple blocks of a CNN to be allocated to the same hardware accelerator through weight only reconfiguration while amortizing the more costly partial reconfiguration for other blocks through using larger batch sizes to achieve low latency.

CascadeCNN (Kouris et al. 2018) switches between accelerator hardware with different numerical precision and hence accuracy. If a sample is misclassified, it is recomputed with a higher precision model which is slower; otherwise, the faster inference is accepted. Partial reconfiguration is suggested as a way to switch between these models.

Reliability and Harsh Environments

Designing electronic systems for harsh environments is challenging. Ionizing radiation affects the properties of transistors, and high-energy particles can damage the materials themselves, all resulting in failures. These long-term effects are alongside immediate *single event effects* (SEEs) which can occur due to high-energy particles colliding with a device. FPGAs suffer in the same way as other devices, but beyond effects in the implemented circuit, the configuration memory is also susceptible to corruption. This depends on how the configuration memory is implemented (Wirthlin 2015). Antifuse and Flash configuration memories are generally robust to SEEs, though Flash is more susceptible to longer-term ionizing doses. SRAM FPGAs, however, which are more prevalent, can suffer single event upsets (SEUs) in their configuration memory, a change in a bit's state. This is highly problematic since that can modify the configured datapath and even lead to an incorrectly configured FPGA.

Space is a commonly considered harsh environment. FPGAs are popular in space applications due to increasing processing requirements and the need for custom processing architectures (Osterloh et al. 2009). Radiation in space is a result of cosmic rays or high-energy particles which can strike devices. FPGAs are also widely used in high-energy physics experiments where they are used to process detector data to extract important features for further analysis. In such experiments, the radiation environment is driven by the types of experiments being performed, which typically involved strong radiation fields and high-energy particles.

There are a variety of approaches for tackling reliability in such environments, including spatial redundancy, such as triple modular redundancy (TMR). However, a solution unique to FPGAs is configuration scrubbing. This is where the configuration memory is repeatedly rewritten to combat potential SEUs. It is possible to read the contents of this memory and rewrite it periodically, or else to continuously write the known good configuration (Stoddard et al. 2016). This approach remains imperfect since it is periodic and takes time, so it is typically combined with spatial approaches (Bolchini et al. 2007; Ichinomiya et al. 2010). An automated framework for isolated partially reconfigurable modules with TMR was presented in Pham et al. (2018). In Iturbe et al. (2011), the authors present the R3TOS framework that combines a runtime scheduler with reliability heuristics to allocate hardware tasks in a way that minimizes the effects of SEUs and ionizing radiation.

FPGAs can also offer enhanced robustness to failures. In Dörr et al. (2019) the authors demonstrate the use of PR to manage a fallback processor to support fail-operation in a complex system. In Oszwald et al. (2018), they demonstrate that this capability can meet safety requirements for automotive applications. In Shreejith et al. (2013), PR is used to recover from faults while a redundant unit manages data processing. This is done at the network level to minimize time overhead.

FPGAs can also serve as a fallback mechanism for any arbitrary faults in the system, potentially saving costs associated with adding redundancy to every subunit. For instance, consider a system with multiple control units that can be spawned on-the-fly within an FPGA. Instead of implementing Triple Module Redundancy (TMR), an FPGA as a fallback option to emulate the faulty unit can be employed.

Research Directions

Recent focus on open-source tool flows for FPGAs has highlighted the obstacles presented by proprietary bitstream formats. Efforts to address this are gaining some attention. The potential to create bitstream formats that offer more flexibility such as relocatability and runtime modification can further simplify the design and management of PR systems.

While the vendor design processes have improved, mapping these to higher-level system design paradigms useful to embedded and adaptive systems designers remains a challenge. Most frameworks discussed still require the application user to know which bitstreams correspond to a particular RM and PRR. An abstracted model that addresses system designers without hardware experience would increase the likelihood that they adopt PR in their designs.

The security considerations of partial reconfiguration in the context of multi-tenancy systems are also under exploration. Considering multiple users sharing a single FPGA, it is possible for side channels to enable information leakage or other attacks which must be mitigated (Ahmed et al. 2022).

Conclusions

Partial reconfiguration of FPGAs has been researched for two decades, yet still remains a niche feature in practical use. While there remain challenges, the enhanced vendor design flows and the increasing requirement for FPGAs to address general-purpose compute acceleration in challenging scenarios mean that PR is gaining increased attention. Present development shell concepts now provided by vendors use PR in the background, demonstrating its robustness and applicability. Recent developments like the abstract shell design flow make the idea of evolving functionality of PR systems after initial deployment feasible. Hierarchical PR also reduces the previous limitations of a single static determination of PRRs on the device. Using these developments, system designers can now apply PR in more contexts. The research community is expected to exploit such developments to further simplify and streamline PR system design leading to potential wider adoption.

References

Agne A, Happe M, Keller A, Lübbers E, Plattner B, Platzner M, Plessl C (2014) Reconos: an operating system approach for reconfigurable computing. IEEE Micro 34(1):60–71

Ahmed MK, Mandebi J, Saha SK, Bobda C (2022) Multi-tenant cloud FPGA: a survey on security. arXiv preprint arXiv:2209.11158

Anup Agarwal DK, Seshan S (2023) StaRRNIC: Enabling Runtime Reconfigurable FPGA NICs. http://reports-archive.adm.cs.cmu.edu/anon/2023/CMU-CS-23-100.pdf

Beckhoff C, Koch D, Torresen J (2012) GoAhead: a partial reconfiguration framework. In: Proceedings of the IEEE international symposium on field-programmable custom computing machines (FCCM), pp 37–44

Beckhoff C, Koch D, Torreson J (2013) Automatic floorplanning and interface synthesis of island style reconfigurable systems with GoAhead. In: International conference on architecture of computing systems, pp 303–316

Beckhoff C, Koch D, Torresen J (2014) Portable module relocation and bitstream compression for Xilinx FPGAs. In: 2014 24th international conference on field programmable logic and applications (FPL), pp 1–8

Benz F, Seffrin A, Huss SA (2012) Bil: a tool-chain for bitstream reverse-engineering. In: International conference on field programmable logic and applications (FPL), pp 735–738

Bolchini C, Miele A, Santambrogio MD (2007) TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs. In: IEEE international symposium on defect and fault-tolerance in VLSI Systems (DFT), pp 87–95

Boutros A, Betz V (2023) Field-programmable gate array architecture. In: Chattopadhyay A (ed) Handbook of computer architecture. Springer, Singapore

Bucknall AR, Shreejith S, Fahmy SA (2019) Network enabled partial reconfiguration for distributed FPGA edge acceleration. In: 2019 international conference on field-programmable technology (ICFPT), pp 259–262

Bucknall AR, Fahmy SA (2023) ZyPR: end-to-end build tool and runtime manager for partial reconfiguration of FPGA SoCs at the edge. ACM Trans Reconfigurable Technol Syst (TRETS) 16(3):34–13433

Bucknall AR (2022) Build framework and runtime abstraction for partial reconfiguration on FPGA SoCs. Phd thesis, University of Warwick

Capalija D, Abdelrahman TS (2013) A high-performance overlay architecture for pipelined execution of data flow graphs. In: Proceedings of the international conference on field programmable logic and applications (FPL)

Caulfield AM, Chung ES, Putnam A, Angepat H, Fowers J, Haselman M, Heil S, Humphrey M, Kaur P, Kim J-Y et al (2016) A cloud-scale acceleration architecture. In: IEEE/ACM international symposium on microarchitecture (MICRO)

Compton K, Hauck S (2002) Reconfigurable computing: a survey of systems and software. ACM Comput Surv 34(2):171–210

Dörr T, Sandmann T, Schade F, Bapp FK, Becker J (2019) Leveraging the partial reconfiguration capability of FPGAs for processor-based fail-operational systems. In: International symposium on applied reconfigurable computing, pp 96–111

Duncan A, Rahman F, Lukefahr A, Farahmandi F, Tehranipoor M (2019) FPGA bitstream security: a day in the life. In: IEEE international test conference (ITC), pp 1–10

Fahmy SA (2018) Design abstraction for autonomous adaptive hardware systems on FPGAs. In: NASA/ESA conference on adaptive hardware and systems (AHS), pp 142–147

Forencich A, Snoeren AC, Porter G, Papen G (2020) Corundum: an open-source 100-GBPS NIC. In: IEEE international symposium on field-programmable custom computing machines (FCCM), pp 38–46

Hussain HM, Benkrid K, Seker H (2015) Dynamic partial reconfiguration implementation of the SVM/KNN multi-classifier on FPGA for bioinformatics application. In: Proceedings of the annual international conference of the IEEE engineering in medicine and biology society (EMBC), pp 7667–7670

Ichinomiya Y, Tanoue S, Amagasaki M, Iida M, Kuga M, Sueyoshi T (2010) Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration. In: IEEE international symposium on field-programmable custom computing machines (FCCM), pp 47–54

Inc. A (2024a) AWS-FPGA. https://github.com/aws/aws-fpga

Inc. X (2024b) XRT. https://github.com/Xilinx/XRT

Inc. X (2024) Open-NIC. https://github.com/Xilinx/open-nic

Inc. X (2024) PYNQ. https://github.com/Xilinx/PYNQ

Irmak H, Ziener D, Alachiotis N (2021) Increasing flexibility of FPGA-based CNN accelerators with dynamic partial reconfiguration. In: Proceedings of the international conference on field-programmable logic and applications (FPL), pp 306–311

Iturbe X, Benkrid K, Arslan T, Hong C, Erdogan AT, Martinez I (2011) Enabling FPGAs for future deep space exploration missions: improving fault-tolerance and computation density with R3TOS. In: NASA/ESA conference on adaptive hardware and systems (AHS), pp 104–112

Intel (2021) Using the Design Security Features in Intel®FPGAs. https://www.intel.com/content/www/us/en/docs/programmable/683269/current/using-the-design-security-features-in-fpgas.html

Jacobsen M, Richmond D, Hogains M, Kastner R (2015) RIFFA 2.1: a reusable integration framework for FPGA accelerators. ACM Trans Reconfigurable Technol Syst (TRETS) 8(4):1–23

Jain AK, Maskell DL, Fahmy SA (2021) Coarse grained FPGA overlay for rapid just-in-time accelerator compilation. IEEE Trans Parallel Distrib Syst 33(6):1478–1490

Janßen B, Zimprich P, Hübner M (2017) A dynamic partial reconfigurable overlay concept for PYNQ. In: Proceedings of the international conference on field programmable logic and applications (FPL)

Karen Horovitz RK (2024) Intel FPGA Secure Device Manager. https://apps.dtic.mil/sti/pdfs/AD1052301.pdf

Koch D (2012) Partial reconfiguration on FPGAs: architectures, tools and applications, vol 153. Springer, New York

Korolija D, Roscoe T, Alonso G (2020) Do OS abstractions make sense on FPGAs? In: 14th USENIX symposium on operating systems design and implementation (OSDI 20). USENIX Association, pp 991–1010

Kouris A, Venieris SI, Bouganis C-S (2018) CascadeCNN: pushing the performance limits of quantisation in convolutional neural networks. In: 2018 28th international conference on field programmable logic and applications (FPL), pp 155–1557

Kuon I, Tessier R, Rose J (2008) FPGA architecture: survey and challenges. Found Trends Electron Des Autom 2:135–253

Li X, Wang X, Liu F, Xu H (2018) DHL: enabling flexible software network functions with FPGA acceleration. In: IEEE international conference on distributed computing systems (ICDCS)

Liu M, Kuehn W, Lu Z, Jantsch A (2009) Run-time partial reconfiguration speed investigation and architectural design space exploration. In: International conference on field programmable logic and applications (FPL), pp 498–502

Montone A, Santambrogio MD, Sciuto D, Memik SO (2010) Placement and floorplanning in dynamically reconfigurable FPGAs. ACM Trans Reconfigurable Technol Syst (TRETS) 3(4):1–34

Nava F, Sciuto D, Santambrogio MD, Herbrechtsmeier S, Porrmann M, Witkowski U, Rueckert U (2011) Applying dynamic reconfiguration in the mobile robotics domain: a case study on computer vision algorithms. ACM Trans Reconfigurable Technol Syst (TRETS) 4(3), 29:1–29:22

Nguyen TD, Kumar A (2020) Maximizing the serviceability of partially reconfigurable FPGA systems in multi-tenant environment. In: Proceedings of the ACM/SIGDA international symposium on field-programmable gate arrays, pp 29–39

Nguyen T, Blair Z, Neuendorffer S, Wawrzynek J (2023) SPADES: A productive design flow for Versal programmable logic. In: 2023 33rd international conference on field-programmable logic and applications (FPL), pp 65–71

Nguyen M, Hoe JC (2018) Time-shared execution of realtime computer vision pipelines by dynamic partial reconfiguration. In: International conference on field programmable logic and applications (FPL), pp 230–2304

Osterloh B, Michalik H, Habinc SA, Fiethe B (2009) Dynamic partial reconfiguration in space applications. In: NASA/ESA conference on adaptive hardware and systems, pp 336–343

Oszwald F, Becker J, Obergfell P, Traub M (2018) Dynamic reconfiguration for real-time automotive embedded systems in fail-operational context. In: IEEE international parallel and distributed processing symposium workshops, pp 206–209

Paulsson K, Hubner M, Becker J (2006) Strategies to on-line failure recovery in self-adaptive systems based on dynamic and partial reconfiguration. In: First NASA/ESA conference on adaptive hardware and systems (AHS'06), pp 288–291

Pham K, Horta E, Koch D, Vaishnav A, Kuhn T (2018) IPRDF: an isolated partial reconfiguration design flow for Xilinx FPGAs. In: International symposium on embedded multicore/many-core systems-on-chip (MCSoC), pp 36–43

Pham TH, Fahmy SA, McLoughlin IV (2017) An end-to-end multi-standard OFDM transceiver architecture using FPGA partial reconfiguration. IEEE Access 5:21002–21015

Pham KD, Horta E, Koch D (2017) BITMAN: a tool and API for FPGA bitstream manipulations. In: Design, automation and test in Europe conference and exhibition (DATE), pp 894–897

Proulx A, Chouinard J-Y, Fortier P, Miled A (2023) A survey on FPGA cybersecurity design strategies. ACM Trans Reconfigurable Technol Syst 16(2):1–33

Rabozzi M, Durelli GC, Miele A, Lillis J, Santambrogio MD (2016) Floorplanning automation for partial-reconfigurable FPGAs via feasible placements generation. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(1):151–164

Reviriego P, Ullah A, Pontarelli S (2019) PR-TCAM: efficient TCAM emulation on Xilinx FPGAs using partial reconfiguration. IEEE Trans Very Large Scale Integr (VLSI) Syst 27(8):1952–1956

Rodríguez A, Valverde J, Portilla J, Otero A, Riesgo T, Torre E (2018) FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: the ARTICo3 framework. Sensors 18(6):1877

Rossi E, Damschen M, Bauer L, Buttazzo G, Henkel J (2018) Preemption of the partial reconfiguration process to enable real-time computing with FPGAs. ACM Trans Reconfigurable Technol Syst (TRETS) 11(2):1–24

Sadek A, Mostafa H, Nassar A Ismail Y (2017) Towards the implementation of multi-band multi-standard software-defined radio using dynamic partial reconfiguration. Int J Commun Syst 30(17):3342

Shreejith S, Vipin K, Fahmy SA, Lukasiewycz M (2013) An approach for redundancy in FlexRay networks using FPGA partial reconfiguration. In: Design, automation and test in Europe conference and exhibition (DATE), pp 721–724

Soni RK, Steiner N, French M (2013) Open-source bitstream generation. In: IEEE international symposium on field-programmable custom computing machines (FCCM), pp 105–112

Steiger C, Walder H, Platzner M (2004) Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. IEEE Trans Comput 53(11):1393–1407

Stitt G, Coole J (2011) Intermediate fabrics: virtual architectures for near-instant FPGA compilation. IEEE Embed Syst Lett 3(3):81–84

Stoddard A, Gruwell A, Zabriskie P, Wirthlin MJ (2016) A hybrid approach to FPGA configuration scrubbing. IEEE Trans Nucl Sci 64(1):497–503

Vaishnav A, Pham KD, Koch D (2018) A survey on FPGA virtualization. In: International conference on field programmable logic and applications (FPL), pp 131–138

Vaishnav A, Pham K, Powell J, Koch D (2020) Fos: a modular FPGA operating system for dynamic workloads. ACM Trans Reconfigurable Technol Syst 20:1–20:28

Vipin K, Fahmy SA (2012) Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In: Reconfigurable computing: architectures, tools and applications: international symposium on applied reconfigurable computing, pp 13–25

Vipin K, Fahmy SA (2012) A high speed open source controller for FPGA partial reconfiguration. In: International conference on field-programmable technology (FPT), pp 61–66

Vipin K, Fahmy SA (2013) Automated partitioning for partial reconfiguration design of adaptive systems. In: IEEE international symposium on parallel and distributed processing workshops, pp 172–181

Vipin K, Fahmy SA (2014) Automated partial reconfiguration design for adaptive systems with CoPR for Zynq. In: Proceedings of the international symposium on field-programmable custom computing machines (FCCM), pp 202–205

Vipin K, Fahmy SA (2014a) DyRACT: a partial reconfiguration enabled accelerator and test platform. In: International conference on field programmable logic and applications (FPL)

Vipin K, Fahmy SA (2014b) ZyCAP: efficient partial reconfiguration management on the Xilinx Zynq. IEEE Embed Syst Lett 6(3):41–44

Vipin K, Fahmy SA (2018) FPGA dynamic and partial reconfiguration: a survey of architectures, methods, and applications. ACM Comput Surv (CSUR) 51(4), 72:1–72:39

Venieris SI, Bouganis C-S (2017) Latency-driven design for FPGA-based convolutional neural networks. In: Proceedings of the international conference on field programmable logic and applications (FPL)

Vliegen J, Mentens N, Verbauwhede I (2013) A single-chip solution for the secure remote configuration of FPGAs using bitstream compression. In: International conference on reconfigurable computing and FPGAs (ReConFig), pp 1–6

Wirthlin M (2015) High-reliability FPGA-based systems: space, high-energy physics, and beyond. Proc IEEE 103(3):379–389

Xiao Y, Park D, Butt A, Giesen H, Han Z, Ding R, Magnezi N, Rubin R, DeHon A (2019) Reducing FPGA compile time with separate compilation for FPGA building blocks. In: 2019 international conference on field-programmable technology (ICFPT), pp 153–161. https://doi.org/10.1109/ICFPT47387.2019.00026

Xiao Y, Hota A, Park D, DeHon A (2022) HiPR: high-level partial reconfiguration for fast incremental FPGA compilation. In: 2022 32nd international conference on field-programmable logic and applications (FPL), pp 70–78

Xilinx (2023) UltraScale Architecture Configuration User Guide. https://docs.amd.com/v/u/en-US/ug570-ultrascale-configuration

Xilinx (2023) Abstract Shell for Dynamic Function eXchange. https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration/Abstract-Shell-for-Dynamic-Function-eXchange

Xilinx (2024) Using Encryption and Authentication to Secure an UltraScale/UltraScale+ FPGA Bitstream Application Note (XAPP1267). https://docs.xilinx.com/r/en-US/xapp1267-encryp-efuse-program/Using-Encryption-and-Authentication-to-Secure-an-UltraScale/UltraScale-FPGA-Bitstream-Application-Note